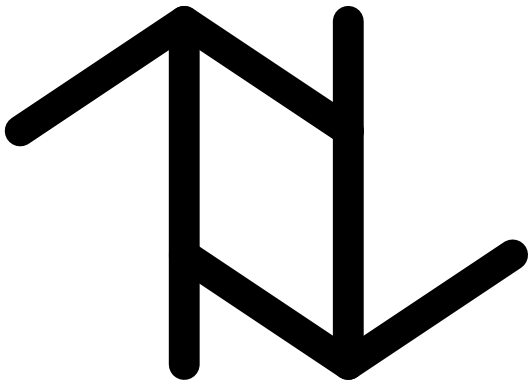


Capturing and Querying Multiple Aspects of Semistructured Data

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

March 26, 1999

TR-36

A TIMECENTER Technical Report

Title	Capturing and Querying Multiple Aspects of Semistructured Data
Author(s)	Curtis E. Dyreson, Michael H. Böhlen, and Christian S. Jensen
Publication History	November 1998. A TIMECENTER Technical Report. February 1999. Revised.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Michael H. Böhlen, Renato Busatto, Curtis E. Dyreson, Heidi Gregersen, Dieter Pfoser, Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Sudha Ram

Individual participants
Anindya Datta, Georgia Institute of Technology, USA
Kwang W. Nam, Chungbuk National University, Korea
Mario A. Nascimento, State University of Campinas and EMBRAPA, Brazil
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, TimeConsult, Switzerland
Vassilis Tsotras, Polytechnic University, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/research/DBS/tdb/TimeCenter/>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

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 and querying *properties* in a semistructured data model. The paper considers temporal properties 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 additional information is stored in properties associated with edges.

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

Contents

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), which includes a data definition language, is expected to replace the Hypertext Markup Language [CKR97, BL98]. An XML web 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 schemas may (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]. In addition, semistructured data models may also be applied to web meta-data, cf. the the RDF standard [LS99]. Somewhat unlike database meta-data, web meta-data is typically taken to mean additional information about a document, such as the author, subject classification, language, URL, etc. In this paper we use the term 'meta-data' to mean both database and web meta-data.

Semistructured data models organize data in graphs [Bun97, FLM98] where each node represents an object or a value, and each edge represents a relationship between the objects or values represented by the edge's nodes. 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].

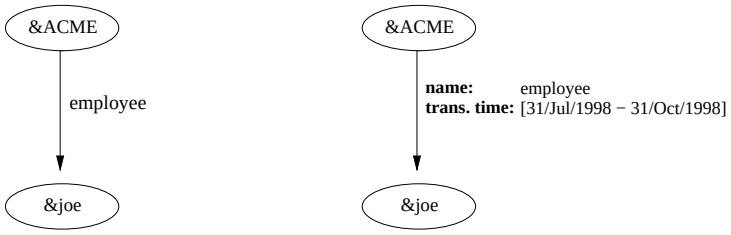
This paper introduces an extensible, semistructured data model that generalizes existing semistructured models. In this model, each label is a set of *descriptive* properties, such as the name of the edge, the time when the edge is valid in the real world, the time during which the edge exists as current 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 charged for using the edge in a query. These or any other properties can be used in a label to describe the nodes that are reachable through that edge.

To the best of our knowledge, this is the first work that treats labels as something other than single words or strings. 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, some of which are addressed in this paper, that have already been addressed in the contexts of different traditional data models. For example, only one paper has addressed support for temporal data [CAW98], and none (to the best of our knowledge) have addressed data security or quality.

To exemplify edge labels, consider Figure 1. Part (a) shows a conventional edge that is labeled 'employee' and connects nodes '&ACME' and '&joe'. In contrast, part (b) shows the kind of label introduced in this paper. This label is a set of '**property name**: *property value*' pairs. Each pair is collectively referred to as a property. This label has two properties: **name** and **transaction time**. The generalizes existing semistructures since the label in part (a) can be assumed to specify an implicit **name** property, with the value *employee.*

The paradigm of using labels with properties can be recursively applied. For instance, the label **name** in Figure 1(b) could itself be transformed into a label with two properties: **name** and **language**, e.g., English, indicating that **name** is an English word. While the recursive nature of labels with properties is theoretically appealing, it is of limited utility since meta-meta-data (and meta-meta-meta-data, etc.) is uncommon in the real-world. So while this framework could capture and query recursively nested properties, in this paper we focus exclusively on a single level of meta-data.

