Journal on Data Semantics manuscript No.
(will be inserted by the editor)

On Modeling and Querying Concept Evolution

Siarhei Bykau · John Mylopoulos · Flavio Rizzolo · Yannis Velegrakis

Received: date / Accepted: date

Abstract Entities and the concepts they instantiate evolve over time. For example, a corporate entity may have resulted from a series of mergers and splits, or a concept such as that of Whale may have evolved along with our understanding of the physical world. We propose a model for capturing and querying concept evolution. Our proposal extends an RDF-like model with temporal features and evolution operators. In addition, we provide a query language that exploits these extensions and supports historical queries. Moreover, we study how evolution information can be exploited to answer queries that are agnostic to evolution details (hence, evolution-unaware). For these, we propose dynamic programming algorithms and evaluate their efficiency and scalability by experimenting with both real and synthetic datasets.

Keywords Evolution · Possible worlds · Steiner Trees · RDF · Query Answering

1 Introduction

Conceptual modeling languages – including the ER Model, UML class diagrams and Description Logics – are all founded on a notion of entity that represents a thing in the application domain. Although the state of an entity can change over its lifetime, entities themselves are uniformly treated as atomic and immutable. Unfortunately, this feature prevents existing modeling languages from capturing phenomena that involve the mutation of an entity into something else, such as a caterpillar becoming a butterfly, or the evolution of one entity into several (or vice versa), such as Germany splitting off into two Germanies right after WWII. The same difficulty arises when we try to model evolution at the class level. Consider the class Whale, which evolved over the past two centuries in the sense that whales were considered a species of fish, whereas today they are known to be mammals. This means that the concept of Whale-as-mammal has evolved from that of Whale-as-fish in terms of the properties we ascribe to whales (e.g., their reproductive system). Clearly, the (evolutionary) relationship between Whale-as-fish and Whale-as-mammal is of great interest to historians who may want to ask questions such as "Who studied whales in the past 3 centuries?" We are interested in modeling and querying such evolutionary relationships between entities and/or classes. In the sequel, we refer to both entities and classes as concepts.

In the database field, considerable amount of research effort has been spent on the development of models, techniques and tools for modeling and managing state changes for a concept, but no work has addressed the forms of evolution suggested above. These range from data manipulation languages, and maintenance of views under changes (Blakeley et al. 1986), to schema evolution (Lerner 2000) and mapping adaptation (Velegrakis et al. 2004; Kondylakis and Plexousakis 2010). To cope with the history of data changes, temporal models have been proposed for the relational (Soo 1991) and ER (Gregersen and Jensen 1999) models, for semi-structured data (Chawathe et al. 1999), XML (Rizzolo and Vaisman 2008) and for RDF (Gutiérrez et al. 2005). Almost in its entirety, existing work on data changes is based

Siarhei Bykau
University of Trento Via Sommarive 14, Trento, Italy
E-mail: bykau@disi.unitn.eu

John Mylopoulos
University of Trento Via Sommarive 14, Trento, Italy
E-mail: jm@disi.unitn.eu

Flavio Rizzolo
University of Ottawa 800 King Edward St., Ottawa, Canada
E-mail: flavio@cs.toronto.edu

Yannis Velegrakis
University of Trento Via Sommarive 14, Trento, Italy
E-mail: velgias@disi.unitn.eu

on a state-oriented point of view. It aims at recording and managing changes that are taking place at the values of the data.

In this work, we present a framework for modeling the evolution of concepts over time and the evolutionary relationships among them. The framework allows posing new kinds of queries that previously could not have been expressed. For instance, we aim at supporting queries of the form: How has a concept evolved over time? From what (or, to what) other concepts has it evolved? What other concepts have affected its evolution over time? What concepts are related to it through evolution and how? These kinds of queries are of major importance for interesting areas which are such as those discussed below.

History. Modern historians are interested in studying the history of humankind, and the events and people who shaped it. In addition, they want to understand how systems, tools, concepts of importance, and techniques have evolved throughout history. For them it is not enough to query a data source for a specific moment in history. They need to ask questions on how concepts and the relationships that exist between them have changed over time. For example, historians may be interested in the evolution of countries such as Germany, with respect to territory, political division or leadership. Alternatively, they may want to study the evolution of scientific concepts, e.g., how the concept of biotechnology has evolved from its humble beginnings as an agricultural technology to the current notion that is coupled to genetics and molecular biology.

Entity Management. Web application and integration systems are progressively moving from tuple and value-based towards entity-based solutions, i.e., systems where the basic data unit is an entity, independently of its logical representation (Dong et al. 2005). Furthermore, web integration systems, in order to achieve interoperability, may need to provide unique identification for the entities in the data they exchange (Palpanas et al. 2008). However, entities do not remain static over time: they get created, evolve, merge, split, and disappear. Knowing the history of each entity, i.e., how it has been formed and from what, paves the ground for successful entity management solutions and effective information exchange.

Life Sciences. One of the fields in Biology focuses on the study of the evolution of species from their origins and throughout history. To better understand the secrets of nature, it is important to model how the different species have evolved, from what, if and how they have disappeared, and when. As established by Darwin's theories, this evolution is very important for the understanding of how species came to be what they are now, or why some disappeared.

The contributions of this work are as follows: (i) We propose a conceptual model enhanced with the notion of a lifetime of a class, individual and relationship, (ii) We extend the temporal model proposed in (Gutiérrez et al. 2005) with consistency conditions and additional constructs to model merges, splits, and other forms of evolution among concepts; (iii) We introduce an evolution-aware query language that supports the answering of queries regarding the lifetime of concepts as well as the way they have evolved over time; (iv) We offer a formal semantics of query answering in the presence of evolution for queries that are evolution unaware; (v) We implement the semantics of this query answering by generating on-the-fly possible instances (similar to the concept of possible worlds in probabilistic databases) where multiple concepts associated through evolution relationships have been collapsed into one; (vi) We introduce the idea of finding a Steiner forest as a means of finding the optimal merges that need to be done to generate the possible instances and present indexing techniques for its efficient implementation; (vii) We present the architecture of the system we have developed that incorporates the above methods; (viii) We describe a case study involving concept evolution and perform a number of experiments for evaluating our evolution-unaware query answering mechanism.

Preliminary versions of these contributions have been presented in two earlier publications (Rizzolo et al. 2009; Bykau et al. 2011). This paper extends and refines our earlier work by the following contributions: (a) We present an integrative framework of the evolution management, i.e. both the modeling and query evaluation; (b) We describe the system architecture of the implementation of our research ideas; (c) We discuss the evolution discovery mechanism which is used to obtain a real data set for our experiments (the trademark dataset).

The rest of the paper is structured as follows. Section 2 describes the motivating examples used in this work. In Section 3 we present our modeling framework for evolution-aware queries. A formal semantics of query answering in the presence of evolution for queries that are evolution-unaware is discussed in Section 4. Section 5 presents the architecture of the system we have developed that incorporates the evolution-unaware evaluation methods. Two case studies which involve evolution are described in Section 6. The description of a number of experiments for evaluating our evolution-unaware query answering mechanism is shown in Section 7. The related work is discussed in Section 8. Finally, we conclude in Section 9.

2 Motivating Scenarios

Consider a historian database that records information about countries and their political governance. A fraction of the database modeling a part of the history of Germany is illustrated in Figure 1 (a). In it, the status of Germany at different times in history has been modeled through different individuals or through different concepts. In particular, Germany

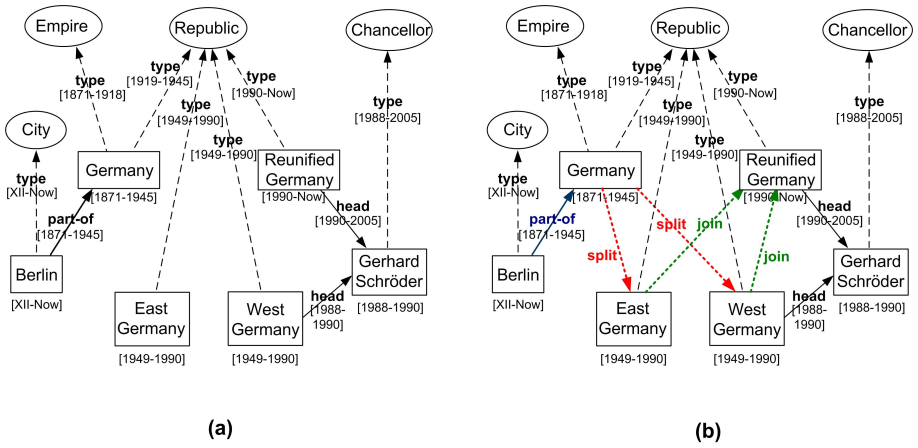


Fig. 1 The concepts of Germany, in a temporal model **(a)** and in our evolutionary model **(b)**

was an empire until 1918 and a republic from 1919 to 1945. This change is modeled by the multiple instantiations of Germany relative to Empire and Republic. Shortly after the end of the World War II, Germany was split in four zones[1] that in 1949 formed East and West Germany. These two entities lasted until 1990, when they were merged to form the republic of Germany as we know it today. This entity is modeled through the individual as Reunified Germany.

To model the validity constraints on the states of Germany during different periods, we use a temporal model similar to temporal RDF (Gutiérrez et al. 2005). The model associates to each concept a specific time frame. The time frames assigned to the individuals that model different Germanies are illustrated in Figure 1 through intervals. The same applies to every property, instance and subclass relationship. Note, for example, how the instantiation relationship of Germany to Empire has a temporal interval from 1871 to 1918, while the one to Republic has a temporal interval from 1918 to 1945. It is important for such a database to contain no inconsistencies, i.e., situations like having an instantiation relationship with a temporal interval bigger than the interval of the class it instantiates. Although temporal RDF lacks this kind of axiomatization, a mechanism for consistency checking needs to be in place for the accurate representation of concepts and individuals in history. Our solution provides an axiomatization of the temporal RDF that guarantees the consistency of the historical information recorded in a database.

