Capturing and Querying Multiple Aspects of Semistructured Data

Curtis E. Dyreson, Michael H. Böhlen, and Christian S. Jensen

November 1, 1998

TR-36

A TimeCenter Technical Report

Title	Capturing and Querying Multiple Aspects of Semistructured Data
	Copyright © 1998 Curtis E. Dyreson, Michael H. Böhlen, and Christian S. Jensen. All rights reserved.
Author(s)	Curtis E. Dyreson, Michael H. Böhlen, and Christian S. Jensen
Publication History	November 1998. A TimeCenter Technical Report.

Abstract

Motivated to a large extent by the substantial and growing prominence of the World-Wide Web and the potential benefits that may be obtained by applying database concepts and techniques to web data management, new data models and query languages have emerged that contend with the semistructured nature of web data. These models organize data in graphs. The nodes in a graph denote objects or values, and each edge is labeled with a single word or phrase. Nodes are described by the labels of the paths that lead to them, and these descriptions serve as the basis for querying.

This paper proposes an extensible framework for capturing more data semantics in semistructured data models. Inspired by the multidimensional paradigm that finds application in on-line analytical processing and data warehousing, the framework makes it possible to associate values drawn from an extensible set of dimensions with edges. The paper considers dimensions that capture temporal aspects of data, prices associated with data access, quality ratings associated with the data, and access restrictions on the data. In this way, it accommodates notions from temporal databases, electronic commerce, information quality, and database security,

The paper defines the extensible data model and an accompanying query language that provides new facilities for matching, slicing, and collapsing the enriched paths and for coalescing edges. The paper describes an implemented, SQL-like query language for the extended data model that includes additional constructs for the effective querying of graphs with enriched paths.

1 Introduction

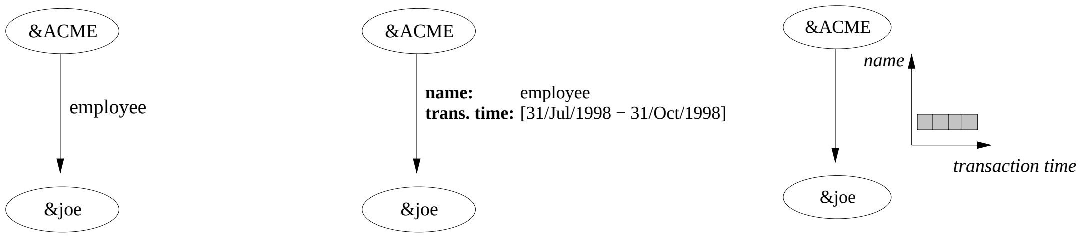
The World-Wide Web ("web") is arguably the world's most frequently used information resource. While current web data has little and mostly local structure, web data will likely have far more in the near future. Specifically, the eXtended Markup Language (XML) is expected to replace the Hypertext Markup Language as the language for web pages [CKR97, BL98], and XML includes a data definition language. So an XML page can have a schema of exactly how the data in the page is structured. This will not result in highly-structured data because the page-level schema can (and likely will) vary widely from page to page. XML will at best only provide semistructure. The ability of semistructured data models to accommodate data that lacks a well-defined schema makes them attractive candidates for querying and managing XML data [FLM98, Suc98].

Semistructured data models organize data in a graph [Bun97, FLM98]. Each node in the graph corresponds to an object or value, while each edge represents a relationship between a pair of objects or an object and a value. Edges are both directed and labeled. The labels are important because they make nodes *self-describing* in the sense that a node is described by the sequences of labels on paths through the graph that lead to the node [Bun97].

Research in semistructured and unstructured data models has concentrated on basic issues such as query language design [BDS95, BDHS96, QRU$^+$97, AQM$^+$97, LHL$^+$98], restructuring of query results [FFLS97, AM98], tools to help naive users query unknown semistructures [GW97, GW98], techniques for improving implementation efficiency [QRU$^+$97, FS98, MDS99], and methods for extracting semistructured data from the web [HGMC$^+$97, NAM97]. But existing research has yet to address a variety of more advanced data modeling issues that have already been addressed in the contexts of more traditional, e.g., relational and object-oriented, data models. In particular, only one paper has addressed support for temporal data [CAW98], and none (to the best of our knowledge) have addressed data security or quality.

This paper presents an extensible, semistructured data model that addresses some of the more advanced data modeling issues. The new model generalizes existing semistructured models by applying a dimensional paradigm to edge labels. A label is a list of *descriptive* properties such as the name of the edge, the time when the edge is valid in the real world, the time during which the edge exists as current in the database, the level of security that protects the edge, the quality of the information source(s) that the edge identifies, and the price for using the edge in a query. These properties, or any other descriptive property, can be used in a label to describe an edge.

The kind of change to edge labels that is presented in this paper is illustrated in Figure 1. Part (a) of the figure shows an edge in an existing semistructured data model. The edge, labeled 'employee', connects two objects, '&ACME' and '&joe'. The edge has a single property chosen from an implicit *name* dimension. Figure 1(b) shows the kind of edge label introduced in this paper. The label is a list of properties. Figure 1(c) graphically depicts the label as a region in a two-dimensional *property space* consisting of *name* and *transaction time* dimensions. In general, property space will be multidimensional, not just two-dimensional.



(a) a typical edge (b) properties in a label (c) a region in property space

Figure 1: The new kind of edge labels

This paper is organized as follows. Section 2 motivates the utility of the extended data model and also discusses why the strategy of adding multidimensional properties to edge labels is useful. To the best of our knowledge, this is the first work that adds properties to edge labels, or treats an edge label as something other than a single word or string. Section 3 presents a semistructured model with multidimensional properties. An important feature of this model is that the dimensions of property spaces are permitted to vary from edge to edge. Consequently, properties found in one label can be *missing* from another label. Missing properties will be common in web data since sites do not share property space dimensions; and even within a site, page authors can use their own property spaces. Existing query languages are unable to handle multidimensional properties in edge labels, so Section 3.2 presents several query operators to query multidimensional properties: *path match*, *path slice*, *path collapse*, and *edge coalesce*. Section 4 incorporates the new query operations into an SQL-like query language, called AUCQL, for querying multidimensional properties in a semistructured data model. AUCQL is a derivative of Lorel [QRU$^+$97, MAG$^+$97, AQM$^+$97]. The paper presents a formal syntax and semantics for AUCQL. The remaining sections cover related work, future work, and summarize the paper.

2 Motivating example

A world-wide movie database maintains a semistructure that spans relevant XML data from the following sites.

- The *Internet Movie Database* is a site that contains a wealth of movie data.
- *Videotastic* is a monthly, on-line movie industry magazine. Portions of the site are only available to paid subscribers.
- *Haus du Flicks* has a collection of film clips, but charges a fee in e-cash for each clip. The fee is collected by an e-cash broker when the clip is accessed.
- *Joe Doe* is a Yankee On-line User who maintains a collection of pages devoted to science fiction movies.
- *Horsing Around Movies* has data about R- and NC-18 rated films. Portions of the site are restricted to web surfers over the age of 18.
- Some of the pages at some of the sites are archived by the *Internet Archives*, a web service. The archiver is a robot that periodically traverses part of the web. The archiver does not visit every page, nor is every version of a page visited.

Figure 2 shows a portion of the movie database. Edges are directed arrows, values are given in italics font, and objects are depicted as ovals. The edge labels in the figure are composed of a list of '**property**

name: property value' pairs, unlike in a typical semistructured database where the label is just a single word or phrase. Each pair is referred to as a *property* of the edge label. Most of the semistructure is not shown—many other edges and nodes exist in the complete movie database.

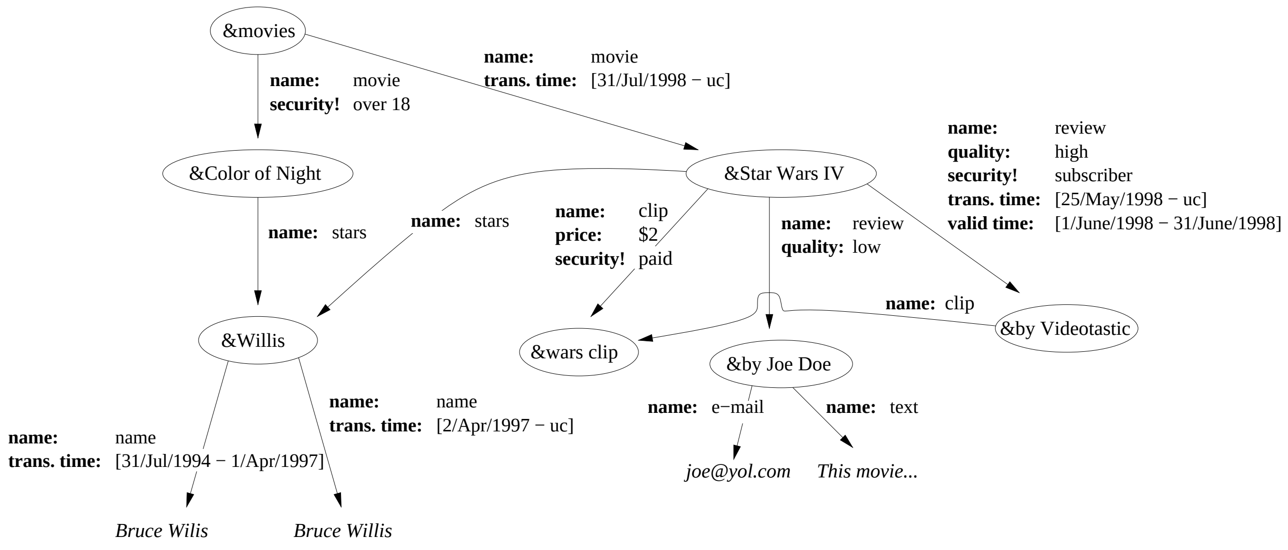


Figure 2: A WWW movie database

The portion that is depicted models the following pertinent facts.