The paper is organized as follows. Section 2 motivates the extended semistructured model, arguing



(a) a typical edge (b) a label with properties

Figure 1: The new kind of labels

the utility of introducing a richer structure for labels. Section 3 presents the extended model. Initially, the format of a database is defined. An important feature is that the set of properties present may vary from label to label. Section 3.2 proceeds to introduce several new or extended query operators to contend with properties in labels. Section 4 incorporates the new query operations into a derivative of the SQL-like Lorel query language [QRU+97, MAG+97, AQM+97], called AUCQL, for querying semistructured data with properties. The remaining sections cover related work, future work, and summarize the paper.

The URL <`www.cs.auc.dk/~curtis/AUCQL`> provides an interactive query engine for the example database given in this paper, documentation and examples on using AUCQL, and a freely-available implementation package.

2 Motivation and background

This section aims to describe the new type of semistructured database proposed, with an emphasis on its background, the underlying design ideas, and its relation to existing semistructures.

2.1 An example database

A sample movie database spans semistructured data from a total of six sites. The *Internet Movie Database* site contains a wealth of movie data; *Videotastic* is a monthly, on-line movie industry magazine, portions of which are available only by subscription; the *Haus du Flicks* site charges a fee in e-cash for access to each of its many film clips, the fee being collected by an e-cash broker when a clip is accessed; the *Joe Doe* is a Yankee On-line User site devoted to science fiction movies; the site *Horsing Around Movies* has data about R- and NC-18 rated films, portions of which are restricted to web surfers over the age of 18; and the *Internet Archives* site offers movie data collected by a robot that periodically traverses part of the web.

Figure 2 shows a portion of the movie database. Edges are directed arrows, values are given in italics, and objects are depicted as ovals. The labels in the figure are sets of '**property name:** property value' pairs. This differs from a typical semistructured database where the label is just a single word or phrase. Most of the semistructure is not shown—many other edges and nodes exist in the complete movie database.

The database models the following pertinent facts. Information about a new movie, *Star Wars IV*, was added to the database on July/31/1998. A review of this movie appeared in the June issue of *Videotastic*, which was made available on 25/May/1988. The review is only available to paid subscribers. *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. *Haus du Flicks* that offers movie clips 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. Finally, *Horsing Around Movies*