Consider now the case of a historian who wants to study the history of Germany. The historian may be interested, for example, in all the leaders and constituents of Germany throughout its history. Using traditional query mechanisms, the historian will only be able to retrieve the individual Germany. Using keyword based techniques, she may be able to also retrieve the remaining individuals modeling Germany at different times, but this under the assumption that each such individual contains the keyword "Germany". Applying terminology evolution (Tahmasebi et al. 2008), it is possible to infer that the four terms for Germany, i.e., Germany, East Germany, West Germany, and Reunified Germany refer to related concepts regardless of the keywords that appear in the terms. Yet, in neither case the historian will be able to reconstruct how the constituents of Germany have changed over time. She will not be able to find that the East and West Germany were made by splitting the pre-war Germany and its parts, neither that East and West Germany were the same two that merged to form the Reunified Germany. We propose the use of explicit constructs to allow the modeling of evolution in databases, as exemplified in Figure 1(b) introducing split, join and part-of relationships.

Consider the case when the user cannot formulate queries using evolution constructs or if the conceptual model that the user has in mind in terms of evolution granularity is different from the one that actually exists in the data repository. Normally it will not be possible to answer any queries. To better illustrate the problem, let us consider the history of AT&T, a company that over the years has gone through a large number of break-ups, mergers and acquisitions. We consider a new evolution graph (see Figure 2) for evolution-unaware queries because it is more simple and has data peculiarities which allow us to demonstrate the details of our approach. The company 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 part-

[1] The information of the four zones is not illustrated in the Figure 1

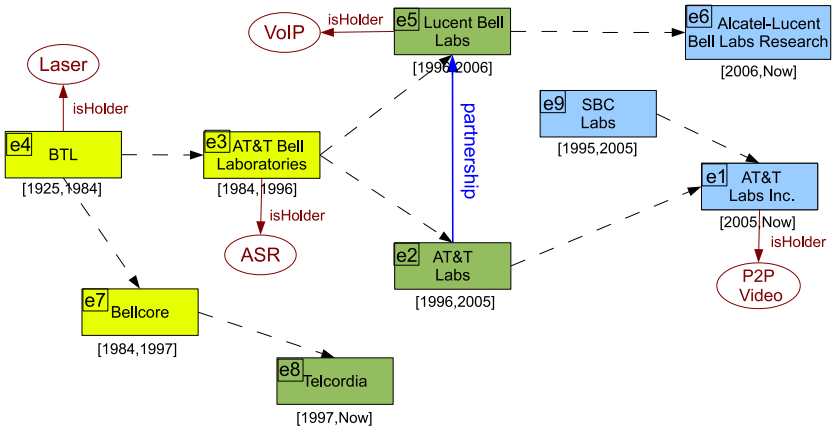


Fig. 2 The history of the AT&T Labs.

nership relationship. All these different labs have produced a large number of inventions, as the respective patents can demonstrate. Examples of such inventions are 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 2 where the labs are modeled by rectangles and the patents by ovals.

Assume now a temporal database that models the above information. For simplicity we do not distinguish among the different kinds of evolution relationships (split, merger, and so on). 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 how the concept has evolved 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 instead. 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 intend 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 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 Supporting Evolution-Aware Queries

This section studies queries which are agnostic to evolution details, namely the evolution-aware queries. In particular, first we introduce the temporal database (Section 3.1), second we describe the evolution modeling technique (Section 3.2), finally we provide a formal definition of the query language (Section 3.3) and briefly talk about its evaluation strategy (Section 3.4).

3.1 Temporal Databases

We consider an RDF-like data model. The model is expressive enough to represent ER models and the majority of ontologies and schemas that are used in practice (Lenzerini 2002). It does not include certain OWL Lite features such as *sameAs* or *equivalentClass*, since these features have been considered to be outside of the main scope of this work and their omission does not restrict the functionality of the model.

We assume the existence of an infinite set of *resources* $\mathcal{U}$, each with a *unique resource identifier* (URIs), and a set of *literals* $\mathcal{L}$. A *property* is a relationship between two resources. Properties are treated as resources themselves. We consider the existence of the special properties: rdfs:type, rdfs:domain, rdfs:range, rdfs:subClassOf and rdfs:subPropertyOf, which we refer to as type, dom, rng, subc, and subp, respectively. The set $\mathcal{U}$ contains three special resources: rdfs:Property, rdfs:Class and rdf:Thing, which we refer to as Prop, Class and Thing, respectively. The semantics of these resources as well as the semantics of the special properties are those defined in RDFS (W3C 2004). Resources are described by a set of triples that form a database.

Definition 1 A *database* Σ is a tuple $\langle U, L, T\rangle$, where $U \subseteq \mathcal{U}$, $L \subseteq \mathcal{L}$, $T \subseteq \mathcal{U}\times \mathcal{U} \times \{\mathcal{U} \cup \mathcal{L}\}$, and U contains the resources rdfs:Property, rdfs:Class, and rdf:Thing. The set of *classes* of the database Σ is the set $C=\{x \mid \exists\langle x, \mathsf{type}, \mathsf{rdfs{:}Class}\rangle \in T\}$. Similarly, the set of *properties* is the set $P=\{x \mid \exists\langle x, \mathsf{type}, \mathsf{rdfs{:}Property}\rangle \in T\}$. The set P contains the RDFS properties type, dom, rng, subc, and subp. A resource i is said to be an *instance* of a class $c \in C$ (or of type c) if $\exists\langle i, \mathsf{type}, c\rangle \in T$. The set of *instances* is the set $I=\{i \mid \exists\langle i, \mathsf{type}, y\rangle \in T\}$.

A database can be represented as a hypergraph called an *RDF graph*. In the rest of the paper, we will use the terms database and RDF graph equivalently.

Definition 2 An *RDF graph* of a database Σ is an hypergraph in which nodes represent resources and literals and the edges represent triples.

Example 1 Figure 1(a) is an illustration of an RDF Graph. The nodes Berlin and Germany represent resources. The edge labeled part-of between them represents the triple ⟨Berlin,part-of,Germany⟩. The label of the edge, i.e., part-of, represents a property.

To support the temporal dimension in our model, we adopt the approach of Temporal RDF (Gutiérrez et al. 2005) which extends RDF by associating to each triple a time frame. Unfortunately, this extension is not enough for our goals. We need to add time semantics not only to relationships between resources (what the triples represent), but also to resources themselves by providing temporal-varying classes and individuals. This addition and the consistency conditions we introduce below are our temporal extensions to the temporal RDF data model.

We consider time as a discrete, total order domain $\mathbb{T}$ in which we define different granularities. Following Dyreson et al. (2000), a *granularity* is a mapping from integers to *granules* (i.e., subsets of the time domain $\mathbb{T}$) such that contiguous integers are mapped to non-empty granules and granules within one granularity are totally ordered and do not overlap. Days and months are examples of two different granularities, in which each granule is a specific day in the former and a month in the latter. Granularities define a lattice in which granules in some granularities can be aggregated in larger granules in *coarser* granularities. For instance, the granularity of months is coarser than that of days because every granule in the former (a month) is composed of an integer number of granules in the latter (days). In contrast, months are not coarser (nor finer) than weeks.

Even though we model time as a point-based temporal domain, we use intervals as abbreviations of sets of instants whenever possible. An ordered pair $[a, b]$ of time points, with a, b granules in a granularity, and $a \leq b$, denotes the closed interval from a to b. As in most temporal models, the current time point will be represented with the distinguished word *Now*. We will use the symbol $\mathcal{T}$ to represent the infinite set of all the possible temporal intervals over the temporal domain $\mathbb{T}$, and the expressions $i.start$ and $i.end$ to refer to the starting and ending time points of an interval i. Given two intervals i_1 and i_2, we will denote by $i_1 \sqsubseteq i_2$ the containment relationship between the intervals in which $i_2.start \leq i_1.start$ and $i_1.end \leq i_2.end$.

Two types of temporal dimensions are normally considered: *valid* time and *transaction* time. Valid time is the time when data is valid in the modeled world whereas transaction time is the time when data is actually stored in the database. Concept evolution is based on valid time.

Definition 3 A *temporal database* Σ_T is a tuple $\langle U, L, T, \tau\rangle$, where $\langle U, L, T\rangle$ is a database and τ is function that maps every resource $r \in U$ to a temporal interval in $\mathcal{T}$. The temporal interval is also referred to as the *lifespan* of the resource. The expressions $r.start$ and $r.end$ denote the start and end points of the interval of r, respectively. The *temporal graph* of Σ_T is the RDF graph of $\langle U, L, T\rangle$ enhanced with the temporal intervals on the edges and nodes.

For a temporal database to be semantically meaningful, the lifespans of the resources need to satisfy certain conditions. For instance, it is not logical to have an individual with a lifespan that does not contain any common time points with the lifespan of the class it belongs to. Temporal RDF does not provide such a mechanism, thus, we are introducing the notion of a consistent temporal database.

Definition 4 A *consistent temporal database* is a temporal database $\Sigma_\tau=\langle U, L, T, \tau\rangle$ that satisfies the following conditions:

1. $\forall r \in L \cup \{\mathsf{Prop}, \mathsf{Class}, \mathsf{Thing}, \mathsf{type}, \mathsf{dom}, \mathsf{rng}, \mathsf{subc}, \mathsf{subp}\}$: $\tau(r)=[0, Now]$;
2. $\forall\langle d, p, r\rangle \in T : \tau(\langle d, p, r\rangle) \sqsubseteq \tau(d)$ and $\tau(\langle d, p, r\rangle) \sqsubseteq \tau(r)$;
3. $\forall\langle d, p, r\rangle \in T$ with $p \in \{\mathsf{type}, \mathsf{subc}, \mathsf{subp}\}$: $\tau(d) \sqsubseteq \tau(r)$.