- Information about a new movie, *Star Wars IV*, was added to the Internet Movie Database, on July/31/1998.
- A review of *Star Wars IV* appears in the June issue of *Videotastic*. This issue of *Videotastic* was made available on 25/May/1988. The review is only available to paid subscribers.
- *Haus du Flicks* charges $2 dollars for a *Star Wars IV* film clip, but under a deal with *Videotastic*, paid subscribers can get the clip for free in one of the magazine's reviews.
- Bruce Willis stars in *Star Wars IV*. His misspelled name was corrected on 2/Apr/1997.
- *Horsing Around Movies* has data about the NC-18 rated movie, *Color of Night*. This movie also stars Bruce Willis. Only surfers with an appropriate security clearance (e.g., the content is "protected" by PICS[1] tags) are permitted to view *Color of Night* data from *Horsing Around Movies*.
- *Joe Doe* also has a review of *Star Wars IV*, but since he is a Yankee On-line User, it is deemed to be of *low* quality.

The framework presented in this paper models the facts described above as properties in edge labels. Edge labels are the most appropriate locations for properties, since objects are described by the paths that lead to them. For instance, while the '`&Willis`' node in Figure 2 has a meaningful internal name, '`&Willis`', this name is of *no* importance, and the node may just as easily be called '`&foo`'. It is only known that the object '`stars`' a '`movie`' because there exists an incoming edge labeled '`stars`', which is turn is reached after traversing a '`movie`' edge. Other descriptions of '`&Willis`', say as a '`father`' or as a '`person`', would only be available as labels along other paths to the node (not shown in the figure).

A partial list of properties is presented below. None of these properties are *required* in an edge label. Furthermore, for most edges, one or more properties may be *missing*, as they are in Figure 2.

[1]Platform for Internet Content Selection, <`http://www.w3.org/PICS`>.

name The name is a text description. The domain for names is the set of finite-length strings over some reasonable alphabet (e.g., Unicode characters). In general, the value of this property could be a set of names, but in this paper, for simplicity, only single names are considered.

security Some edges have security restrictions. This paper uses a Netscape-like security model[2], but in general, multiple security-related properties, with either database or web security models can coexist in edge labels. We assume that security is controlled through *certificates*. A certificate (or combination of certificates) permits access to various services. In this paper, the security is given as a formula built of individual certificates, AND, and OR. For instance a security of 'over 18 AND subscriber' would mean that a user needs *both* certificates to access a service, a security of 'over 18 OR subscriber' would mean that either certificate alone would suffice.

transaction time The transaction time of an edge label is the time when the label is current in the database. It is called transaction time since it is typically the time points between the time of the transaction that led to the label and the time of the transaction that deleted or updated the label [JD98]. Edge labels that are current have a special transaction time end value, *until changed*, which indicates that the edge is current and will remain so until it is changed (deleted or updated). In this paper, transaction times are represented using (closed) interval notation. An interval represents a non-empty, contiguous set of transaction-time points. To properly record transaction time, the database should enforce the special semantics in database modifications, (presented in Section 3.3).

As an aside, separate transaction-time timestamps on nodes are unnecessary as long as the roots of the semistructure are maintained (e.g., by adding explicit `root` edges and timestamping those edges). The transaction time of a node must contain all of the transaction times of incoming edges (since an edge can only connect existing nodes). If the node exists at a transaction time not represented in any of the incoming edges then for that transaction time, the node is a root node.

valid time The valid time of a database fact or relationship indicates when that fact or relationship is true in the modeled reality [JD98]. In our context, the valid-time property of an edge label thus indicates when that label reflects the real world. Often the valid time and transaction time are the same (a degenerate relationship between the two), but this is only one of many possible relationships between these times [JS94]. Valid-time timestamps are represented using closed interval notation. The interval represents a non-empty, contiguous set of valid-time points.

price When data is spread over a network, accessing some data may have substantially greater cost (in terms of size, time, or money). The price property reflects these differing costs in obtaining data. Multiple price properties can comfortably coexist within in our framework. We will assume that the price is a dollar amount.

quality Information on the web arises from many sources, some of which are far more credible than others. For instance, one would commonly rate information from the CNN server as more credible than information from a user's personal home page. The quality property records the quality of the source of the information. We will assume that the quality is an intuitive ranking from *low* to *high*.

The above list only covers properties used in the movie database examples, and does not exclude other properties such as language or URL space. The most common property is **name**—only in unusual circumstances will an edge be unnamed. The framework, however, treats all properties orthogonally. No property has a special status.

Many edge labels in Figure 2 contain several properties. The description represented by the label is the *combination* of all of the properties. Consider the two edges from the '`&Willis`' node shown in Figure 2. Both edges have the same value for the **name** property, but at different transaction times.

[2]Netscape Security Home Page, <`http://home.mcom.com/products/security/index.html`>.

An important requirement of a semistructured (and unstructured) data model is that it be flexible, so that it is capable of accommodating many of the schema irregularities found in web data. In keeping with this requirement, the framework presented in the paper has several features worth mentioning.

The most important feature is that properties can be *missing*. As exemplified by Figure 2 properties are missing from many of the labels. Generally, properties are missing for one of the following reasons.

1. The property is irrelevant, and its presence would not improve a description. A **price** property is irrelevant when the data is free.

2. The property value is unknown. A page author does not know the valid time, and so does not include a valid time property in the label.

The first case can be characterized as *don't care* information, as in, this property is missing because we don't care if it is present, it will not help the description. This is how missing properties are interpreted and used in this paper. In data derived from the web, missing properties will be common since different sites will use different properties to describe data; and even within a site, page authors will commonly only be interested in using a few properties to describe their data and will not care about other properties. The second case in an example of an *unknown* value of a (potentially) relevant property. This is not and should not be handled by a missing property; instead the property should be explicitly listed with an unknown value (e.g., a null value).

A second feature is that some properties can be specified as being *required*. A required property can be thought of as an *essential* part of a description. By specifying a property to be required, the author of the data forces the property to be matched during a query. The **security** property on the edge to '`&Color of Night`' is a required property (specially indicated by affixing a '!' to the property name). It is meant to indicate that a user *must* have a matching security clearance to retrieve any path through that edge from the database. Further details on how required properties are matched are presented in Section 3.2.2.

Finally, multiple edges may connect the same pair of nodes with overlapping or redundant descriptions. Several edges, with slightly different labels, between the same pair of nodes will be common in this framework. Requiring these labels to contain unique descriptions would be an unnecessary restriction on the freedom of data organization and description.

While multiple properties in an edge label can capture more data semantics, they break existing query languages. To take one example, consider the path from '`&movies`' through '`&Star Wars IV`' to the misspelled value *Bruce Wilis*. It would be easy to retrieve that path by using an appropriate regular expression over the **name** property in each label (e.g., `movie.stars.name`). While this is a path, it is not a *valid* path since the transaction times of the first and last edges in the path are disjoint. In other words, when the first edge in the path was inserted, the final edge was already deleted. So for no time did that path actually exist as a current path in the database.

A contribution of this paper is a collection of "extensible" query language operators to more correctly manipulate the extended edge labels. Each operator is extensible in the sense that the semantics of properties are not fixed in the data model, rather the meaning is supplied by a database designer. For instance, to test the validity of a path, the **transaction time** property will be tested quite differently than the **name** property. Several new query operators are also described. *PathSlice* cuts a section from each property on a path. *PathCollapse* computes a label for an entire path from the labels on each edge in the path. And *PropertyCoalesce* computes the value of a property which is physically distributed in a number of different labels on edges between the same pair of nodes.

3 Extending a semistructured data model with properties

In this section the notion of an edge label is generalized to include a number of properties. A formal semantics for properties is presented, and various properties are informally discussed.

3.1 A semistructured model with properties

A semistructured database, $DB = (V, E, \&root, \Gamma)$, consists of a set of nodes, V, a set of labeled, directed edges, E, a single root node, $\&root$, and a collection of semantics for properties, Γ. Let $ROOTS \subseteq E$ be the set of edges from $\&root$.[3] An edge in E from node v to node w with the label $\mathcal{R}$ is denoted $v \xrightarrow{\mathcal{R}} w$. Label $\mathcal{R}$ is a *property space region*, or *region* for short.