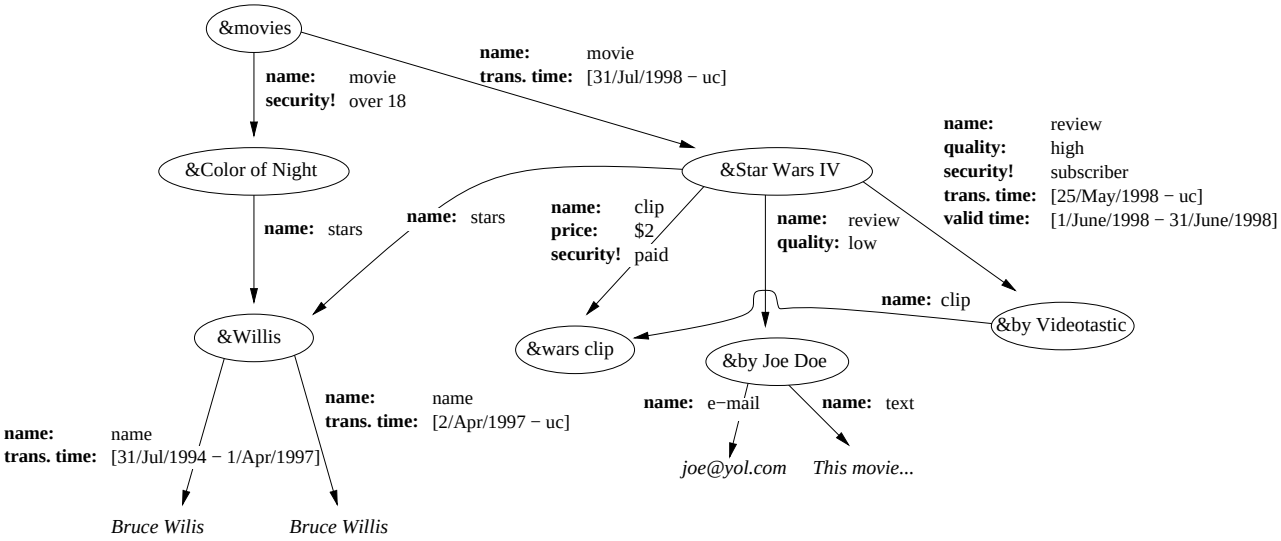


Figure 2: A web movie database

has data about the NC-18 rated movie, *Color of Night*, which also stars Bruce Willis. Only surfers with an appropriate security clearance are permitted to view this movie.

We will use this sample database for illustration throughout the paper.

2.2 Sample properties

The data model presented in this paper is capable of capturing the facts described above, in part, by using properties in labels. Labels are the most appropriate locations for properties since nodes are *completely* 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 '`&Willis`' '`stars`' in 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).

The data model is extensible, in that any properties may be used. Below, we discuss a partial list of such properties. None of them are *required* in a label. Indeed, for most labels, one or more properties may be *missing*.

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

security Some data has security restrictions, which are intended to indicate that only qualifying users are allowed access to the data. The essential ingredient to supporting this kind of security is to provide a method to restrict access to edges in queries. We use required properties for this purpose, as will be discussed further in Section 3.2.2.

This paper assumes that security is controlled through Netscape-like *certificates*[1]. So a more descriptive property name would be **security.netscape.read**, but for brevity we have shortened it. Several protocols exist for obtaining and managing these certificates. Once obtained, a certificate or combination of certificates

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

permits access to various services and documents. For clarity, we render a certificate in plain text rather than in its encrypted form. 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; and a security of 'over 18 OR subscriber' would mean that a either certificate alone would suffice. This is only one possible security property; the extensible data model can be used to support others.

transaction time The transaction time is the time when the edge is current in the database. It is called transaction time since it is the time interval between the time of the transaction that led to the edge and the time of the transaction that deleted or updated the edge [JD98]. Edges 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). The special role of transaction time in database modifications is elaborated in Section 3.4.

It should be noted that separate transaction-time timestamps on nodes are redundant 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 the transaction times of incoming edges, since an edge can only connect existing nodes. And if the node is current at a time when no incoming edge is current, then for that time, the node is a root node. Observe that associating a transaction time 'attribute' with each node would be insufficient since several edges may point to a node and a single attribute could not possibly distinguish between the lifetimes of the multiple descriptions of that node.

valid time The valid time of a database fact indicates when that fact is true in the modeled reality [JD98]. In our context, the valid-time property thus indicates when that edge reflects the real world. Often the valid time and the transaction time are the same, but this is only one of many possible relationships between these times [JS94]. Valid-time timestamps are closed intervals.

price When data is spread over a network, accessing some data may have substantially greater cost than other data, e.g., in terms of size, time, or money. The price property reflects these differing costs in obtaining data. Multiple price properties can comfortably coexist, but we assume that the price is a U.S. dollar amount, so more precisely the name of this property in the context of this example is **price.usdollars**.

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 and does not exclude other properties such as language, Dublin Core tags, or URL space.

2.3 Features of properties in labels

Many labels consist of several properties. For example, the two edges from the '`&Willis`' node shown in Figure 2 have the same value for the **name** property, but different transaction times. The most common property is **name**—only in unusual circumstances will an edge be unnamed.

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

The most important feature is that the label semistructures can also have irregular schemas, that is, a property found in one label can be *missing* from other labels. In Figure 2 the transaction time property is in only a few of the labels. Generally, a property is missing because it is *don't care* information, as in, this property is missing because we don't care if it is present, it is not germane to or will not improve the description of the data.