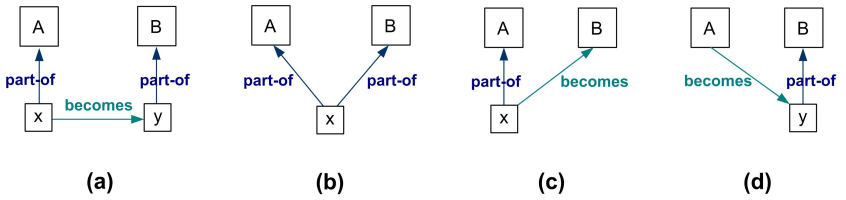


Fig. 3 Liaison examples

Intuitively, literals and the special resources and properties defined in RDFS need to be valid during the entire lifespan of the temporal database, which is $[0, Now]$ (Condition 1). In addition, the lifespan of a triple needs to be within the lifespan of the resources that the triple associates (Condition 2). Finally, the lifespan of a resource has to be within the lifespan of the class the resource instantiates, and any class or property needs to be within the lifespan of its superclasses or superproperties (Condition 3).

3.2 Modeling Evolution

Apart from the temporal dimension that was previously described, two new dimensions need to be introduced to successfully model evolution: the mereological and the causal.

Mereology (Keet and Artale 2008) is a sub-discipline in philosophy that deals with the ontological investigation of the part-whole relationship. It is used in our model to capture the parthood relationship between concepts in a way that is carried forward as concepts evolve. Such a relationship is modeled through the introduction of the special property part-of, which is reflexive, antisymmetric and transitive. A property part-of is defined from a resource x to a resource y if the concept modeled by resource x is part of the concept modeled by resource y. Note that the above definition implies that every concept is also a part of itself. When x is a part of y and $x \neq y$ we say that x is a *proper part* of y. Apart from this special semantics, part-of behaves as any other property in a temporal database. For readability and presentation reasons, we may use the notation $x \overset{\text{part-of}}{\longrightarrow} y$ to represent the existence of a triple $<x$,part-of,$y>$ in the set T of a temporal database Σ_τ.

To capture causal relationships, i.e., the interdependency between two resources, we additionally introduce the notion of becomes, which is an antisymmetric and transitive relation. For similar reasons as before, we use the notation $x \overset{\text{becomes}}{\longrightarrow} y$ to represent the fact that $(x, y) \in$ becomes. Intuitively, $x \overset{\text{becomes}}{\longrightarrow} y$ means that the concept modeled by resource y originates from the concept modeled by resource x. We require that $\tau(x).end < \tau(y).start$.

To effectively model evolution, we also need the notion of a *liaison*. A liaison between two concepts is another concept that keeps the former two linked together in time by means of part-of and becomes relationships. In other words, a liaison is part of at least one of the concepts it relates and has some causal relationship to a part of the other.

Definition 5 (Liaison) Let A, B be two concepts of a temporal database with $\tau(A).start < \tau(B).start$, and x, y concepts for which $x \overset{\text{part-of}}{\longrightarrow} A$ and $y \overset{\text{part-of}}{\longrightarrow} B$. A concept x (or y) is said to be a *liaison between A and B* if either $x \overset{\text{becomes}}{\longrightarrow} y$ or $x = y$.

The most general case of a liaison is graphically depicted in Figure 3 (a). The boxes A and B represent the two main concepts whereas the x and y represent two of their respective parts. Figure 3 (b) illustrates the second case of the definition in which x and y are actually the same concept. Figure 3 (c) (respectively, (d)) shows the special case in which y (respectively, x) is exactly the whole of B (respectively, A) rather than a proper part of it.

To model the different kinds of evolution events that may exist, we introduce four evolution terms: join, split, merge, and detach.

[join] The join term, denoted as $\text{join}(c_1 \ldots c_n, c, t)$, models the fact that every part of a concept c born at time t comes from a part of some concept in $\{c_1,\ldots,c_n\}$. In particular:

- $\tau(c).start{=}t$;
- $\forall x$ s.t. $x \overset{\text{part-of}}{\longrightarrow} c$: $\exists c_i$ s.t. x is a liaison between c_i and c, or $x = c_i$, with $1{\leq}i{\leq}n$.

[split] The split term, denoted as $\text{split}(c, c_1 \ldots c_n, t)$, models the fact that every part of a concept c ending at time t becomes the part of some concept in $\{c_1,\ldots,c_n\}$. In particular:

- $\tau(c).end{=}t$;
- $\forall x$ s.t. $x \overset{\text{part-of}}{\longrightarrow} c$: $\exists c_i$ s.t. x is a liaison between c and c_i, or $x = c_i$, with $1{\leq}i{\leq}n$.

[merge] The merge term, denoted as $\text{merge}(c, c', t)$, models the fact that at least a part of a concept c ending at a time t becomes part of an existing concept c'. In particular:

- $\tau(c).end{=}t$;
- $\exists x$ s.t. $x \overset{\text{part-of}}{\longrightarrow} c'$ and x is a liaison between c and c'.

[detach] The detach term, denoted as $\text{detach}(c, c', t)$, models the fact that the new concept c' is formed at a time t with at least one part from c. In particular:

- $\tau(c').start{=}t$;

– $\exists x$ s.t. $x \xrightarrow{\text{part-of}} c$ and x is a liaison between c and c'.

Note that in each evolution term there is only one concept whose lifespan has necessarily to start or end at the time of the event. For instance, we could use a join to represent the fact that different countries joined the European Union (EU) at different times. The information of the period in which each country participated in the EU is given by the interval of each respective part-of property.

We record the becomes relation and the evolution terms in the temporal database as *evolution triples* $\langle c, term, c' \rangle$, where $term$ is one of the special *evolution properties* becomes, join, split, merge, and detach. Evolution properties are *meta-temporal*, i.e., they describe how the temporal model changes, and thus their triples do not need to satisfy the consistency conditions in Definition 4. A temporal database with a set of evolution properties and triples defines an *evolution base*.

Definition 6 An *evolution base* Σ_T^E is a tuple $\langle U, L, T, E, \tau \rangle$, where $\langle U, L, T, \tau \rangle$ is a temporal database, U contains a set of evolution properties, and E is a set of evolution triples. The *evolution graph* of Σ_T^E is the temporal graph of $\langle U, L, T, \tau \rangle$ enhanced with edges representing the evolutions triples.

The time in which the evolution event took place does not need to be recorded explicitly in the triple since it can be retrieved from the lifespan of the involved concepts. For instance, detach(Kingdom of the Netherlands, Belgium, 1831) is modeled as the triple: ⟨Kingdom of the Netherlands, detach, Belgium⟩ with $\tau(\text{Belgium}).start = 1831$.

For recording evolution terms that involve more than two concepts, e.g. the join, multiple triples are needed. We assume that the terms are indexed by their time, thus, the set of (independent) triples that belong to the same terms can be easily detected since they will all share the same start or end time in the lifespan of the respective concept. For instance, split(Germany, East Germany, West Germany, 1949) is represented in our model through the triples ⟨Germany, split, East Germany⟩ and ⟨Germany, split, West Germany⟩ with $\tau(\text{East Germany}).start = \tau(\text{West Germany}).start = 1949$.

Note that the evolution terms may entail facts that are not explicitly represented in the database. For instance, the split of Germany into West and East implies the fact that Berlin, which is explicitly defined as part of Germany, becomes part of either East or West. This kind of reasoning is beyond the scope of the current work.

3.3 Query Language

To support evolution-aware querying we define a navigational query language to traverse temporal and evolution edges in an evolution graph. This language is analogous to nSPARQL (Pérez et al. 2008), a language that extends SPARQL with navigational capabilities based on nested regular expressions. nSPARQL uses four different axes, namely **self**, **next**, **edge**, and **node**, for navigation on an RDF graph and node label testing. We have extended the nested regular expressions constructs of nSPARQL with temporal semantics and a set of five *evolution axes*, namely **join**, **split**, **merge**, **detach**, and **becomes** that extend the traversing capabilities of nSPARQL to the evolution edges. The language is defined according to the following grammar:

$$exp := \mathbf{axis} \mid \mathbf{t\text{-}axis} :: a \mid \mathbf{t\text{-}axis} :: [exp] \mid exp[I] \mid exp/exp \mid exp|exp \mid exp^*$$

where a is a node in the graph, I is a time interval, and **axis** can be either **forward**, **backward**, **e-edge**, **e-node**, a **t-axis** or an **e-axis**, with **t-axis** $\in$ {**self**, **next**, **edge**, **node**} and **e-axis** $\in$ {**join**, **split**, **merge**, **detach**, **becomes**}.

The evaluation of an evolution expression exp is given by the semantic function $\mathcal{E}$ defined in Figure 4. $\mathcal{E}[\![exp]\!]$ returns a set of tuples of the form $\langle x, y, I \rangle$ such that there is a path from x to y satisfying exp during interval I. For instance, in the evolution base of Figure 1, $\mathcal{E}[\![\mathbf{self} :: \text{Germany}/\mathbf{next} :: \text{head}/\mathbf{next} :: \text{type}]\!]$ returns the tuple ⟨Germany, Chancellor, $[1988, 2005]$⟩. It is also possible to navigate an edge from a node using the **edge** axis and to have a nested expression $[exp]$ that functions as a predicate which the preceding expression must satisfy. For example, $\mathcal{E}[\![\mathbf{self}[\mathbf{next} :: \text{head}/\mathbf{self} :: \text{Gerhard Schröder}]]\!]$ returns ⟨Reunified Germany, Reunified Germany, $[1990, 2005]$⟩ and ⟨West Germany, West Germany, $[1988, 1990]$⟩.