Definition 3.1 [Property space region]
A property space region, $\mathcal{R}$, is a set of m pairs, $\{(p_1\colon x_1),\ (p_2\colon x_2),\ \ldots\ ,\ (p_m\colon x_m)\}$, where

- each p_i is the *name* of a *property space dimension*,
- x_i is a set of *values* drawn from the domain of that property space dimension, that is $x_i \subseteq domain(p_i)$,
- the semantics of each p_i is in Γ, and
- each name of a property space dimension is unique, that is, $\forall i, j (p_i = p_j \Rightarrow i = j)$.

A *required* property, say p_i with value x_i, is denoted $(p_i!\ x_i)$. ▌

A property space region $\mathcal{R} = x_1 \times \ldots \times x_m$ is thus a subset of the property space, $domain(p_1) \times \ldots \times domain(p_n)$, where each p_i is a *property space dimension*. To manipulate property values in queries, several operations for each property space dimension are needed (see Section 3.2); we assume that these operations are available in Γ.

Example 3.2 In Figure 2 a single edge connects '`&movies`' to '`&Color of Night`'. The edge label is the property space region {(**name:** movie), (**security!** over 18)}. The **security** property is a requied property. It is intended to limit access to the node to individuals over 18 years of age. The **name** property is not required. ▌

This framework does not assume that the dimensions of property space are the same for every edge in the database. New dimensions may be introduced at any time.

3.2 Queries

The main difference between retrieving data in a relational database and retrieving it from a semistructured database is in how the data is accessed. In most semistructured and unstructured query languages, a *path expression* is used to access data in the graph. Typically, this path expression is a regular expression over a language of edge labels. To determine if a node is accessible, the path expression is matched against the actual sequence of labels on each path from a root to that node. If a sequence (path) matches, then the node can be accessed.

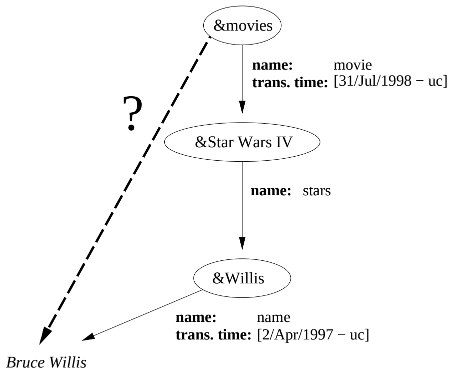
This section discusses operations to access nodes in a semistructured data model with labels that are property space regions. A semantics for each operation is given, and the utility of the definition is motivated with an example. In Section 4 these additional operations are incorporated into an SQL-like query language.

3.2.1 Path validity

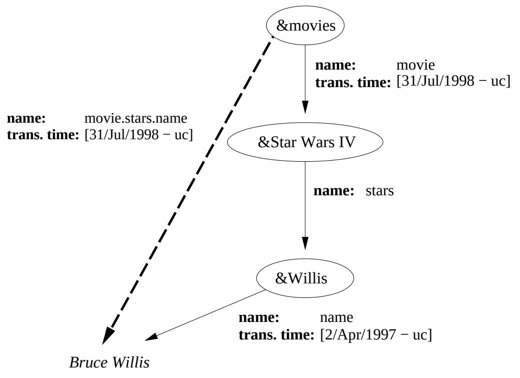
Only some of the paths in the semistructure are *valid*. Intuitively a path is valid if it transits through some "common" property space region. The common region is computed by *collapsing* the path.

Consider the case of a path to a movie star's name. One such path is shown in Figure 3(a). Intuitively the path is a kind of *virtual edge* from a root (in this case the root is '`&movies`') to a name. In the figure, the virtual edge is depicted as a dashed line. The virtual edge should have a label that describes it, just like any other edge. This label is determined by *collapsing* the labels along the path into a single label.

[3]This is the set of edges to what would normally be considered the roots of the semistructure; the extra level of indirection is added to record the additional properties of the root nodes, such as their quality or security.



(a) uncollapsed



(b) collapsed

Figure 3: A (virtual) edge for the name of a movie star

A path is collapsed by recursively collapsing regions along the path. A pair of regions is collapsed by determining their common properties. If a region alone has some property, that property is propagated to the collapsed region. This is because a missing property in a region is don't care information, meaning that any value of the property is acceptable for the region. For properties that appear in both regions, a dimension-specific collapsing constructor is used to compute the value of the property. This constructor could construct an *empty* property, which signifies that these regions do not have any commonality for that property.

Definition 3.3 $[\mathit{CollapsePath}_\Gamma : \mathit{PATH} \rightarrow \mathit{EDGE}]$
$\mathit{CollapsePath}_\Gamma$ takes a path and computes the common region on a virtual edge between the first and last nodes in the path. It is defined recursively below. The definition is extensible in that it depends on the semantics of the property space dimensions as given by Γ. $\mathit{PropertyCollapse}_p$ is a dimension-specific constructor in Γ that is used to collapse a pair of property values in dimension p. In this operation, required properties are treated the same as other properties.

$$\begin{aligned}
&\mathit{CollapsePath}_\Gamma(v \xrightarrow{\mathcal{R}} w) = v \xrightarrow{\mathcal{R}} w \\
&\mathit{CollapsePath}_\Gamma(v \xrightarrow{\mathcal{R}_1} x \xrightarrow{\mathcal{R}_2} w) = \quad v \xrightarrow{\mathcal{R}} w \text{ where} \\
&\qquad \mathcal{R} = \{(p\colon \mathit{PropertyCollapse}_p(x,\ y) \mid (p\colon x) \in \mathcal{R}_1 \ \wedge\ (p\colon y) \in \mathcal{R}_2\} \bigcup \\
&\qquad\qquad \{(p\colon x) \mid (p\colon x) \in \mathcal{R}_1 \ \wedge\ (p\colon y) \notin \mathcal{R}_2\} \bigcup \\
&\qquad\qquad \{(p\colon y) \mid (p\colon x) \notin \mathcal{R}_1 \ \wedge\ (p\colon y) \in \mathcal{R}_2\} \\
&\mathit{CollapsePath}_\Gamma(v \xrightarrow{\mathcal{R}_1} x \xrightarrow{\mathcal{R}_2} \ldots \xrightarrow{\mathcal{R}_m} w) = \\
&\qquad \mathit{CollapsePath}_\Gamma(v \xrightarrow{\mathcal{R}_1} \mathit{CollapsePath}_\Gamma(x \xrightarrow{\mathcal{R}_2} \ldots \xrightarrow{\mathcal{R}_m} w)) \qquad \blacksquare
\end{aligned}$$

The collapsing constructor, $\mathit{PropertyCollapse}_p$, depends on the semantics of the dimension.

- **name** (string concatenation) A name is a string. A pair of names is collapsed by concatenating the names. An additional character (e.g., '.') can be inserted as desired. For example, the path with name properties "(**name:** movies) (**name:** stars) (**name:** name)" would be collapsed into the string "movies.stars.name".
- **security** (AND) To access a path, the security restriction of each edge on the path must be satisfied. The security restriction property of a path is then the conjunction of the security restrictions of each edge in the path.
- **transaction time** (time interval intersection) For a path to be accessible at a particular time, each individual edge on the path must be current at that time, and so a path is accessible if the transaction-time properties of each edge label on the path have a non-empty intersection.
- **valid time** (time interval intersection) Valid time is collapsed like transaction time.

- **price** (sum) The individual prices along a path to obtain an overall price for a path is a typical collapsing constructor for this property. Alternatively, the maximum price could be chosen to model "network flow" situations [Hol89].
- **quality** (minimum) The lowest quality, that is, the least credible information, is the quality of the overall path.

In general, since each dimension is collapsed independently, the collapse constructor for a dimension should either be a *mutator*, which transforms one domain value into another, e.g., concatenation, or a *restrictor*, which reduces the set of domain values, e.g., time interval intersection.

Example 3.4 [The transaction time of a path to *Bruce Willis*]
The **transaction time** property in the collapsed path in Figure 3(b) is [31/Jul/1998 - uc]. This is the intersection of the times on the first and last edges in the path (the transaction time of the middle edge is missing and so does not play a part in the computation). The transaction time in the description indicates that the value *Bruce Willis* was described in the database by `movie.stars.name` from 31/Jul/1998 to the current time (until it is changed). Note that this is not an *exclusive* description—a different `movie.stars.name` path (through '`&Color of Night`') is current over a slightly longer transaction-time interval. ∎

To determine if a path is valid, the path is first "collapsed" and then each property is checked to ensure that it is non-empty.