While irregular schemas are a feature of all semistructures, the next feature is unique, as far as we know, to our model. In a label, a property can be specified as being *required*. A required property is required to be matched in a query to gain access to nodes below that edge, but otherwise is just like any other property.[2] 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, i.e., an appropriate certificate, to traverse that edge. Further details on required properties are presented in Section 3.2.2.

There are few restrictions on the properties in labels. Common properties may be shared by a number of labels. Meta-data is often specified for a bag or container for a collection of objects [LS99]. Since a label is a set, it can easily be shared, in part or in whole, among a number of labels. In addition, multiple edges may connect the same pair of nodes with overlapping or redundant labels. Requiring labels to contain disjoint descriptions would be an unnecessary restriction.

Multiple properties in a label can capture more data semantics but 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: when the first edge in the path was inserted, the final edge was already deleted. So at no time did the two edges coexist in the current database state.

A contribution of this paper is a minimal collection of 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. *LabelMatch* compares two labels to determine if they match. This is a common operation performed in all queries when matching the labels in a path regular expression to the labels along a path in the semistructure. *Collapse* collapses entire paths to single edges that have their labels computed from the labels on each edge in the path. *Coalesce* computes the value of a property which is distributed among a number of different labels on edges between the same pair of nodes. Finally, *Slice* restructures the labels along paths by slicing a portion from selected properties in each label.

2.4 Contrast with existing semistructured data models

Our proposed model is not the only one capable of representing meta-data; existing semistructured data models with simple string labels can also explicitly capture meta-data. For example, the "property" information in a label could be encoded by splitting an edge into separate *data* and *meta-data* edges, with the properties branching from the end of the meta-data edge. But there are at least two problems with this approach of encoding the meta-data together with the data.

First, meta-data has no special status in such a model, so queries may unintentionally access meta-data rather than data. Consider the following Lorel-like query, which uses a wildcard to follow *all* paths in the database.

```
SELECT X FROM %* X
```

This query will retrieve both data and, unintentionally, meta-data. A user could formulate a query that follows only *data* edges, but this is challenging and, we believe, unnecessary. It should not be left to the user to guess how the meta-data is represented in the database and to write queries to explicitly avoid such data.

[2] A required property is similar to the **MUST** keyword in a proposal for privacy meta-data [W3C99].

A second, more fundamental problem, is that *some* of the meta-data has special semantics that must be accounted for in queries. For instance, assume that in a semistructured database with simple string labels, the transaction time for an object is represented as a **trans-time** edge from that node. As discussed above, a path is only valid if its edges are concurrent in the database—any other semantics is just wrong. Below we give a Lorel-like query to correctly retrieve only **movie.star.name**s that are concurrent (assuming that the INTERSECT operation computes the intersection of two time intervals).

```
SELECT N
FROM   movie M, M.star S, S.name N
WHERE  NOT_EMPTY(M.trans-time INTERSECT S.trans-time
                 INTERSECT N.trans-time)
```

The WHERE clause tests the transaction times of objects along the path to ensure that they are concurrent.

Although a user may explicitly formulate each query to correctly manipulate the transaction time and other properties, such a strategy has several highly undesirable features. First, all properties must be accounted for in all queries. For example, the query given above is incorrect since it does not correctly handle the security property. Second, the semantics of a property cannot be enforced. For example, a user could simply omit the WHERE clause in the query given above, or test some other condition on transaction time. The query will run to completion and return a result. But since the semantics of the transaction time property has not been observed, the result may include *fictive* paths. Third, naive users cannot formulate queries. A user has to know which properties exist, be familiar with the semantics of those properties, and must appropriately contend with all properties in every query. Fourth, queries become brittle. Even correctly formed queries will have a short shelf-life since adding a new property, or deleting an existing one can break existing queries.

In summary, it is theoretically possible, but unattractive and beyond the capabilities of users to represent and query properties using an ordinary semistructured database. The extensible data model presented in this paper can be viewed as, and perhaps can even be implemented as, a layer on top of a normal semistructured data model. The layer implements the semantics for each property and correctly translates queries and results between the user and the underlying database.