In order to support evolution expressions, we need to extend nSPARQL triple patterns with temporal and evolution semantics. In particular, we redefine the evaluation of an nSPARQL triple pattern $(?X, exp, ?Y)$ to be the set of triples $\langle x, y, I \rangle$ that result from the evaluation of the evolution expression exp, with the variables X and Y bound to x and y, respectively. In particular:

$$\mathcal{E}[\![(?X, exp, ?Y)]\!] := \{(\theta(?X), \theta(?Y)) \mid \theta(?X) = x \text{ and } \theta(?Y) = y \text{ and } \langle x, y, I \rangle \in \mathcal{E}[\![exp]\!]\}$$

Our language includes all nSPARQL operators such as AND, OPT, UNION and FILTER with the same semantics as in nSPARQL. For instance:

$$\mathcal{E}[\![(P_1 \text{ AND } P_2)]\!] := \mathcal{E}[\![(P_1)]\!] \bowtie \mathcal{E}[\![(P_2)]\!]$$

where P_1 and P_2 are triple patterns and $\bowtie$ is the join on the variables P_1 and P_2 have in common. A complete list of all the nSPARQL operators and their semantics is given by (Pérez et al. 2008).

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathbf{self}]\!] & := & \{\langle x, x, \tau(x)\rangle \mid x \in U \cup L\} \\
\mathcal{E}[\![\mathbf{self}::r]\!] & := & \{\langle r, r, \tau(r)\rangle\} \\
\mathcal{E}[\![\mathbf{next}]\!] & := & \{\langle x, y, \tau(t)\rangle \mid t = \langle x, z, y\rangle \in T\} \\
\mathcal{E}[\![\mathbf{next}::r]\!] & := & \{\langle x, y, \tau(t)\rangle \mid t = \langle x, r, y\rangle \in T\} \\
\mathcal{E}[\![\mathbf{edge}]\!] & := & \{\langle x, y, \tau(t)\rangle \mid t = \langle x, y, z\rangle \in T\} \\
\mathcal{E}[\![\mathbf{edge}::r]\!] & := & \{\langle x, y, \tau(t)\rangle \mid t = \langle x, y, r\rangle \in T\} \\
\mathcal{E}[\![\mathbf{node}]\!] & := & \{\langle x, y, \tau(t)\rangle \mid t = \langle z, x, y\rangle \in T\} \\
\mathcal{E}[\![\mathbf{node}::r]\!] & := & \{\langle x, y, \tau(t)\rangle \mid t = \langle r, x, y\rangle \in T\} \\
\mathcal{E}[\![\mathbf{e\text{-}edge}]\!] & := & \{\langle x, \mathbf{e\text{-}axis}, [0, Now]\rangle \mid t = \langle x, \mathbf{e\text{-}axis}, z\rangle \in E\} \\
\mathcal{E}[\![\mathbf{e\text{-}node}]\!] & := & \{\langle \mathbf{e\text{-}axis}, y, \tau(t)\rangle \mid t = \langle z, \mathbf{e\text{-}axis}, y\rangle \in E\} \\
\mathcal{E}[\![\mathbf{e\text{-}axis}]\!] & := & \{\langle x, y, [0, Now]\rangle \mid \exists\, t = \langle x, \mathbf{e\text{-}axis}, y\rangle \in E\} \\
\mathcal{E}[\![\mathbf{forward}]\!] & := & \bigcup_{\mathbf{e\text{-}axis}} \mathcal{E}[\![\mathbf{e\text{-}axis}]\!] \\
\mathcal{E}[\![\mathbf{backward}]\!] & := & \bigcup_{\mathbf{e\text{-}axis}} \mathcal{E}[\![\mathbf{e\text{-}axis}^{-1}]\!] \\
\mathcal{E}[\![\mathbf{self}::[exp]]\!] & := & \{\langle x, x, \tau(x) \cap I\rangle \mid x \in U \cup L, \exists \langle x, z, I\rangle \in \mathcal{P}[\![exp]\!], \tau(x) \cap I \neq \emptyset\} \\
\mathcal{E}[\![\mathbf{next}::[exp]]\!] & := & \{\langle x, y, \tau(t) \cap I\rangle \mid t = \langle x, z, y\rangle \in T, \exists \langle z, w, I\rangle \in \mathcal{P}[\![exp]\!], \tau(t) \cap I \neq \emptyset\} \\
\mathcal{E}[\![\mathbf{edge}::[exp]]\!] & := & \{\langle x, y, \tau(t) \cap I\rangle \mid t = \langle x, y, z\rangle \in T, \exists \langle z, w, I\rangle \in \mathcal{P}[\![exp]\!], \tau(t) \cap I \neq \emptyset\} \\
\mathcal{E}[\![\mathbf{node}::[exp]]\!] & := & \{\langle x, y, \tau(t) \cap I\rangle \mid t = \langle z, x, y\rangle \in T, \exists \langle z, w, I\rangle \in \mathcal{P}[\![exp]\!], \tau(t) \cap I \neq \emptyset\} \\
\mathcal{E}[\![\mathbf{axis}^{-1}]\!] & := & \{\langle x, y, \tau(t)\rangle \mid \langle y, x, \tau(t)\rangle \in \mathcal{E}[\![\mathbf{axis}]\!]\} \\
\mathcal{E}[\![\mathbf{t\text{-}axis}^{-1}::r]\!] & := & \{\langle x, y, \tau(t)\rangle \mid \langle y, x, \tau(t)\rangle \in \mathcal{E}[\![\mathbf{t\text{-}axis}::r]\!]\} \\
\mathcal{E}[\![\mathbf{t\text{-}axis}^{-1}::[exp]]\!] & := & \{\langle x, y, \tau(t)\rangle \mid \langle y, x, \tau(t)\rangle \in \mathcal{E}[\![\mathbf{t\text{-}axis}::[exp]]\!]\} \\
\mathcal{E}[\![exp[I]]\!] & := & \{\langle x, y, I \cap I'\rangle \mid \langle x, y, I'\rangle \in \mathcal{E}[\![exp]\!] \text{ and } I \cap I' \neq \emptyset\} \\
\mathcal{E}[\![exp/e\text{-}exp]\!] & := & \{\langle x, y, I_2\rangle \mid \exists \langle x, z, I_1\rangle \in \mathcal{E}[\![exp]\!], \exists \langle z, y, I_2\rangle \in \mathcal{E}[\![e\text{-}exp]\!]\} \\
\mathcal{E}[\![exp/t\text{-}exp]\!] & := & \{\langle x, y, I_1 \cap I_2\rangle \mid \exists \langle x, z, I_1\rangle \in \mathcal{E}[\![exp]\!], \exists \langle z, y, I_2\rangle \in \mathcal{E}[\![t\text{-}exp]\!] \text{ and} \\
 & & I_1 \cap I_2 \neq \emptyset\} \\
\mathcal{E}[\![exp_1|exp_2]\!] & := & \mathcal{E}[\![exp_1]\!]\ cup \mathcal{E}[\![exp_2]\!] \\
\mathcal{E}[\![exp^*]\!] & := & \mathcal{E}[\![\mathbf{self}]\!] \cup \mathcal{E}[\![exp]\!] \cup \mathcal{E}[\![exp/exp]\!] \cup \mathcal{E}[\![exp/exp/exp]\!] \cup \ldots \\
 & & \\
\mathcal{P}[\![e\text{-}exp]\!] & := & \mathcal{E}[\![e\text{-}exp]\!] \\
\mathcal{P}[\![t\text{-}exp]\!] & := & \mathcal{E}[\![t\text{-}exp]\!] \\
\mathcal{P}[\![t\text{-}exp/exp]\!] & := & \{\langle x, y, I_1 \cap I_2\rangle \mid \exists \langle x, z, I_1\rangle \in \mathcal{E}[\![t\text{-}exp]\!], \exists \langle z, y, I_2\rangle \in \mathcal{E}[\![exp]\!] \text{ and} \\
 & & I_1 \cap I_2 \neq \emptyset\} \\
\mathcal{P}[\![e\text{-}exp/exp]\!] & := & \{\langle x, y, I_1\rangle \mid \exists \langle x, z, I_1\rangle \in \mathcal{E}[\![e\text{-}exp]\!], \exists \langle z, y, I_2\rangle \in \mathcal{E}[\![exp]\!]\} \\
\mathcal{P}[\![exp_1|exp_2]\!] & := & \mathcal{E}[\![exp_1|exp_2]\!] \\
\mathcal{P}[\![exp^*]\!] & := & \mathcal{E}[\![exp^*]\!] \\
 & & \\
t\text{-}exp & \in & \{\mathbf{t\text{-}axis}, \mathbf{t\text{-}axis}::r, \mathbf{t\text{-}axis}::[exp], \mathbf{t\text{-}axis}[I]\} \\
e\text{-}exp & \in & \{\mathbf{e\text{-}axis}, \mathbf{e\text{-}axis}::[exp], \mathbf{e\text{-}axis}[I], \mathbf{forward}, \mathbf{backward}\}
\end{array}
$$

Fig. 4 Formal semantics of nested evolution expressions

3.4 Query Evaluation

The query language presented in the previous section is based on the concepts of nSPARQL and can be implemented as an extension of it by creating the appropriate query rewriting procedures that implement the semantics of Figure 4. Since our focus in query evaluation strategies is mainly on the evolution-unaware queries, we will not elaborate further on this.