Definition 3.5 [$PathValid_{\Gamma} : PATH \rightarrow BOOLEAN$]
A path, P, is valid if after collapsing the path, there are no empty properties.

$$PathValid_{\Gamma}(P) = \forall p[(p\colon \emptyset) \notin \mathcal{R}] \text{ where } v \xrightarrow{\mathcal{R}} w = PathCollapse_{\Gamma}(P)$$ ∎

Example 3.6 [The transaction time of a path to *Bruce Wilis*]
Consider the path from '`&movies`' through '`&Star Wars IV`' to the misspelled value *Bruce Wilis* in Figure 2. When the path is collapsed, the **name** property in the resulting region has the value `movie.stars.name`. But the **transaction time** property is $\emptyset$. The transaction times of the first and last edges in the path are disjoint, and so their intersection results in $\emptyset$ (the transaction time of the middle edge is missing and so does not play a part in the computation). Consequently the path is not a valid path. ∎

3.2.2 Path *Match*

In this section, we give a semantics for determining whether a user-given *descriptor* matches a property space region. The next section shows how to use these descriptors in *regular expressions* to match paths.

Label matching in existing semistructured query languages is straightforward. The descriptor is typically a single word or phrase that is compared, using string comparison, to the edge label. For example, in the regular expression `(person | employee).name?`, the descriptors, the basic building blocks of the regular expression, are: `person`, `employee`, and `name`. During evaluation of this expression, the descriptor '`person`' would only match a label '`person`' on an edge. More flexible string comparisons between descriptors and labels are supported in some languages, such as Lorel [AQM+97], which reuses the wildcard operator '`%`' from SQL. So the descriptor '`per%`' would match any label that starts with 'per'.

The semantics of the label matching is more involved in our model because of required and missing properties. In addition, string comparison is an insufficient matching operation since there are many properties which are not strings.

These complications are addressed in the *RegionMatch* operation defined below. In general, *RegionMatch* succeeds if every individual property in the descriptor has a match in the region or is missing from the region. Extra properties are ignored, and different $PropertyMatch_p$ operations may be used in different dimensions. There are several cases to consider.

- A *required* property in one region is *missing* from the other region. In this case, the match does not succeed. A required property must be present in both regions.
- A non-required property in one region is *missing* from the other region. In this case, the match succeeds. The missing property is treated as don't care information.
- The property is present in both regions. A predicate, $\mathit{PropertyMatch}_p$, which is specific to the property space dimension, is used to determine if the property values match. Required and non-required properties are treated the same.

Definition 3.7 [$\mathit{RegionMatch}_\Gamma : \mathit{REGION} \times \mathit{REGION} \rightarrow \mathit{BOOLEAN}$]
Region $\mathcal{R}$ is matched against region $\mathcal{S}$ as follows. *RegionMatch* depends on the semantics of the property space dimensions specified in Γ, since properties in the regions are individually matched.

$$\begin{aligned}\mathit{RegionMatch}_\Gamma(\mathcal{R},\ \mathcal{S}) = \ & \forall (p!\ x) \in \mathcal{R},\ \exists y\ [((p\!:\ y) \in \mathcal{S}\ \vee\ (p!\ y) \in \mathcal{S}) \wedge\ \mathit{PropertyMatch}_p(x,y)] \vee \\ & \forall (p\!:\ x) \in \mathcal{R}\ [(\forall (q\!:\ y) \in \mathcal{S}\ [p \neq q]\ \wedge\ \forall (q!\ y) \in \mathcal{S}\ [p \neq q])\ \vee \\ & \qquad \exists y\ [(p\!:\ y) \in \mathcal{S}\ \vee\ (p!\ y) \in \mathcal{S}) \wedge\ \mathit{PropertyMatch}_p(x,y)]] \vee \\ & \forall (p!\ x) \in \mathcal{S},\ \exists y\ [((p\!:\ y) \in \mathcal{R}\ \vee\ (p!\ y) \in \mathcal{R}) \wedge\ \mathit{PropertyMatch}_p(y,x)] \vee \\ & \forall (p\!:\ x) \in \mathcal{S}\ [(\forall (q\!:\ y) \in \mathcal{R}\ [p \neq q]\ \wedge\ \forall (q!\ y) \in \mathcal{R}\ [p \neq q])\ \vee \\ & \qquad \exists y\ [(p\!:\ y) \in \mathcal{R}\ \vee\ (p!\ y) \in \mathcal{S}) \wedge\ \mathit{PropertyMatch}_p(y,x)]] \quad \blacksquare\end{aligned}$$

In the above definition, $\mathit{PropertyMatch}_p$ matches two property values from dimension p. The predicate to use depends on the semantics of the dimension.

- **name** (=) The equals predicate is a good choice for single names. The names must match exactly.
- **security** (truth assignment) The security is usually a required property. The security in the first region is the set of certificates owned by a user, and the security in the second region is the set of certificates required (expressed as a boolean formula). All certificates present in the first region are assigned value *True* (meaning, yes, the certificate is owned by a user), and all others value *False* value (meaning, no, the certificate is not owned by a user). Then the formula giving the second region is evaluated with these assignments. For instance the property "(**security:** subscriber)" would not match "(**security!** subscriber AND over 18)", but would match "(**security!** subscriber OR over 18)".
- **transaction time** (time interval overlaps) Overlaps determines if two intervals overlap.
- **valid time** (time interval overlaps) Valid time is matched like transaction time.
- **price** ($\geq$) The price in the first region should be equal to or larger than the price on the second region.
- **quality** ($\leq$) The quality of the first region must be less than or equal to that of the second region.

Example 3.8 [Looking for a movie]
Below is a region that requires a movie description.

$\mathcal{R}_{movie}$ = {(**name!** movie)}

In Figure 2, there are two labels with a '`movie`' name property. One describes '`&Color of Night`'; the other, '`&Star Wars IV`'.

$\mathcal{S}_{color}$ = {(**name:** movie), (**security!** over 18)}
$\mathcal{S}_{star}$ = {(**name:** movie), (**trans. time:** [31/Jul/1998 - uc])}

These regions are matched as follows.

- $\mathit{RegionMatch}_\Gamma(\mathcal{R}_{movie},\ \mathcal{S}_{color})$ = *False*, since the required **security**, over 18, is missing from $\mathcal{R}_{movie}$.

- $\mathit{RegionMatch}_{\Gamma}(\mathcal{R}_{movie},\ \mathcal{S}_{star}) = \mathit{True}$, since the extra **transaction time** property in $\mathcal{S}_{star}$ is ignored. ▮

RegionMatch is the basis for interpreting *regular expressions* of descriptors. Generally, these regular expressions are interpreted exactly as in other semistructured and unstructured query languages, and usual features of regular expressions have their standard meaning, e.g., '`*`' means zero or more. However, there is one *major* difference between our language and standard semistructured query languages. The matched path must be checked to ensure that it is *valid.*

The following operation extends a set of paths in a semistructure, if the sequence of labels on the extended path matches the regular expression and the path is valid.

Definition 3.9 [$\mathit{Match}_{DB} : \{\mathit{PATH}\} \times \mathit{REGEXP} \to \{\mathit{PATH}\}$]
A descriptor regular expression, X, *matches* a path, P, if the sequence of labels on the path is a legal sentence in the regular grammar specified by X, $W(X)$. This path may extend a path in a starting set of paths, S, as follows.

$$\begin{aligned}\mathit{Match}_{DB}(S,\ X) = \{v_1 \xrightarrow{\mathcal{R}_1} \dots \xrightarrow{\mathcal{R}_m} v_m \mid {} & v_1 \xrightarrow{\mathcal{R}_1} \dots \xrightarrow{\mathcal{R}_{i-1}} v_i \in S \ \wedge \\ & v_i \xrightarrow{\mathcal{R}_i} v_{i+1}, v_{i+1} \xrightarrow{\mathcal{R}_{i+1}} v_{i+2}, \dots, v_{m-1} \xrightarrow{\mathcal{R}_m} v_{m-1} \in E \ \wedge \\ & \mathcal{R}_{i+1}, \dots, \mathcal{R}_m \in W(X) \ \wedge \\ & \mathit{PathValid}_{\Gamma}(v \xrightarrow{\mathcal{R}_1} \dots \xrightarrow{\mathcal{R}_m} w)\}\end{aligned}$$ ▮

Sometimes only the set of matched nodes is desired.

Definition 3.10 [$\mathit{Nodes} : \{\mathit{PATH}\} \to \{\mathit{NODES}\}$]
Let P be a set of paths.

$$\mathit{Nodes}(P) = \{w \mid v \xrightarrow{\mathcal{R}_1} \dots \xrightarrow{\mathcal{R}_m} w \in P\}$$ ▮

Example 3.11 [Movie stars' names as of 31/Jul/1998]
A user is interested in retrieving information about movie stars as of 31/Jul/1998. That set of nodes can be obtained as follows.

$\mathcal{R}_{movie}$ = {(**name!** movie), (**trans. time:** [31/Jul/1998 - 31/Jul/1998])}
$\mathcal{R}_{stars}$ = {(**name!** stars), (**trans. time:** [31/Jul/1998 - 31/Jul/1998])}
$\mathcal{R}_{name}$ = {(**name!** name), (**trans. time:** [31/Jul/1998 - 31/Jul/1998])}
$\mathit{Nodes}(\mathit{Match}_{DB}(\mathit{ROOTS},\ \mathcal{R}_{movie}.\mathcal{R}_{stars}.\mathcal{R}_{name}))$

Recall that *ROOTS* is the set of edges from &*root* to roots in the semistructure. The regular expression in this example is a sequence of descriptors. In each descriptor, the **name** is required (so an edge that is missing a **name** will not match), but the transaction time is not required (an edge that is missing a transaction time is presumed to exist at all transaction times). Properties not mentioned in the descriptor are ignored in the path matching, unless the property is required in which case the label is not matched. Four paths in Figure 2 match the **name** property criteria.