3 Extending a semistructured data model with properties

This section first defines a semistructure with properties, then defines the foundation necessary for querying such a semistructure, and finally considers update.

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 so-called property operations, Γ, that determine the semantics of properties. We also define $ROOTS \subseteq E$ to be the set of edges emanating from $\&root$. (These edges lead to what would normally be considered the roots of the semistructure; the extra level of indirection serves to record the properties of the root nodes.) An edge in E from node v to node w with the label $\mathcal{L}$ is denoted $v \xrightarrow{\mathcal{L}} w$. $\mathcal{L}$ is a *label with properties.*

Definition 3.1 [Label with properties]
A label with properties, $\mathcal{L}$, is a set of m pairs, $\{(p_1\colon x_1),\ (p_2\colon x_2),\ \ldots\ ,\ (p_m\colon x_m)\}$, where (i) each p_i is the *name* of a property, (ii) x_i is a *value* drawn from the property's domain, that is $x_i \in domain(p_i)$, (iii) property functions exist in Γ for each p_i, and (iv) each property name 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 $(p_i! \; x_i)$. ∎

Example 3.2 In Figure 2 a single edge connects '`&movies`' to '`&Color of Night`'. The label is the set of properties {(**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. ∎

To accommodate properties in queries, several operations for each property are needed (see Section 3.2). These operations determine the semantics of properties and are included in Γ.

Definition 3.3 [Property operations]
For each property p in a label, operations with the following signatures should be present in Γ. For brevity, let T be $domain(p)$.

- $PropertyCollapse_p : T \times T \rightarrow T \bigcup undefined$
- $PropertyMatch_p : T \times T \rightarrow BOOLEAN$
- $PropertyCoalesce_p : 2^T \rightarrow 2^T$
- $PropertySlice_p : T \times T \rightarrow T \bigcup undefined$ ∎

New properties may be introduced at any time, by registering the appropriate operations with the database. Default semantics are available for the operations as discussed in Section 3.4.2. The role of the property operations will be come clear when querying is considered, next.

3.2 Retrieving information from semistructures with properties

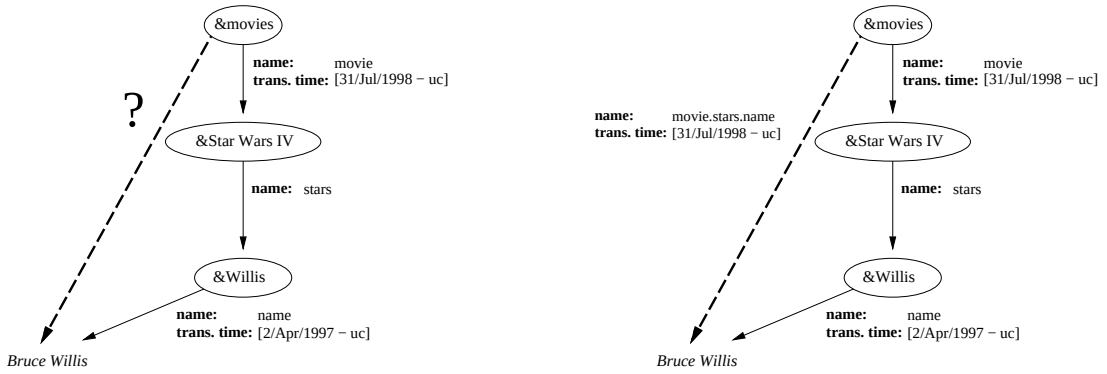
This section extends the information retrieval capability of an ordinary semistructured query language to handle labels with properties. Emphasis is on the query language aspects that are affected by the new labels. Thse aspects are quite localized, since labels are used only in so-called *path regular expressions* to traverse paths in the semistructure. There are two parts to the extension. First, when retrieving data, only *valid* paths should be followed, as discussed in Section 2. We define a *PathValid* predicate to test whether a path is valid by determining whether the path can be *collapsed* to a single edge with a label that preserves the information content of all the labels along the path. Second, path regular expressions must be generalized to support labels with properties and required properties within those labels. This involves redefining how labels are matched in the evaluation of an expression. We conclude by observing that the extension is strictly additive, the extended retrieval mechanism works as expected on a semistructure with simple string labels.