4 Supporting Evolution-Unaware Queries

In this section we discuss the evolution-unaware queries. In particular, we present the model of evolution in Section 4.1, then we show a number of query evaluation techniques starting from naive ways and ending up with our solution (Section 4.2). In Section 4.3 we introduce the Steiner forest algorithm, which constitutes the core of our evaluation strategy, along with an optimization technique.

4.1 Modeling Evolution

To support queries that are unaware of the evolution relationships we need to construct a mechanism that performs various kinds of reasoning in a way transparent to the user. This reasoning involves the consideration of a series of data structures associated through the evolution relationships as one unified concept. For simplicity of the presentation, and also to abstract from the peculiarities of RDF, in what follows we use a *concept model* (Dalvi et al. 2009) as our data model. Furthermore, we do not consider separately the different kinds of evolution events but we consider them all under one unified relationship that we call *evolve*. This allows us to focus on different aspects of our proposal without increasing its complexity.

The fundamental component of the model is the *concept*, which is used to model a real world object. A concept 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 a concept identi-

fier. More formally, assume the existence of an infinite set of concept identifiers $\mathcal{O}$, an infinite set of names $\mathcal{N}$ and an infinite set of atomic values $\mathcal{V}$.

Definition 7 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. A *concept* 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 *concept identifier* while the set A as the set of *attributes* of the concept.

We will use the symbol $\mathcal{E}$ to denote the set of all possible concepts that exist and we will also assume the existence of a Skolem function Sk (Hull and Yoshikawa 1990). Recall that a Skolem function is a function that guarantees that different arguments are assigned different values. Each concept is uniquely identified by its identifier, thus, we will often use the terms *concept* and *concept identifier* interchangingly if there is no risk of confusion. A *database* is a collection of concepts, that is closed in terms of associations between the concepts.

Definition 8 A *database* is a finite set of concepts $E\subseteq\mathcal{E}$ such that for each association $\langle n, e'\rangle$ of a concept $e\in E$: $e'\in E$.

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 a concept 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 a concept 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 concept 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 2 Consider the query:

$\$x(isHolder{:}\$y)$:-
  $\$x(name{:}'AT\&TLabsInc.', isHolder{:}\$y)$

that looks for concepts called "AT&T Labs Inc." and are holders of a patent. For every such concept that is found, a concept 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 concepts represent and the evolution relationship between them. For the former, we assume that we have a temporal database, i.e., each concept is associated to a time period (see Section 3.1); however, this is not critical for the evolution-unaware queries so we will omit that part from the following discussions. To model the evolution relationship for evolution-unaware queries 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 concept 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. Note that the whole family of evolution operators from Section 3.2 is reduced to only one relationship.

In the next, with the abuse of notation we re-introduce the notion of evolution database with respect to the evolution-unaware queries. This allows us to focus only on the parts of our data model which are essential to the evolution-unaware queries. In Figure 2, the dotted lines between the concepts illustrate evolution relationships. A database with evolution relationships is an *evolution database*.

Definition 9 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 concepts and as edges its evolution relationships. We refer to this graph as the *evolution graph* of the database.

Our proposal is that concepts 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 concepts that are connected through a series of evolution relationships; the coalescence of those concepts is a new concept that replaces them and has as attributes the union of their attributes (including associations).

Definition 10 Given an evolution database $\langle E,\Omega\rangle$, The *coalescence* of two concepts $e_1{:}\langle id_1,A_1\rangle$, $e_2{:}\langle id_2,A_2\rangle\in E$, 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 concept 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 concept $e\in E$, 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 concepts. We call that order *subsumption*.

Definition 11 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 $id=Sk(id_1, id_x)$. A concept $e_1=\langle id_1,\ A_1\rangle$ is said to be *subsummed* by a concept $e_2=\langle id_2,\ A_2\rangle$, 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 concepts that an evolution relationship in the Ω_s associates.

Definition 12 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 concepts in a database that should be considered as one, while the possible world represents the database in which these concepts have actually been coalesced. Our notion of a possible world is similar to the notion of a possible worlds in probabilistic databases (Dalvi and Suciu 2007). Theorem 1 is a direct consequence of the definition of a possible world.

Theorem 1 *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 13 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.

Definition 14 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 concepts 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 concept is actually an evolution of the first.

Example 3 Consider again the query of Example 2 and assume that it is to be evaluated on the database of Figure 2. 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.

4.2 Query Evaluation

In this section we present evaluation strategies for the evolution-unaware queries, namely the naive approach (Section 4.2.1), the materializing all the possible worlds method (Section 4.2.2), the materializing only the maximum world approach(Section 4.2.3) and, finally, the on-the-fly coalescence computations (Section 4.2.4).

4.2.1 The naive approach

The straightforward 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 brute force solution, not desirable for online query answering.

4.2.2 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 (Blakeley et al. 1986) 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 concepts. Given that the number of attributes are typically larger than the number of concepts, and that concepts associated with evolution relationships are far fewer than the total number of concepts 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.

4.2.3 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 a concept (phases that may have to be considered individual concepts) but the approach coalesces them in one just because they are connected through evolution relationships.

4.2.4 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 concepts to which they belong. If all the attributes satisfying the conditions are on the same concept, then the concept is added in the answer set. However, different query conditions may be satisfied by attributes in different concepts. In these cases we identify sets of concepts for each one of which the union of the attributes of its concepts satisfy all the query conditions. For each such a set, we coalesce all its concepts 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 $L_1, L_2, \ldots, L_N$ 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 concept-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 concepts 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 concepts in V_x. To do so, these concepts 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 concepts in a set V_i, for i=1..k, are coalesced to the same concept. 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 (Gassner 2010),

Example 4 Let us consider again the query of Example 2. 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 concept 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. Concepts $e1$, $e2$ 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. ∎

4.3 Steiner Forest Algorithm

The last step of the evaluation algorithm for evolution unaware query language takes as input a set of concept sets and needs to perform a series of coalesce operations such that all the concepts within each set will become one. To do so, it needs to find an interconnect on the evolution graph among all the concepts within each set. Note that the interconnect may involve additional concepts 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 concepts 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 concepts, this is known as the problem of finding the Steiner tree (Dreyfus and Wagner 1972). However, given a set of sets of concepts, it turns out that finding the optimal solution, i.e., the minimum cost interconnect of all the concepts, 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 (Gassner 2010).

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

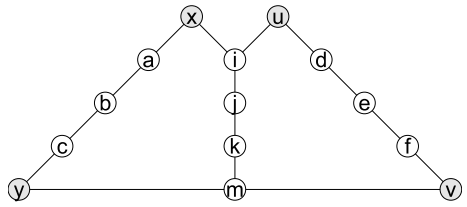


Fig. 5 An illustration of the Steiner forest problem.

forest problem takes as input set 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 5 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 nodes a, b and c. Similarly the minimum cost branch that connects u and v is the one that goes through 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 four nodes x, y, u and v through 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.

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 (Bhalotia et al. 2002; Kacholia et al. 2005; He et al. 2007) as well as a number of exact solution using Dynamic Programming (Dreyfus and Wagner 1972; Ding et al. 2007; Kimelfeld and Sagiv 2006) for the discovery of Steiner trees. However, for the Steiner forest problem (which is known to be NP-hard (Gassner 2010)) although there are approximate solutions (Gassner 2010), 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, $flat(\mathcal{V})= \{U \mid U=\bigcup_{V_i\in S} V_i$ 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., $maxflat(\mathcal{V})=\{n \mid n\in V_1\cup\ldots\cup V_L\}$.

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 $flat(\mathcal{V})$
Q_T: priority queue sorted in the increasing order
$Q_T \Leftarrow \emptyset$
for all $s_i \in maxflat(\mathcal{V})$ **do**
  enqueue $T(s_i,\{s_i\})$ into Q_T;
end for
while $Q_T \neq \emptyset$ **do**
  dequeue Q_T to $T(v,\mathbf{p})$;
  if $\mathbf{p} \in flat(\mathcal{V})$ **then**
    $ST(\mathbf{p}) = T(v,\mathbf{p})$
  end if
  if ST has all values **then**
    return ST
  end if
  for all $u \in N(v)$ **do**
    if $T(v,\mathbf{p}) \oplus (v,u) < T(u,\mathbf{p})$ **then**
      $T(u,\mathbf{p}) \Leftarrow T(v,\mathbf{p}) \oplus (v,u)$;
      update Q_T with the new $T(u,\mathbf{p})$;
    end if
  end for
  $\mathbf{p}_1 \Leftarrow \mathbf{p}$;
  for all $\mathbf{p}_2$ s.t. $\mathbf{p}_1 \cap \mathbf{p}_2 = \emptyset$ **do**
    if $T(v,\mathbf{p}_1) \oplus T(v,\mathbf{p}_2) < T(v,\mathbf{p}_1 \cup \mathbf{p}_2)$ **then**
      $T(u,\mathbf{p}_1 \cup \mathbf{p}_2) \Leftarrow T(v,\mathbf{p}_1) \oplus T(v,\mathbf{p}_2)$;
      update Q_T with the new $T(u,\mathbf{p}_1 \cup \mathbf{p}_2)$;
    end if
  end for
end while

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 (Ding et al. 2007). 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.

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

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: $SF(\mathcal{V}) \Leftarrow u$

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

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

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 (Ding et al. 2007) 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, 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 1 *Each branch of a Steiner forest is a Steiner tree.*

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:

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

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 $O(2^{\sum l_i}n + 2^L)$.