1. The path through '`&Color of Night`' to the misspelled value '*Bruce Wilis*' is not matched since the required level of **security** (over 18) is missing from the descriptors. The user must have the appropriate digital certificate, one that authenticates them as being over 18, and must add that to the first descriptor to match that edge.

2. The path through '`&Color of Night`' to the value '*Bruce Willis*' is also not matched for the same reason.

3. The path through '`&Star Wars IV`' to the misspelled value '*Bruce Wilis*' matches the regular expression, but is not a valid path (see Example 3.6).

4. The path through '`&Star Wars IV`' to the value '*Bruce Willis*' is the only path that both matches the regular expression and is a valid path. ▮

3.2.3 Path *Slice*

It is often useful to slice a portion from each property space region along a path. The most common example is a transaction-time slice, or *rollback* query, that determines the other properties as of a particular transaction time. A path is sliced by slicing each region in the path, and checking whether the resulting path is valid.

Definition 3.12 [$\mathit{Slice}_\Gamma : \mathit{REGION} \times \{\mathit{PATHS}\} \rightarrow \{\mathit{PATHS}\}$]
A descriptor, $\mathcal{L}$, *slices* the labels along each path in a set of paths, S, as follows.

$$\begin{aligned}\mathit{Slice}_\Gamma(\mathcal{L},\ S) = \{v \xrightarrow{\mathcal{R}'_1} \ldots \xrightarrow{\mathcal{R}'_m} w \mid v \xrightarrow{\mathcal{R}_1} \ldots \xrightarrow{\mathcal{R}_m} w \in S \ \wedge\ \mathit{PathValid}_\Gamma(v \xrightarrow{\mathcal{R}'_1} \ldots \xrightarrow{\mathcal{R}'_m} w) \wedge \\ \forall i\ [\mathcal{R}'_i = \mathit{RegionSlice}_\Gamma(\mathcal{L}, \mathcal{R}_i)]\} \quad \blacksquare\end{aligned}$$

A region is sliced property by property. Slicing a region is complicated by missing properties.

- A property that is missing from the descriptor, but present in the region, is passed unchanged into the result.
- Missing properties in a region are also missing in the result, except if the descriptor *requires* the property, in which case the property from the descriptor is added to the result.
- Finally, if the property is both in the region and the descriptor then a dimension-specific constructor slices the property appropriately and adds it to the result.

Definition 3.13 [$\mathit{RegionSlice}_\Gamma : \mathit{REGION} \times \mathit{REGION} \rightarrow \mathit{REGION}$]
A descriptor, $\mathcal{L} = \{(p_1\colon x_1),\ \ldots\ ,\ (p_m\colon x_m)\}$, *slices* a region, $\mathcal{R}$, as follows.

$$\begin{aligned}\mathit{RegionSlice}_\Gamma(\mathcal{L},\ \mathcal{R}) = \ & \{(p\colon \mathit{PropertySlice}_p(x,\ y)) \mid ((p!\ x) \in \mathcal{R} \vee (p\colon x) \in \mathcal{R}) \ \wedge \\ & \qquad ((p\colon y) \in \mathcal{L} \vee (p!\ y) \in \mathcal{L})\} \bigcup \\ & \{(p\colon y) \mid (p!\ y) \in \mathcal{L}\ \wedge\ (p\colon x) \notin \mathcal{R}\} \bigcup \\ & \{(p\colon x) \mid (p!\ x) \in \mathcal{R}\ \wedge\ (p\colon y) \notin \mathcal{L}\} \quad \blacksquare\end{aligned}$$

$\mathit{PropertySlice}_p$ is a dimension-specific constructor that slices a property, but the semantics of the constructor depends on the semantics of the property space dimension.

- **name** (substring) This constructor slices a portion from each name.
- **security** (conjunct elimination) This would limit the security certificates in a region to those that overlap one of the conjuncts in in the descriptor, so a slice with 'over 18' on a security of 'over 18 OR paid subscriber' would result in 'over 18'.
- **transaction and valid time** (intersection) Slicing as of a point in time will be the most common kind of slice.
- **price** (greater-than pruning) The slice eliminates costly edges, e.g., slice with respect to regions that have a price of \$1 or less.
- **quality** (less-than pruning) Slicing can be used to eliminate low quality edges.

Example 3.14 [Transaction-time slice for movie stars' names as of now]
A user is interested in retrieving the other properties about movie stars names as of the current time. That set of paths can be obtained as follows.

$\mathcal{R}_{movie} = \{(\textbf{name!}\ \text{movie})\}$
$\mathcal{R}_{stars} = \{(\textbf{name!}\ \text{stars})\}$
$\mathcal{R}_{name} = \{(\textbf{name!}\ \text{name})\}$
$\mathcal{L}_{now} = \{(\textbf{trans. time:}\ [\text{now - now}])\}$
$\mathit{Slice}_\Gamma(\mathcal{L}_{now}, \mathit{Match}_{DB}(\mathit{ROOTS},\ \mathcal{R}_{movie}.\mathcal{R}_{stars}.\mathcal{R}_{name}))$

Note that a Slice_Γ with $\mathcal{L}_{now}$ as its first argument differs from a *Match* with that descriptor since the **transaction time** property of every label (that has a transaction time) in the sliced path is [now - now], whereas the **transaction time** property in the matched path would be unchanged from the underlying data. ▮

3.2.4 Path *Collapse*

In this section, the $\mathit{PathCollapse}_\Gamma$ operation introduced in Section 3.2.1 is trivially generalized to collapse every path in a set of paths. Typically, Match_{DB} first chooses a set of paths that match some regular expression, then the paths are collapsed, and a property is coalesced from the collapsed paths.

Definition 3.15 $[\mathit{Collapse}_\Gamma : \{\mathit{PATHS}\} \rightarrow \{\mathit{PATHS}\}]$
A set of paths, S, is collapsed by collapsing each path independently.

$$\mathit{Collapse}_\Gamma(S) = \{\mathit{CollapsePath}_\Gamma P \mid P \in S \ \wedge \ \mathit{PathValid}_\Gamma(P)\}$$ ▮

3.2.5 Coalescing a property

Several (virtual) edges may connect a pair of nodes. For example, two edges connect the pair of nodes in Figure 4. The first edge was added when the review began to be developed on 15/Mar/1998. The security was set to restrict the edge to page developers. By 25/May/1998, the edge was publicly released as part of the June issue and so the security was weakened to include paid subscribers.

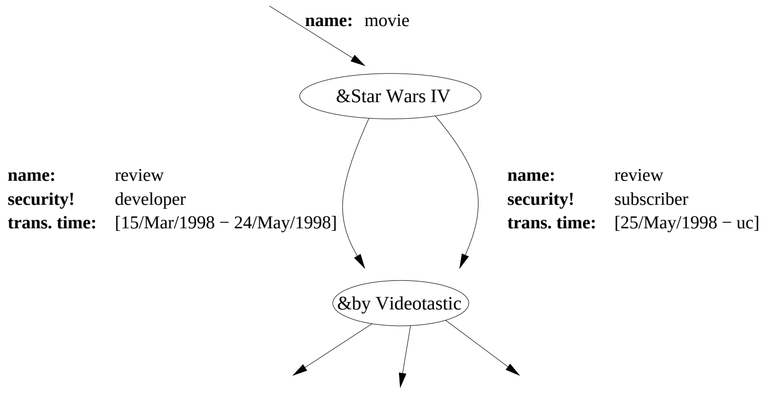


Figure 4: Evolving information about the *Videotastic* review of *Star Wars IV*

When several edges connect a pair of nodes, information about a single property may be distributed among multiple property space regions. In order to determine the full extent of a property that (conceptually) pertains to a relationship between a pair of nodes, regardless of whether information about that property is distributed among a number of edges, it is advantageous to *coalesce* the property from the set of edges.

Definition 3.16 $[\mathit{Coalesce}_\Gamma : \mathit{PROPERTY} \times \{\mathit{EDGES}\} \rightarrow \{\mathit{PROPERTIES}\}]$
A set of edges, F, is coalesced in a *single* property space dimension as follows.

$$\mathit{Coalesce}_\Gamma(p, F) = \{(v,\ (p\text{: } z),\ w) \mid z = \mathit{PropertyCoalesce}_p(\{x \mid ((p\text{: } x) \in \mathcal{R} \vee (p!\ x) \in \mathcal{R}) \ \wedge \ v \xrightarrow{\mathcal{R}} w \in F\})\}$$ ▮

The *PropertyCoalesce* operation is a dimension-specific constructor. But, unlike the collapsing constructor, the coalescing constructor does not have to be a restrictor or mutator. Also, the result is not an edge labeled with a property space region. Rather, it is a list of both nodes and a single, coalesced property.