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" properties. The commonality is computed by collapsing the labels on 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, '`&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.

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



(a) uncollapsed (b) collapsed

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

Definition 3.4 [$CollapsePath_\Gamma : PATH \rightarrow EDGE$]
$CollapsePath_\Gamma$ takes a path and computes the label for the virtual edge between the first and last nodes in the path. The operation is extensible in that it depends on the semantics of the property as given by Γ. The constructor $PropertyCollapse_p$ in Γ is property-specific and is used to collapse a pair of property values for property p. In this operation, required properties are treated the same as other properties.

$$CollapsePath_\Gamma(v \xrightarrow{\mathcal{L}} w) \triangleq v \xrightarrow{\mathcal{L}} w$$

$$CollapsePath_\Gamma(v \xrightarrow{\mathcal{L}_1} u \xrightarrow{\mathcal{L}_2} w) \triangleq \; v \xrightarrow{\mathcal{L}} w \text{ where}$$

$$\begin{aligned}\mathcal{L} = &\{(p\colon PropertyCollapse_p(x,\ y) \mid (p\colon x) \in \mathcal{L}_1 \ \wedge \ (p\colon y) \in \mathcal{L}_2\} \bigcup \\ &\{(p\colon x) \mid (p\colon x) \in \mathcal{L}_1 \ \wedge \ (p\colon y) \notin \mathcal{L}_2\} \bigcup \\ &\{(p\colon y) \mid (p\colon x) \notin \mathcal{L}_1 \ \wedge \ (p\colon y) \in \mathcal{L}_2\}\end{aligned}$$

$$CollapsePath_\Gamma(v \xrightarrow{\mathcal{L}_1} u \xrightarrow{\mathcal{L}_2} \ldots \xrightarrow{\mathcal{L}_m} w) \triangleq CollapsePath_\Gamma(v \xrightarrow{\mathcal{L}_1} CollapsePath_\Gamma(x \xrightarrow{\mathcal{L}_2} \ldots \xrightarrow{\mathcal{L}_m} w))$$ ∎

The collapsing constructor, $PropertyCollapse_p$, depends on the semantics of the property. Appendix A.1 suggests constructors for a few common properties. In general, since each property is collapsed independently, the collapse constructor for a property should either be a *mutator*, which transforms one domain value into another, e.g., concatenation, or a *restrictor*, which reduces the extent of the domain value, e.g., time interval intersection.

Example 3.5 [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 transaction times on the edges on the path. It follows that the value *Bruce Willis* was described in the database as a `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 "collapsed" and then each property is checked to ensure that it is defined.

Definition 3.6 [$PathValid_\Gamma : PATH \rightarrow BOOLEAN$]
A path, P, is valid if after collapsing the path, there are no properties with *undefined* values.

$$PathValid_\Gamma(P) \triangleq \forall p\ [\ (p\colon undefined) \notin \mathcal{L} \wedge v \xrightarrow{\mathcal{L}} w = PathCollapse_\Gamma(P)\ ]$$ ∎

Example 3.7 [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 label has the value `movie.stars.name`. But the **transaction time** property is *undefined.* The transaction times of the first and last edges in the path are disjoint, and so their intersection does not produce a valid transaction time value. Consequently the path is not valid. ∎

The cost of checking path validity is $\mathcal{O}(n \cdot m)$, where n is the length of the path and m is the maximum number of properties in a label. We expect that in practice m will usually be much smaller than n. Path validity can be checked as a path is matched, as discussed next.

3.2.2 Path *Match*

In this section, we provide means of determining whether a user-given *descriptor* matches a label. 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 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 since each label is a semistructure. In addition, string comparison is insufficient because many properties are not strings.