4.3.1 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 2 *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 $V'' \setminus V'$

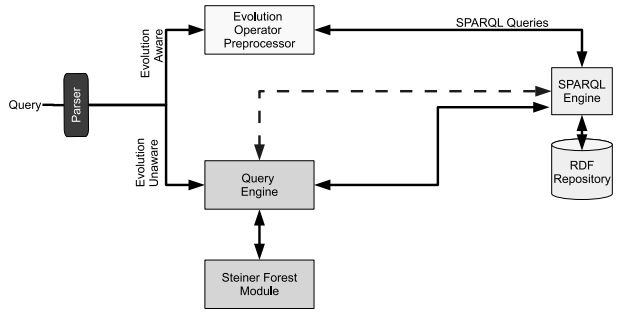


Fig. 6 The TrenDS System Architecture

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 2 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.

5 System Implementation

We have built a system in order to materialize the ideas described previously and see them in practice. The system has been implemented in Java, is called `TrenDS`, and the architecture of which is illustrated in Figure 6. It consists of four main components. One is the data repository for which it uses an RDF storage. The RDF storage has a SPARQL interface through which the data can be queried. It has no specific requirements thus, it can be easily replaced by any other RDF storage engine. The evolution relationships among the concepts are modeled as RDF attributes.

Once a query is issued to the system, it is first parsed by the *Query Parser*. The parser checks whether the query contains evolution related operators, in which case it forwards it to the *Evolution Operator Processor* module. The model is responsible for the implementation of the semantics of the evolution expressions as described in Figure 4. To achieve this it rewrites the query into a series of queries that contain no evolution operators, thus, they can be sent for execution to the repository. The results are collected back and sent to the user or the application that asked the query.

In the case in which we deal with the evolution-unaware queries, it is then sent to the *Query Engine* module. This module is responsible for implementing the whole evaluation procedure described in Section 4.2.4. In particular it first asks the repository and retrieves all the concepts that satisfy at least one of the query conditions (dotted line in Figure 6). Then it calls the *Steiner Forest* module to compute the different ways they can be coalesced in order to produce on-the-fly possible worlds. Finally, for each such a world, the results are generated and returned, alongside any additional fields that need to be retrieved by the repository. In the case in which only the top-k results are to be retrieved, the module activates the optimization algorithm described in Section 4.3.1 and prunes the number of Steiner forests that are to be computed.

6 Case Studies

We have performed two main case studies of modeling and querying evolution. Consider an evolution database which is an extension of the example introduced in Section 1 that models how countries have changed over time in terms of territory, political division, type of government, and other characteristics. Classes are represented by ovals and instances by boxes. A small fragment of that evolution base is illustrated as a graph in Figure 7.

Germany, for instance, is a concept that has changed several times along history. The country was unified as a nation-state in 1871 and the concept of Germany first appears in our historical database as Germany at instant 1871. After WWII, Germany was divided into four military zones (not shown in the figure) that were merged into West and East Germany in 1949. This is represented with two split edges from the concept of Germany to the concepts of West Germany and East Germany. The country was finally reunified in 1990, which is represented by the coalescence of the West Germany and East Germany concepts into Unified Germany via two merge edges. These merge and split constructs are also defined in terms of the parts of the concepts they relate. For instance, a part-of property indicates that Berlin was part of Germany during $[1871, 1945]$. Since that concept of Germany existed until 1945 whereas Berlin exists until today, the part-of relation is carried forward by the semantics of split and merge into the concept of Reunified Germany. Consider now a historian who is interested in finding answers to a number of evolution-aware queries.

[Example Query 1]: How has the notion of Germany changed over the last two centuries in terms of its constituents, government, etc.? The query can be expressed in our extended query language as follows:

$$\textbf{Select } ?Y, ?Z, ?W$$
$$(?X, \textbf{self} :: \text{Reunified Germany}/$$
$$\textbf{backward}^*[1800, 2000]/, ?Y) \text{ AND}$$
$$(?Y, \textbf{edge}, ?Z) \text{ AND } (?Z, \textbf{edge}, ?W)$$

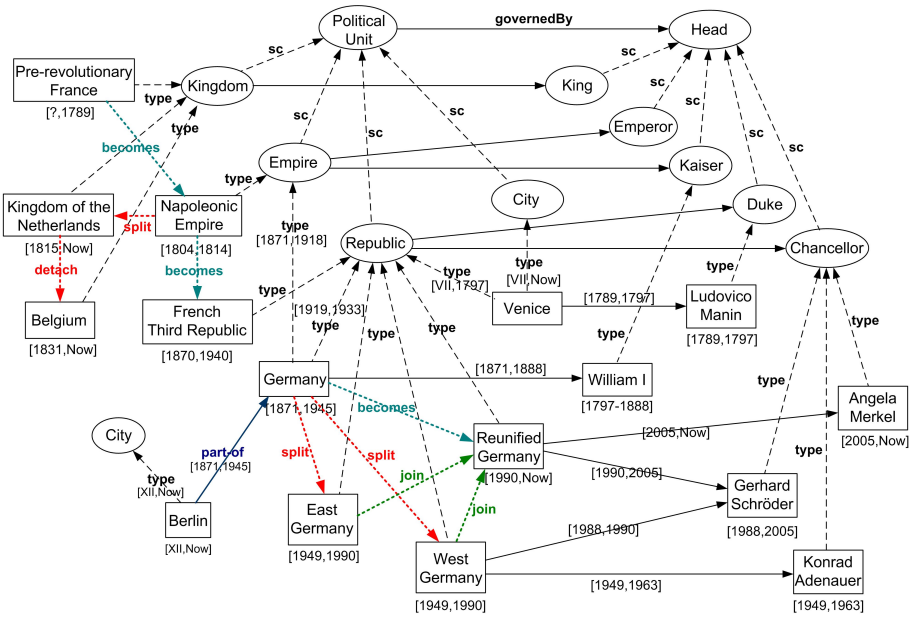


Fig. 7 The evolution of the concepts of Germany and France and their governments (full black lines represent governedBy properties)

The query first binds $?X$ to Reunified Germany and then follows all possible evolution axes backwards in the period $[1800, 2000]$. All concepts bound to $?Y$ are in an evolution path to Reunified Germany, namely Germany, West Germany, and East Germany. Note that, since the semantics of an * expression includes **self** (see Figure 4), then Reunified Germany will also bind $?Y$. The second triple returns in $?Z$ the name of the properties of which $?Y$ is the subject, and finally the last triple returns in $?W$ the objects of those properties. By selecting $?Y, ?Z, ?W$ in the head of the query, we get all evolutions of Germany together with their properties.

[Example Query 2]: Who was the head of the German government before and after the unification of 1990? The query can be expressed as follows:

$$\begin{aligned}&\textbf{Select } ?Y\\&(?X, \textbf{self} ::\text{Reunified Germany}/\textbf{join}^{-1}[1990]/\\&\textbf{next} :: \text{head}[1990], ?Y) \text{ AND}\\&(?Z, \textbf{self} ::\text{Reunified Germany}/\textbf{next} :: \text{head}[1990], ?Y)\end{aligned}$$

The first triple finds all the heads of state of the Reunified Germany before the unification by following $\textbf{join}^{-1}[1990]$ and then following $\textbf{next} :: \text{head}[1990]$. The second triple finds the heads of state of the Reunified Germany. Finally, the join on $?Y$ will bind the variable only to those heads of state that are the same in both triples, hence returning the one before and after the mentioned unification.

Consider now the evolution of the concept of biotechnology from a historical point of view. According to historians, biotechnology got its current meaning (related to molecular biology) only after the 70s. Before that, the term biotechnology was used in areas as diverse as agriculture, microbiology, and enzyme-based fermentation. Even though the term "biotechnology" was coined in 1919 by Karl Ereky, a Hungarian engineer, the earliest mentions of biotechnology in the news and specialized media refer to a set of ancient techniques like selective breeding, fermentation and hybridization. From the 70s the dominant meaning of biotechnology has been closely related to genetics. However, it is possible to find news and other media articles from the 60s to the 80s that use the term biotechnology to refer to an environmentally friendly technological orientation unrelated to genetics but closely related to bioprocess engineering. Not only the use of the term changed from the 60s to the 90s, but also the two different meanings coexisted in the media for almost two decades.

Figure 8 illustrates the evolution of the notion of biotechnology since the 40s. As in the previous example, classes in the evolution base are represented by ovals and instances by boxes. The used-for property is a normal property that simply links a technological concept to its products. The notions of Selective breeding, Fermentation and Hybridization existed from an indeterminate time until now and in the 40s joined the new topic of Conventional Biotech, which groups ancient techniques like the ones mentioned above. Over the next decades, Conventional Biotech started to include more modern therapies and products such as Cell Therapies, Penicillin and Cortisone. At some point in the 70s, the notions of Cell Therapies and Bioprocess Engineering matured and detached from Conventional Biotech becoming independent concepts. Note that Cell Therapies is a class-level concept that detached from the an instance-level concept. The three concepts coexisted in time during part of the 70s, the latter two coexist even now. During the 70s, the notion of Con-

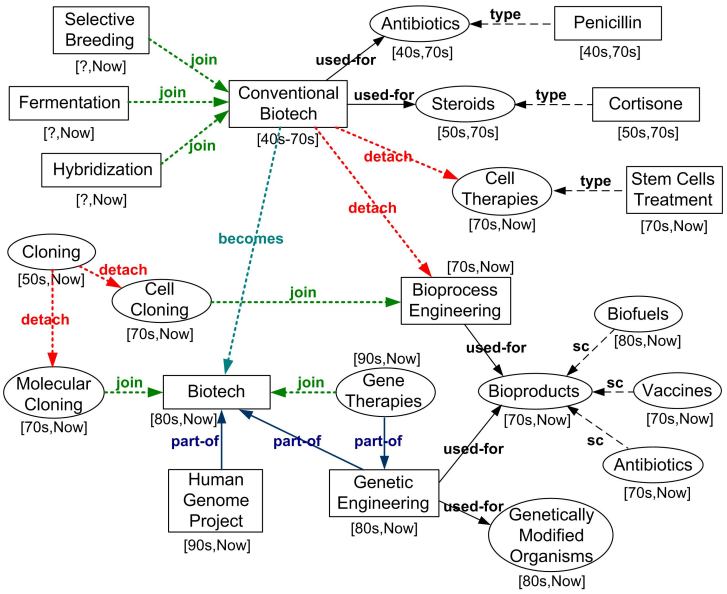


Fig. 8 The evolution of the concept of Biotechnology

ventional Biotech stopped being used and all its related concepts became independent topics. In parallel to this, the new topic of Biotech started to take shape. We could see Biotech as an evolution of the former Conventional Biotech but using Genetic Engineering instead of conventional techniques. Concepts and terms related to the Biotech and Genetic Engineering topics are modeled with a part-of property. In parallel to this, the concept of Cloning techniques started to appear in the 50s, from which the specialized notions of Cell Cloning and Molecular Cloning techniques detached in the 70s and joined the notions of Bioprocess Engineering and Biotech, respectively. The latter is an example of class-level concepts joining instance-level concepts.