Typically a property is coalesced from a set of collapsed paths. The coalescing constructor to use depends on the semantics of the dimension.

- **name** (union) A pair of names is coalesced through union; this implies the coalesced edge is accessible using either name.
- **security** (OR) The coalesced security is any of the security over the set of argument edges.
- **transaction and valid time** (temporal coalesce) When more than one edge connects a pair of nodes, access is available at any time that at least one of the edges exists. Temporal coalesce computes the set of maximal intervals equivalent to the set of argument intervals.
- **price** (min) In the computation of the coalesced price, the minimum price dominates since the user will typically want the cheapest price.
- **quality** (average) A weighted average of the quality is computed since the experts disagree on the quality. Alternatively, a max quality could dominate if the user will accept the highest rating as the "right" rating.

Example 3.17 [The coalesced transaction time for the *Star Wars IV* review]
The following strategy can be used to determine the **transaction time** for the review of *Star Wars IV* by *Videotastic*, irrespective of the **security**, **valid time**, etc. First, find all the paths from a root to the review. Note that this requires a certain level of **security**. Second, collapse each path into a virtual edge. Finally, coalesce the **transaction time** from the virtual edges.

$\mathcal{R}_{movie} = \{(\textbf{name!}\ \text{movie}), (\textbf{security:}\ \text{developer})\}$
$\mathcal{R}_{review} = \{(\textbf{name!}\ \text{review}), (\textbf{security:}\ \text{developer})\}$
$\mathit{Coalesce}_{\Gamma}(\textbf{trans. time},\ \mathit{Collapse}_{\Gamma}(\mathit{Match}_{DB}(\mathit{ROOTS},\ \mathcal{R}_{movie}.\mathcal{R}_{review})))$

The result is given below.

{(`&root`, (**trans. time:** [15/Mar/1998 - uc]), `&by Videotastic`)}.

The coalesced transaction time property, [15/Mar/1998 - uc], is the union of the two transaction time intervals in Figure 4. ∎

3.3 Updates

When transaction time is one of the supported dimensions in property space, special semantics for update must be enforced to accommodate transaction time. In a transaction-time database the database is trusted to enforce these semantics. On the web, no such trusted mechanism is available for updates.

In this section, we describe the constraints that should exist to correctly support transaction time, but leave open the issue of how these constraints are enforced on update. An update can be either at the data level, consisting of a change to an edge, label, or node, or at the meta-data level, consisting of the addition of a property space dimension. We discuss each kind of modification in detail.

3.3.1 Data updates

An edge can be inserted at any time into the database. On insertion, the transaction time of the property space region on the inserted edge is set to $[\mathit{current\ time} - \text{uc}]$.

Definition 3.18 [Edge insertion]
Let T be the current transaction time. An edge is inserted into a database, $DB = (V,\ E,\ \&\mathit{root},\ \Gamma)$, as follows.

$$\mathit{Insert}_{DB}(v \xrightarrow{\mathcal{R}} w) = (V \bigcup \{v,\ w\},\ E \bigcup \{v \xrightarrow{\mathcal{R}'} w\},\ \&\mathit{root},\ \Gamma)$$

where $\mathcal{R}' = \mathcal{R} \bigcup \{(\textbf{trans. time:}\ [T - \text{uc}])\}$. ∎

Redundant and overlapping property space regions are permitted on edges, i.e., the data is not stored *coalesced.* Note also that edge insertion inserts nodes if the nodes not already exist in the database. We do not give a separate operation to insert only a node (our focus is on the relevant changes needed to support property space).

Edges are (logically) deleted by terminating their transaction-time interval.

Definition 3.19 [Edge deletion]
Let T be the current transaction time. An edge is deleted from a database, $DB = (V,\ E,\ \&root,\ \Gamma)$, as follows.

$$Delete_{DB}(T,\ v \xrightarrow{\mathcal{R}} w) = (V,\ (E\ -\ \{v \xrightarrow{\mathcal{R}} w\}) \bigcup \{v \xrightarrow{\mathcal{R}'} w\},\ \&root,\ \Gamma)$$

where the region $\mathcal{R}'$ is exactly the same as $\mathcal{R}$ except in the transaction time property. If $\mathcal{R}$ has a transaction time property, say (**trans. time:** x), then

$$\mathcal{R}' = \mathcal{R} - \{(\textbf{trans. time: } x)\} \bigcup \{(\textbf{trans. time: } (x \bigcap [\text{beginning} - T]))\}.$$

Otherwise, if the transaction time property is missing from $\mathcal{R}$ then

$$\mathcal{R}' = \mathcal{R} \bigcup \{(\textbf{trans. time: } [\text{beginning} - T])\}.$$ ■

Finally, a node can be (logically) deleted by removing all incoming edges, and an edge modification is modeled as an edge deletion followed by an edge insertion.

Example 3.20 [The transactions for movie review]
The transactions that created the two edges in Figure 4 are given below. Let

- $v =$ `&Star Wars IV`,
- $w =$ `&by Videotastic`,
- $\mathcal{R}_1 = \{$(**name:** review), (**security!** developer), (**trans. time:** [15/Mar/1998 - uc])$\}$, and
- $\mathcal{R}_2 = \{$(**name:** review), (**security!** paid subsriber), (**trans. time:** [25/May/1998 - uc])$\}$.

On 15/Mar/1998, the first edge is inserted:	$Insert_{DB}(15/\text{Mar}/1998,\ v \xrightarrow{\mathcal{R}_1} w)$
On 24/May/1998, the first edge is deleted:	$Delete_{DB}(24/\text{May}/1998,\ v \xrightarrow{\mathcal{R}_1} w)$
On 25/May/1998, the second edge is inserted:	$Insert_{DB}(25/\text{May}/1998,\ v \xrightarrow{\mathcal{R}_2} w)$ ■

3.3.2 Adding and removing dimensions

Just as data evolves over transaction time, property space dimensions can also be added and (logically) deleted. This requires no changes to the data model.

A dimension may be added at any time. Each dimension consists of a unique *name*, a *domain* or type, and four operations: $PropertyCoalesce_p$, $PropertyCollapse_p$, $PropertySlice_p$, and $PropertyMatch_p$. A dimension is added by adding this information to the semantics of property space dimensions, Γ, within DB. For all existing labels, the new property is simply missing. When a property space region is subsequently inserted or updated, the new property can be used as needed.

A dimension can be deleted by removing the dimension semantics from Γ. Although existing labels in the data store will mention the property, the property is ignored in all subsequent operations (except for labels with a required property in the deleted dimension, which will fail to match any subsequent query). To save space, and remove required properties, the property should also be deleted from each edge, but this might be costly and disruptive.

This simple support for dimensions could be enhanced by keeping a history of dimension insertions and deletions. Then previous database states could be queried with the dimension semantics as of that previous state, but this issue of transaction time support for dimension changes is beyond the scope of this paper.

4 AUCQL

This section is a brief overview of an SQL-like query language, called AUCQL, for querying a semistructured database that has been extended with properties. AUCQL is like Lorel [AQM+97], but has additional constructs to permit queries to exploit properties. The focus of this presentation is on the small changes to the SELECT statement to support the extended query language operators discussed in the previous sections. Several examples of AUCQL are provided, and the reader is encouraged to interactively try these or other queries at the AUCQL website: `<http://www.cs.auc.dk/~curtis/AUCQL/index.html>`.

4.1 Variables in AUCQL

The key to understanding AUCQL is understanding the specification and use of variables. Variables in AUCQL are very much like variables in Lorel, the primary difference being that in AUCQL, a variable can range over the result of any of the extended query operators discussed in Section 3.2. Below is an AUCQL (or Lorel) query to find the names of movie stars.

```
SELECT N
FROM   movie.stars.name N;
```

(This is not the shortest, or best possible query, but is adequate for the purposes of this discussion). This query sets up a variable `N` that ranges over the terminal nodes of paths that match the regular expression `movie.stars.name`. In terms of the operations discussed in Section 3.2, the variable has the following meaning.

$\mathcal{R}_{movie} = \{(\textbf{name!}\ \text{movie})\}$
$\mathcal{R}_{stars} = \{(\textbf{name!}\ \text{stars})\}$
$\mathcal{R}_{name} = \{(\textbf{name!}\ \text{name})\}$
$N \in Nodes(Match_{DB}(ROOTS,\ \mathcal{R}_{movie}.\mathcal{R}_{stars}.\mathcal{R}_{name}))$

In fact, in AUCQL, this interpretation can be given explicitly.

```
SELECT N
FROM   NODES(MATCH(roots, (NAME! movie).(NAME! stars).(NAME! name))) N;
```

In AUCQL, a bareword descriptor (e.g., `movie`) defaults to a required use of the **name** property (e.g., to `(NAME! movie)`), since that will be the most commonly used property.