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

- A *required* property in one label is *missing* from the other label. In this case, the match does not succeed. A required property must be present in both labels.
- A non-required property in one label is *missing* from the other label. In this case, the match succeeds because missing properties are treated as don't care information.
- The property is present in both labels. The predicate, $PropertyMatch_p$, specific to the property, is used to determine if the property values match. Required and non-required properties are treated the same.

Definition 3.8 [$LabelMatch_\Gamma : LABEL \times LABEL \rightarrow BOOLEAN$]
Label $\mathcal{L}$ is matched against label $\mathcal{S}$ as follows. *LabelMatch* depends on the semantics of the properties as specified in Γ, since properties in the labels are individually matched.

$$\begin{aligned} LabelMatch_\Gamma(\mathcal{L},\ \mathcal{S}) \triangleq\ & \forall p, x[(p!\ x) \in \mathcal{L} \Rightarrow \exists y[(p\colon y) \in \mathcal{S} \wedge PropertyMatch_p(x, y)]] \\ & \forall p, y[(p!\ y) \in \mathcal{S} \Rightarrow \exists x[(p\colon x) \in \mathcal{L} \wedge PropertyMatch_p(x, y)]] \\ & \forall p, x, y[(p\colon x) \in \mathcal{L} \wedge (p\colon y) \in \mathcal{S} \Rightarrow \wedge\ PropertyMatch_p(x, y)] \end{aligned}$$ ∎

Which definition to use for $PropertyMatch_p$ to match two values depends on the specifics of the property. For example, equality may be used for **name**, and time interval overlaps may be used for **transaction time**. Appendix A.2 discusses matching of common properties.

Example 3.9 [Looking for a movie]
Below is a label that requires a movie description.

$\mathcal{L}_{movie}$:= {(**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)}
$\mathcal{S}_{wars}$:= {(**name:** movie), (**trans-time:** [31/Jul/1998 - uc])}

These labels are matched as follows.

- $LabelMatch_\Gamma(\mathcal{L}_{movie},\ \mathcal{S}_{color}) = False$; the required **security**, over 18, is missing from $\mathcal{L}_{movie}$.
- $LabelMatch_\Gamma(\mathcal{L}_{movie},\ \mathcal{S}_{wars}) = True$; the extra **transaction time** property in $\mathcal{S}_{wars}$ is ignored. ∎

LabelMatch is the basis for interpreting regular expressions of descriptors. Generally, these regular expressions are interpreted exactly as in other semistructured query languages, and the usual features of regular expressions have their standard meaning.

Definition 3.10 [Regular language]
Let X be a well-formed regular expression over an alphabet of labels. Then $W(X)$ is the set of sequences of labels that form valid sentences for that expression, composed as follows. Symbol ϵ denotes the empty sequence.

$W(\mathcal{L}) = \{\mathcal{L}\}$ ($\mathcal{L}$ is a label)
$W(X.Y) = W(X)\ \times\ W(Y)$
$W(X*) = \{\epsilon\}\ \bigcup\ W(X+)$
$W(X+) = W(X)\ \bigcup\ W(X.X)\ \bigcup\ \ldots$
$W(X?) = \{\epsilon\}\ \bigcup\ W(X)$
$W(X|Y) = W(X)\ \bigcup\ W(Y)$ ∎

The only essential difference between our language and standard semistructured query languages is that the matched path is 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 path matches the regular expression and the path is valid.

Definition 3.11 [$Match_{DB} : 2^{PATH} \times REGEXP \rightarrow 2^{PATH}$]
A descriptor regular expression, X, *matches* a path, P, if the sequence of labels on the path is contained in $W(X)$, the regular language specified by X. This path may extend a path in a starting set of paths, S, as follows.

$$Match_{DB}(S,\ X) \triangleq \{v_1 \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_m} v_{m+1} \mid v_1 \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_{i-1}} v_i \in S\ \wedge$$
$$v_i \xrightarrow{\mathcal{L}_i} v_{i+1}, v_{i+1} \xrightarrow{\mathcal{L}_{i+1}} v_{i+2}, \ldots, v_m \xrightarrow{\mathcal{L}_m} v_{m+1} \in E\ \wedge$$
$$\mathcal{Y}_i \ldots \mathcal{Y}_m \in W(X)\ \wedge$$
$$\forall j[1 \leq j \leq m\ \wedge\ (LabelMatch_\Gamma(\mathcal{Y}_j, \mathcal{L}_j)]\ \wedge$$
$$PathValid_\Gamma(v_1 \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_m} v_{m+1})\}$$ ∎