[Example Query 3]: Is the academic discipline of biotechnology a wholly new technology branch or has it derived from the combination of other disciplines? Which ones and how? The query requires to follow evolution paths and return the traversed triples in addition to the nodes in order to answer the question of "how." The query is expressed in our language as follows:

$$\begin{aligned}&\textbf{Select } ?Y, ?Z, ?W\\&(?X, \textbf{self} :: \text{Biotechnology}/\textbf{backward}^*, ?Y) \text{ AND}\\&(?Y, \textbf{e-edge}/\textbf{self}, ?Z) \text{ AND } (?Z, \textbf{e-node}, ?W)\end{aligned}$$

The first triple binds $?Y$ to every node reachable from Biotechnology following evolution edges backwards. Then, for each of those nodes, including Biotechnology, the second triple gets all the evolution axes of which the bindings of $?Y$ are subjects whereas the third triple get the objects of the evolution axes. This query returns, (Biotech, $\textbf{becomes}^{-1}$, Conventional Biotech), (Conventional Biotech, $\textbf{join}^{-1}$, Hybridization), (Conventional Biotech, $\textbf{join}^{-1}$, Fermentation), and (Conventional Biotech, $\textbf{join}^{-1}$, Selective Breeding).

[Example Query 4]: Which scientific and engineering concepts and disciplines are related to the emergence of cell cloning? We interpret "related" in our model as being immediate predecessors/successors and siblings in the evolution process. That is, from a concept we first find its immediate predecessors by following all evolution edges backwards one step. We then follow from the result all evolution edges forward on step and we get the original concept and some of its siblings. Finally, we repeat the same process in the opposite direction following evolution edges one step, first forward and then backwards. Based on this notion, we can express the query as follows:

$$\begin{aligned}&\textbf{Select } ?Y, ?Z, ?W\\&(?X, \textbf{self} :: \text{Cell Cloning}, ?Y) \text{ AND}\\&(?Y, \textbf{backward} \mid \textbf{backward}/\textbf{forward}, ?Z)\\&\text{AND } (?Y, \textbf{forward} \mid \textbf{forward}/\textbf{backward}, ?W)\end{aligned}$$

The first triple will just bind Cell Cloning to $?Y$. The second triple follows the **detach** edge back to Cloning, and then the **detach** edge forward to Molecular Cloning. The third triple starts again from Cell Cloning and follows the **join** edge forward to Bioprocess Engineering and then the *detach* edge backwards to Conventional Biotech. All these concepts will be returned by the query.

7 Evolution-Unaware Query Performance Evaluation

To evaluate the efficiency of the evolution-unaware query evaluation 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 improvements in performance with regard to 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 concepts 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 (Johnsonbaugh and Kalin 1991) 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[2]. The trademarks are a kind of intellectual property which may belong to an individual or a company. If some change in ownership occurs the corresponding trademarks have to be re-registered accordingly. For instance, in the domain of corporate mergers, acquisitions and spin-offs, one can observe how the intellectual property is allocated among its owners throughout time. The United States Patent and Trademark Office provides a set of trademarks along with the lists of their owners. To illustrate this, let us take a standard character trademark "Interwise" which was initially registered by `Interwise Inc.` and then, after the `AT&T Corp.` acquisition, it became a property of `AT&T Corp.`. A modeller has two options to store this information: either she can consider both companies as one concept with two name attributes or create two concepts which own the same trademark. Note, that because of temporal consistency constraints we have to split the trademark into two separate concepts in the second modelling choice. With respect to the evolution world semantics we interpret these two modelling possibilities as two possible worlds, because in the second case `Interwise Inc.` and `AT&T Corp.` should be involved into an evolution relation. The dataset extracted from the UPTSO contained approximately 16K unique companies, 200K attributes. In this dataset we have discovered 573 evolution graphs of various sizes between 5 and 373. The example of discovered evolution graph is presented in Figure 7 which depicts the fraction of evolution of AT&T.

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

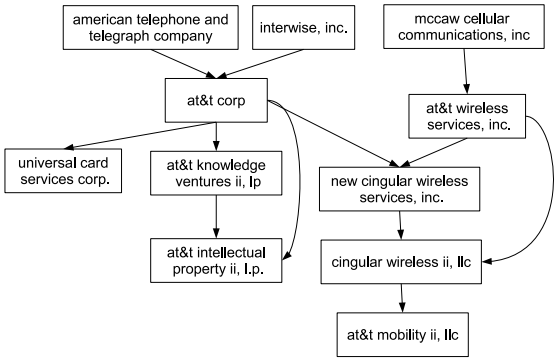


Fig. 9 A fraction of the discovered evolution graph of AT&T

The attributes' frequency in the trademark dataset is distributed according to the Zipfian law, i.e. the most frequent attribute-value pair appear twice as often as the second most frequent attribute value pair, and so on. More specifically, the value of exponent of the found Zipfian distribution is approximately equal to 3. 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 one, 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 Sections 4.2.1 and 4.2.2 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.

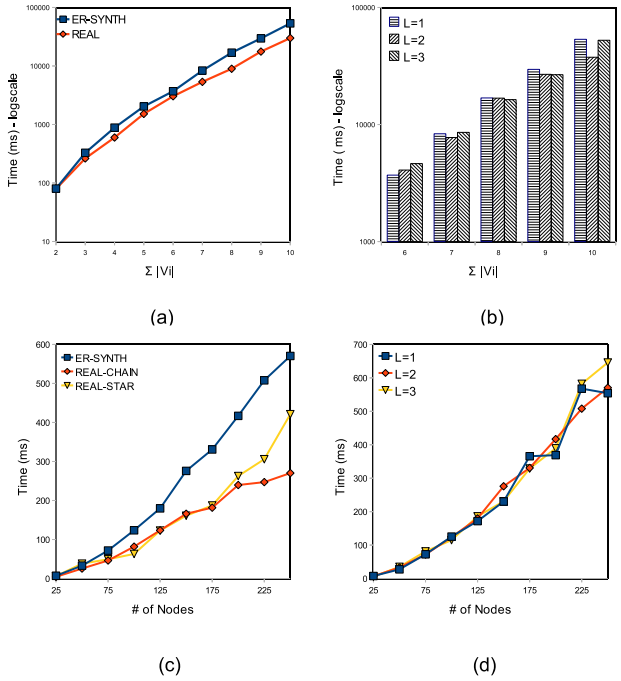


Fig. 10 Steiner forest discovery performance

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 Erds-Rnyi 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 10(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 10(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 10(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 10(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 ran 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

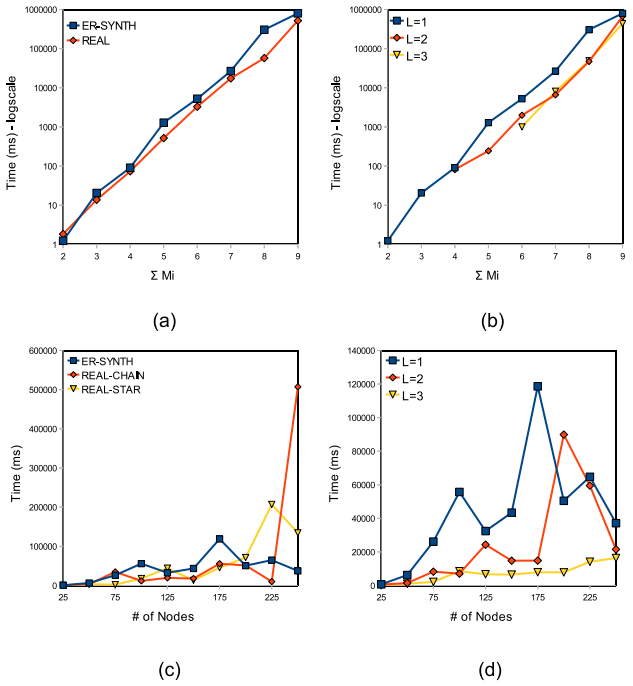


Fig. 11 Query evaluation performance

ER-SYNTH dataset through four parameters, and in particular, the *pool of concepts*, 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 11(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 11(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 11(c) illustrates.

We further tested how the number of concept variables in the query affect the performance. Note that we are mainly interested in the concept-bound variables. Let M represent the number of distinct concept-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 11(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 11(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.

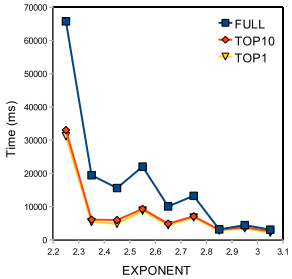
Furthermore, we studied how the query evaluation time scales for different values of M, i.e. for different distinct variables but with the same total number of variable appearances in the query (i.e., the $\sum_{i=1}^{M} M_i$). We used the *ER-SYNTH* dataset again with sizes from 25 to 250, using a step 25. The number of evolution relationships was 110% of the number of concepts. 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 (concepts). 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 11(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 12(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 spread over a number of evolution 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 4.3.1 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 concepts, and 1000 associations distributed over 15 concepts. 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 12(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 4.3.1), thus, when the exponent is small the input sets share many concepts.