In general, the FROM clause is a list of *variable specifications*. The technical details are presented in Appendix A. The appendix has a complete BNF and denotational semantics for a value specification in AUCQL's FROM clause. Essentially, the formal details confirm that the values assigned to variables can be the result of any of the extended query operators. For example, the names of movie stars as of 'now' could be determined.

```
SELECT N
FROM   movie.stars S,
       S.(NAME! name, TRANSACTION_TIME: [now - now]) N;
```

The same query is given below, except that the extended query operators are explicitly coded.

```
SELECT N
FROM   MATCH(roots, (NAME! movie).(NAME! stars)) S,
       NODES(MATCH(S, (NAME! name, TRANSACTION_TIME: [now - now]))) N;
```

Alternatively, the paths as of now could be sliced from the semistructure.

```
SELECT N
FROM   SLICE((TRANSACTION_TIME: [now - now]), movie.stars.name) N;
```

Or perhaps, a user would like to determine which movie stars were added to the database this year.

```
SELECT N
FROM  movie.stars.name N,
      COALESCE(TRANSACTION_TIME, COLLAPSE(N)) TT
WHERE not (TT overlaps [beginning - 31/Dec/1997]);
```

In the above query, the meaning of the variable `N`, as a node or as a set of paths, is context-dependent. An alternative would be to use a path variable, e.g., `N@`, to distinguish the uses, but path variables are currently unsupported in AUCQL.

4.2 Failure in AUCQL

The `SLICE`, `MATCH`, `COLLAPSE`, and `COALESCE` operations may return an empty set. At least two strategies for coping with this result are possible.

- **null value** — An inapplicable null value is generated [AQM$^+$97]. This value indicates that the desired path, node, edge, or property is missing from the schema. This flexible strategy enables disjunctive queries (queries that have one or more disjuncts in the WHERE clause) to make progress on irregular schemas.
- **failure** — This "round" of query evaluation fails and so no variable assignments are generated. This strategy, while less flexible, ensures that only combinations of paths, nodes, and properties that actually exist in the graph are used in queries.

AUCQL uses the null-value strategy; essentially, the same is used in Lorel. For example, the following query would permit `Videotastic` subscribers to obtain the free film clips, which are available through their reviews, for Bruce Willis movies or movies that were panned, despite the fact that there is no text in the database for the `Star Wars IV` review.

```
SELECT R.clip
FROM  movie. M,
      M.(NAME! review, SECURITY: paid subscriber)) R,
WHERE M.stars.name like 'Bruce Willis' OR R.text like '%movie stinks%';
```

4.3 Defaults

Default properties can be set to simplify queries. Once a default is set, that value is used for the property in all subsequent operations. Properties specifically mentioned in an operation override their default values. The syntax for setting defaults is straightforward. Below is an example that retrieves movie stars' names that are current in the semi-structure.

```
SET DEFAULT PROPERTY (TRANSACTION_TIME: [now-now]);
SELECT movie.star.name;
```

Security is one of the most common default settings. A user can advertise their security certificates in all subsequent queries by setting a default.

```
SET DEFAULT PROPERTY (SECURITY: over 18 AND paid subscriber);
SELECT movie, movie.review.clip;
```

5 Related Work

This paper synthesizes research from several areas. There is an extensive body of research in semistructured and unstructured query languages, and several well-designed languages have been presented [BDS95, AQM$^+$97, LHL$^+$98, FFLS97, FLM98]. The closest related work in this area is the Chlorel query language for the DOEM data model [CAW98]. DOEM extends OEM with special annotations on edges to record information about updates; in particular, the (transaction) time and kind of update. This permits a history of changes to a semistructure to be maintained. We further extend the scope and power of the annotations on edge labels into a more general framework. Chlorel is a language for querying the extended data model. Chlorel supports a limited kind of temporal query, which lacks both coalescing and collapsing. We believe these operations are important to correctly supporting temporal semantics [BSS96].

Temporal database research has traditionally separated meta-data evolution (schema evolution, cf. [Rod92, RS95a]) from data evolution (transaction-time databases, cf. [JMR91, LS93, MS87, RS95b]). A transaction-time database records the history of tuple transactions, independent of table-level modifications. Schema evolution research on the other hand studies changes to the schema over time, independent of changes to the tuples within those tables. In a semistructured data model, there is not a crisp separation between data and meta-data. Instead, the schema is "folded into" the data, and modifications to both must be considered in tandem, along with modifications to other meta-data, such as security [CFMS94].

6 Summary and Future Work

This paper proposes an extensible framework for capturing more data semantics in semistructured data models. The framework is extensible so that it can incorporate the latest advances in diverse domains, from web security and e-commerce to transaction-time databases. The additional semantics for each domain are captured in enriched edge labels. The new labels are lists of descriptive properties. The properties used as examples in this paper include transaction time, price, security, quality, and valid time. But the properties do not have to be the same for every database or even for every label within a database since this framework permits missing properties. Support for required properties, to model dimensions such as security, is also built into the framework. Several new operations are needed to manipulate the enriched labels. Path *Match* chooses a set of paths from the semistructure that meet a user-given regular expression on the enriched labels. Path *Collapse* combines the properties in edges along a path to create a new label for the entire path. Path *Slice* slices a portion from each label on a path. Finally, edge *Coalesce* coalesces a property from a set of edges. These operations are built into the AUCQL query language. AUCQL is an implemented, Lorel-like query language, which is briefly described in this paper.

This work may be extended in a number of directions. Currently the semantics for properties are *statically scoped*. A single semantics for each dimension is supplied by a database designer. *Dynamic scoping* of the semantics would more closely model the lack of cooperation and organization among sites on the web. With dynamic scoping, the semantics of a property can be loaded along with the data, so the meaning of a property can change along a path as the path transits through various sites.

The query language could be extended to include so-called *sequenced* queries. A sequenced query computes the property space regions in one or more dimensions with respect to yet another dimension. For instance, to determine what quality ratings exists over what valid-time intervals, a user would give a sequenced query for quality with respect to valid time. The query might determine that low quality holds from 1995-1996, and high quality from 1997-now. Sequenced queries have been researched with respect to two dimensions in bitemporal databases, but not, to our knowledge, in more dimensions.

Labels can be further extended to include a *set* of property space regions. This does not greatly increase the modeling power since multiple descriptions of the same relationshp can be split into single regions on a number of edges. However, it is essential to storing coalesced property space regions, which may be of some convenience to the user.

Finally, and perhaps most importantly, the impact of our framework on path indexes must be addressed. We expect that a spatial or (bi)-temporal index can be generalized to index paths through property space, and we plan to investigate this issue in the future.

Acknowledgements

This research was supported in part by a grant from the Nykredit Corporation, by the Danish Technical Research Council through grant 9700780, and by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, as a Networks Activity of the Training and Mobility of Researchers Programme, contract no. FMRX-CT96-0056.

References

[AM98] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Proceedings of the International Conference on Data Engineering*, pages 24–33, Orlando, FL, February 1998. IEEE.

[AQM$^+$97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal of Digital Libraries*, 1(1):68–88, 1997.

[BDHS96] Peter Buneman, Susan B. Davidson, Gerd G. Hillebrand, and Dan Suciu. A Query Language and Optimization Techniques for Unstructured Data. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Quebec, Canada, 4–6 June 1996.

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *DBPL-5*, Gubbio, Italy, 1995.

[BL98] T. Berners-Lee. Keynote Address. In *Seventh International World Wide Web Conference (WWW7)*, Brisbane, Australia, April 1998.

[BSS96] M. Böhlen, R. Snodgrass, and M. Soo. Coalescing in Temporal Databases. In *Proceedings of the International Conference on Very Large Databases (VLDB '96)*, pages 180–191, Mumbai, India, September 1996.

[Bun97] P. Buneman. Semistructured Data. In *SIGMOD/PODS '97 (tutorial notes)*, Tucson, AZ, May 1997.

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *Proceedings of the International Conference on Data Engineering*, pages 4–13, Orlando, FL, February 1998. IEEE.

[CFMS94] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1994.

[CKR97] D. Connolly, R. Khare, and A. Rifkin. The Evolution of Web Documents: The Ascent of XML. *XML special issue of the World Wide Web Journal*, 2(4):119–128, Autumn 1997.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for a Web-Site Management System. *SIGMOD Record*, 26(3), September 1997.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[FS98] M. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the International Conference on Data Engineering*, pages 14–23, Orlando, FL, February 1998. IEEE.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the International Conference on Very Large Databases (VLDB '97)*, pages 436–445, Athens, Greece, September 1997.

[GW98] R. Goldman and J. Widom. Interactive Query and Search in Semistructured Databases. In *Proceedings of the First International Workshop on the Web and Databases (WebDB '98)*, pages 42–48, Valencia, Spain, March 1998.

[HGMC$^+$97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web. In *Proceeding of the Workshop on the Management of Semistructured Data (in association with SIGMOD'97)*, Tucson, AZ, June 1997.

[Hol89] R. Hole. *Basic Graph and Network Algorithms.* Addison-Wesley, 1989.

[JD98] C. Jensen and C. eds. Dyreson. *A Consensus Glossary of Temporal Database Concepts - February 1998 Version*, pages 367–405. Springer-Verlag, 1998.