# 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)

Fig. 12 Query evaluation time for graphs with different numbers of connected components and for varying exponent of data distribution

8 Related Work

Managing Time in Databases: Temporal data management has been extensively studied in the relational paradigm (Soo 1991). For semi-structured data, one of the first models for managing historical information is an extension of the Object Exchange Model (OEM) (Chawathe et al. 1999). There is also a versioning scheme for XML (Chien et al. 2001). Versioning approaches store the information of the entire document at some point in time and then use edit scripts and change logs to reconstruct versions of the entire document. In contrast, Buneman et al. (2002) and Rizzolo and Vaisman (2008) maintain a single temporal document from which versions of any document fragment (even single elements) can be extracted directly when needed. A survey on temporal extensions to the Entity-Relationship (ER) model is presented by Gregersen and Jensen (1999).

Almost in its entirety, existing work on data changes is based on a data-oriented point of view. It aims at recording and managing changes that are taking place at the values of the data. What has been completely overlooked are other types of changes, such as a concept evolving/mutating into another, or a concept "splitting off" into several others.

Change Management in Ontologies: There is a fundamental distinction between an actual update and a revision in knowledge bases (Katsuno and Mendelzon 1991). An up-

date brings the knowledge base up to date when the world it models has changed. Our evolution framework models updates since it describes how real-world concepts have changed over time. In contrast, a revision incorporates new knowledge from a world that has not changed. An approach to model revision in RDF ontologies has been presented (Konstantinidis et al. 2007). The work on ontology evolution in data integration (Kondylakis and Plexousakis 2010) automatically finds ways to update mappings when the underlying ontologies were changed. In contrast to our kind of evolution,(Kondylakis and Plexousakis 2010) considers structural changes of ontologies with the main focus on mapping recomputation. The survey (Flouris et al. 2008) provides a thorough classification of the types of changes that occur in ontologies. However, there is no entry in their taxonomy that corresponds to the kind of concept evolution we developed in this work; in fact, they view evolution as a special case of versioning. Similarly to versioning in databases, ontology versioning study the problem of maintaining changes in ontologies by creating and managing different variants of it (Klein and Fensel 2001). In our case, we focus on the evolution of a concept that spans different concepts (e.g., student to professor, research lab to independent corporate concept). Highly related, yet different, to concept evolution is the problem of terminology evolution that studies how terms describing the same notion in a domain of discourse are changing over time (Tahmasebi et al. 2008). The high-level changes in RDF/S knowledge bases (Papavassiliou et al. 2009) represent similar to our evolution operators ideas. However, our work focuses on this kind of changes and elaborates on their properties and query evaluation strategies. Closer to our work is the proposal (Kauppinen and Hyvönen 2007) for modeling changes in geographical information systems (GIS). They use the notion of a change bridge to model how the area of geographical concept (countries, provinces) evolve. A change bridge is associated with a change point and indicates what concepts become obsolete, what new concepts are created, and how the new concepts overlap with older ones. Since they focus on the GIS domain, they are not able to model causality and types of evolution involving abstract concepts beyond geographical concepts.

9 Conclusion

This work studies the problem of concept evolution. In contrast to temporal models and schema evolution, concept evolution deals with mereological and causal relationships between concepts. Recording concept evolution also enables users to pose queries on the history of a concept.

We present a framework for modeling evolution as an extension of temporal RDF with mereology and causal properties. These properties are expressed with a set of evolution terms and its query language is an extension of nSPARQL that allows navigation over the history of the concepts. Furthermore, we have designed and implemented an evaluation technique that allows query answering over databases where 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 studied two real use cases where we show the applicability of the proposed framework. In addition, we have performed a number of an extensive experimental evaluation to determine the efficiency of the evaluation technique in the case for evolution-unaware queries.

As future work, we plan to investigate possible algorithmic improvements for the Steiner forest algorithm by parallelization and approximization. Moreover, we look into other non-temporal domains where we could leverage the evolution (casual) relationships.

References

Bhalotia, Gaurav, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan (2002), "Keyword searching and browsing in databases using banks." In *ICDE*, 431–440.

Blakeley, J., P. A. Larson, and F. W. Tompa (1986), "Efficiently Updating Materialized Views." In *SIGMOD*, 61–71.

Buneman, P., S. Khanna, K Tajima, and W. Tan (2002), "Archiving scientific data." In *SIGMOD*, 1–12.

Bykau, Siarhei, John Mylopoulos, Flavio Rizzolo, and Yannis Velegrakis (2011), "Supporting queries spanning across phases of evolving artifacts using steiner forests." In *CIKM*, 1649–1658.

Chawathe, S., S. Abiteboul, and J. Widom (1999), "Managing historical semistructured data." In *Theory and Practice of Object Systems*, 143–162.

Chien, S., V. Tsotras, and C. Zaniolo (2001), "Efficient management of multiversion documents by object referencing." In *VLDB*, 291–300.

Dalvi, N. N., R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu (2009), "A web of concepts." In *PODS*, 1–12.

Dalvi, Nilesh and Dan Suciu (2007), "Efficient query evaluation on probabilistic databases." *The VLDB Journal*, 16, 523–544.

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

Dong, X., A. Y. Halevy, and J. Madhavan (2005), "Reference Reconciliation in Complex Information Spaces." In *SIGMOD*, 85–96.

Dreyfus, S E and R A Wagner (1972), "The Steiner problem in graphs." *Networks*, 1, 195–207.

Dyreson, Curtis E., William S. Evans, Hong Lin, and Richard T. Snodgrass (2000), "Efficiently supported temporal granularities." *IEEE Trans. Knowl. Data Eng.*, 12, 568–587.

Flouris, Giorgos, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou (2008), "Ontology change: classification and survey." *Knowledge Eng. Review*, 23, 117–152.

Gassner, Elisabeth (2010), "The Steiner Forest Problem revisited." *Journal of Discrete Algorithms*, 8, 154–163.

Gregersen, Heidi and Christian S. Jensen (1999), "Temporal Entity-Relationship models - a survey." *IEEE Trans. Knowl. Data Eng.*, 11, 464–497.

Gutiérrez, Claudio, Carlos A. Hurtado, and Alejandro A. Vaisman (2005), "Temporal RDF." In *ESWC*, 93–107.

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

Hull, R. and M. Yoshikawa (1990), "ILOG: Declarative Creation and Manipulation of Object Identifiers." In *VLDB*, 455–468.

Johnsonbaugh, Richard and Martin Kalin (1991), "A graph generation software package." In *SIGCSE*, 151–154.

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

Katsuno, Hirofumi and Alberto O. Mendelzon (1991), "On the difference between updating a knowledge base and revising it." In *KR*, 387–394.

Kauppinen, Tomi and Eero Hyvönen (2007), "Modeling and reasoning about changes in ontology time series." In *Ontologies: A Handbook of Principles, Concepts and Applications in Information Systems*, 319–338.

Keet, C. Maria and Alessandro Artale (2008), "Representing and reasoning over a taxonomy of part-whole relations." *Applied Ontology*, 3, 91–110.

Kimelfeld, B. and Y. Sagiv (2006), "New algorithms for computing steiner trees for a fixed number of terminals." Http://www.cs.huji.ac.il/ bennyk/papers/steiner06.pdf.

Klein, Michel C. A. and Dieter Fensel (2001), "Ontology versioning on the semantic web." In *SWWS*, 75–91.

Kondylakis, Haridimos and Dimitris Plexousakis (2010), "Enabling ontology evolution in data integration." In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, 38:1–38:7, ACM, New York, NY, USA.

Konstantinidis, George, Giorgos Flouris, Grigoris Antoniou, and Vassilis Christophides (2007), "On RDF/S ontology evolution." In *SWDB-ODBIS*, 21–42.

Lenzerini, M. (2002), "Data Integration: A Theoretical Perspective." In *PODS*, 233–246.

Lerner, B. S. (2000), "A Model for Compound Type Changes Encountered in Schema Evolution." *TODS*, 25, 83–127.

Palpanas, T., J. Chaudhry, P. Andritsos, and Y. Velegrakis (2008), "Entity Data Management in OKKAM." In *SWAP*, 729–733.

Papavassiliou, Vicky, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides (2009), "On detecting high-level changes in rdf/s kbs." In *ISWC*, 473–488.

Pérez, Jorge, Marcelo Arenas, and Claudio Gutierrez (2008), "nSPARQL: A navigational language for RDF." In *ISWC*, 66–81.

Rizzolo, F. and A. A. Vaisman (2008), "Temporal XML: modeling, indexing, and query processing." *VLDBJ*, 17, 1179–1212.

Rizzolo, F., Y. Velegrakis, J. Mylopoulos, and S. Bykau (2009), "Modeling Concept Evolution: A Historical Perspective." In *ER*, 331–345.

Soo, M. D. (1991), "Bibliography on Temporal Databases." *SIGMODREC*, 20, 14–23.

Tahmasebi, Nina, Tereza Iofciu, Thomas Risse, Claudia Niederee, and Wolf Siberski (2008), "Terminology evolution in web archiving: Open issues." In *International Web Archiving Workshop*.

Velegrakis, Yannis, Renée J. Miller, and Lucian Popa (2004), "Preserving mapping consistency under schema changes." *VLDB J.*, 13, 274–293.

W3C (2004), "RDF vocabulary description language 1.0: RDF Schema." Http://www.w3.org/TR/rdf-schema/.