[JMR91] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461–473, December 1991.

[JS94] C. S. Jensen and R. Snodgrass. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):954–974, 1994.

[LHL$^+$98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Datat with FLORID: A Deductive Object-Oriented Perspective. *to appear in Information Systems*, 1998.

[LS93] D. Lomet and B. Salzberg. *Transaction-Time Databases*, chapter 16, pages 388–417. Benjamin/Cummings, 1993.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MDS99] T. Milo and D. D. Suciu. Index Structures for Path Expressions. In *to appear in Proceedings of the International Conference on Database Theory (ICDT '99)*, 1999.

[MS87] E. McKenzie and R. Snodgrass. Extending the Relational Algebra to Support Transaction Time. In U. Dayal and I. Traiger, editors, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 467–478, San Francisco, CA, May 1987. Association for Computing Machinery.

[NAM97] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring Structure from Semistructured Data. In *Proceeding of the Workshop on the Management of Semistructured Data (in association with SIGMOD'97)*, Tucson, AZ, June 1997.

[QRU$^+$97] D. Quass, A. Rajaraman, J. D. Ullman, J. Widom, and Y. Sagiv. Querying Semistructured Heterogeneous Information. *Journal of Systems Integration*, 7(3/4):381–407, 1997.

[Rod92] J.F. Roddick. Schema Evolution in Database Systems — An Annotated Bibliography. *SIGMOD Record*, 21(4):35–40, December 1992.

[RS95a] J. F. Roddick and R. T. Snodgrass. Schema Versioning. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 22, pages 427–449. Kluwer Academic Publishers, 1995.

[RS95b] J. F. Roddick and R. T. Snodgrass. Transaction Time Support. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 17, pages 319–325. Kluwer Academic Publishers, 1995.

[Suc98] D. Suciu. Semistructured Data and XML. In *to appear in Proceedings of the International Conference on the Foundations of Data Organization (FODO '98)*, 1998.

A AUCQL's FROM Clause

In this discussion, the GROUP BY, HAVING, and ORDERING clauses in a SELECT statement are ignored. We also depend on the reader's understanding of the SELECT statement in SQL, and to a much lesser extent, in Lorel.

One interpretation of the meaning of a SELECT statement is that it has three main phases. First, the Cartesian product of the tables in the FROM clause is determined. The Cartesian product computes *all* of the information necessary to evaluate the query. Next, the WHERE clause predicate is evaluated for each tuple in the Cartesian product. Tuples that satisfy the WHERE clause predicate are retained. Finally, values from the satisfying tuples are projected as specified by the generalized attribute list in the SELECT clause.

In AUCQL only the first phase, the FROM clause, differs. The meaning of the other clauses remain essentially unchanged from SQL. The FROM clause in AUCQL also constructs a "Cartesian product" table, but not in the same way, nor with the same meaning as in an SQL query.

AUCQL's FROM clause is a list of *variable assignments*. Each variable in the list is assigned the value that results from one of the operations discussed in Section 3.2. There is one column in the Cartesian product table for each *variable* in the FROM clause, and one tuple in the table for each legal combination of variable assignments. Consider the example FROM clause below that collects movie star names and their transaction times.

```
FROM MATCH(roots, (NAME! movie).(NAME! stars).(NAME! name)) NamePath,
     NODES(NamePath) NameNode,
     COLLAPSE(NamePath) CollapsedName,
     COALESCE(TRANSACTION_TIME, CollapsedName) TransTime
```

The first assignment matches all the paths using a regular expression over the **name** property. The second assignment extracts the nodes from those paths. The third assignment collapses the paths to names. Finally, the fourth assignment coalesces the transaction time from the collapsed paths.

This FROM clause would build one table with four columns: `NamePath` through `TransTime`. The domains of the column in the table vary; the `NamePath` and `CollapsedName` columns are defined on *path* domains, `NameNode` is a *node* column, and `TransTime` is defined on the domain of sets of intervals.

Each tuple in the Cartesian product table represents a valid combination of column values. Some of the columns are *independent*, that is, their computation does not utilize a value in another column. The only independent column is `NamePath`. All combinations of values in independent columns is represented in the Cartesian product. But some columns are *dependent* on other column values. For instance, `NameNode` is dependent on `NamePath`. We assume that dependent columns are populated appropriately.

AUCQL supports the following BNF for the specification of a value (assigned to a variable).

```
⟨value spec⟩        ::= ⟨path⟩ | ⟨node⟩ | ⟨coalesced value⟩ | ⟨dimension value⟩
⟨path⟩              ::= ⟨path⟩ | ⟨collapsed path⟩ | ⟨sliced path⟩ | ⟨matched path⟩ | roots
⟨collapsed path⟩    ::= COLLAPSE ( ⟨path⟩ )
⟨sliced path⟩       ::= SLICE ( ⟨descriptor⟩ , ⟨path⟩ )
⟨matched path⟩      ::= MATCH ( ⟨path⟩ , ⟨descriptor regexp⟩ )
⟨node⟩              ::= ⟨identifier⟩ | NODES ( ⟨path⟩ )
⟨coalesced value⟩   ::= COALESCE ( ⟨dimension⟩ , ⟨collapsed path⟩ )
⟨descriptor regexp⟩ ::= regular expression over ⟨descriptor⟩s
⟨descriptor⟩        ::= ( ⟨property list⟩ )
⟨property list⟩     ::= ⟨property⟩ [ , ⟨property list⟩ ]
⟨property⟩          ::= ⟨dimension⟩ : ⟨literal⟩ | ⟨dimension⟩ ! ⟨literal⟩
⟨dimension⟩         ::= NAME | TRANSACTION_TIME | ... | VALID_TIME
```

$\langle$*dimension value*$\rangle$ ::= $\langle$*dimension*$\rangle$ ($\langle$*path*$\rangle$)

$\langle$*literal*$\rangle$ ::= $\langle$*string literal*$\rangle$ | $\langle$*integer literal*$\rangle$ | $\langle$*time literal*$\rangle$ | ... | $\langle$*identifier*$\rangle$

Below are whitespace and case-insensitive (the SQL defaults) examples of legal value specifications.

the descriptor for an edge matching the **name** movie
```
(NAME! movie)
```
the descriptor for matching movie.review
```
(NAME! movie).(NAME! review)
```
a variable M (previously matched to a set of nodes)
```
M
```
the descriptor for M.review
```
M.(NAME: review)
```
movie as of [1992-now]
```
(NAME! movie, TRANSACTION_TIME: [1992-now])
```
movie valid overlaps [1994-1998]
```
(NAME! movie).(VALID_TIME: [1994-1998])
```
movie reviews as of [1992-now]
```
(NAME! movie, TRANSACTION_TIME: [1992-now]).(NAME! review)
```
collapsed path to movie reviews
```
COLLAPSE(MATCH(roots, (NAME! movie).(NAME! reviews)))
```
movie transaction times coalesced
```
COALESCE(TRANSACTION_TIME, COLLAPSE(MATCH(roots, (NAME! movie))))
```

The denotational semantics for this BNF is given below. We assume that $Coordinates_p$ projects the property value for property p.

$[\![\,\texttt{MATCH(}\langle path\rangle\texttt{, }\langle descriptor\ regexp\rangle\texttt{)}\,]\!] \triangleq Match_{DB}([\![\,\langle path\rangle\,]\!], \langle descriptor\ regexp\rangle)$

$[\![\,\texttt{NODES(}\langle path\rangle\texttt{)}\,]\!] \triangleq Nodes([\![\,\langle path\rangle\,]\!])$

$[\![\,\texttt{SLICE(}\langle descriptor\rangle\texttt{, }\langle path\rangle\texttt{)}\,]\!] \triangleq Slice_{\Gamma}(\langle descriptor\rangle, [\![\,\langle path\rangle\,]\!])$

$[\![\,\texttt{COALESCE(}\langle dimension\rangle, \langle collapsed\ path\rangle\texttt{)}\,]\!] \triangleq Coalesce_{\Gamma}(\langle dimension\rangle, [\![\,\langle collapsed\ path\rangle\,]\!])$

$[\![\,\texttt{COLLAPSE(}\langle path\rangle\texttt{)}\,]\!] \triangleq Collapse_{\Gamma}([\![\,\langle path\rangle\,]\!])$

$[\![\,\langle dimension\rangle\texttt{(}\langle path\rangle\texttt{)}\,]\!] \triangleq Coordinates_{\langle dimension\rangle}([\![\,\langle path\rangle\,]\!])$

$[\![\,\texttt{roots}\,]\!] \triangleq ROOTS$

Once the Cartesian product table has been constructed, the WHERE and SELECT clauses are trivial to compute. Where a variable appears in one of these clauses, it just refers to the value in the appropriate column in the Cartesian product. Like Lorel, we assume that *nodes* are coerced as needed in expressions (e.g., in the expression `ReqNode = 'Ph.D.'`, if `ReqNode` represents a node that is a value, the value is retrieved, coerced to a string, and then compared).