We note that the presence of cycles in the semistructure can lead to an infinite result set, just like matching in any semistructured query language. Consequently, when this operation is implemented, some strategy must be adopted to either break cycles (e.g., node marking is used for Lorel) or otherwise generate a finite result sets (e.g., stop after the first N matches). Which strategy to use is a decision best left to a language designer; AUCQL uses node marking to break cycles.

The cost of *Match* is essentially the same as path matching in a normal semistructured database: at worst the entire semistructure is explored. The path validity can be computed as each path is explored, although it costs an extra factor of $\mathcal{O}(m)$, the number of properties in each label. *LabelMatch* is also an $\mathcal{O}(m)$ operation, assuming that the properties in a label are sorted or hashed. So overall, the cost of matching in our framework grows by a factor of the size of each label.

Sometimes only the set of final nodes in a set of paths is desired.

Definition 3.12 [$Nodes : 2^{PATHS} \rightarrow 2^{NODES}$]
Let P be a set of paths.

$$Nodes(P) \stackrel{\triangle}{=} \{w \mid v \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_m} w \in P\}$$ ∎

Example 3.13 [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.

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

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 without 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 a digital certificate that authenticates her or him 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 3.7).
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 Backwards compatibility

Compatibility with current semistructured models is achieved by assuming that string labels in those models implicitly default to **name** properties. Hence our framework can represent any existing semistructured database by modeling it as a database in which every label contains exactly one **name** property. Using the same default assumption, retrieval queries also remain unchanged. In existing semistructured databases all paths are valid. In our framework if every label consists of a single **name** property, then all paths are valid (**name**s are collapsed using string concatenation which never results in an *undefined* value). In existing semistructured databases, the labels are matched using string comparison, just like in our framework, so regular path expressions match exactly the same paths in both models. Finally, we observe that our framework seamlessly supports the mixing of data from existing semistructures with data that has richer meta-data since properties can vary from label to label. Hence as much or as little data as desired can be migrated to the new framework.

3.3 Additional query operators

In this section we present several query language operators that are useful when querying the information within labels. First, a label restructuring operation, called *Slice*, is given to carve a portion from each label on a path. Next, *PathCollapse* is trivially generalized to operate on the result of a *Match*. Finally, a *Coalesce* operation is defined to extract the value of a property that is distributed in several labels.

3.3.1 *Slice*

It is often useful to slice a portion from a property in each label 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 property in a label on the path, and checking whether the resulting path is valid.

Definition 3.14 $[Slice_\Gamma : LABEL \times 2^{PATHS} \rightarrow 2^{PATHS}]$
A descriptor, $\mathcal{L}$, *slices* the labels along each path in a set of paths, S, as follows.

$$Slice_\Gamma(\mathcal{L},\ S) \triangleq \{v \xrightarrow{\mathcal{L}'_1} \ldots \xrightarrow{\mathcal{L}'_m} w \mid v \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_m} w \in S \ \wedge\ PathValid_\Gamma(v \xrightarrow{\mathcal{L}'_1} \ldots \xrightarrow{\mathcal{L}'_m} w) \wedge \forall i\ [1 \leq i \leq m \wedge \mathcal{L}'_i = LabelSlice_\Gamma(\mathcal{L}, \mathcal{L}_i)]\}$$ ∎

A label is sliced property by property. This slicing is complicated by missing properties. Specifically, if a property is missing from the descriptor, but present in the label, it is passed unchanged into the result. A missing property in a label is 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 label and the descriptor then a property-specific constructor slices the property appropriately and adds it to the result.

Definition 3.15 $[LabelSlice_\Gamma : LABEL \times LABEL \rightarrow LABEL]$
A label, $\mathcal{L}$, *slices* a label, $\mathcal{S}$, as follows.

$$\begin{aligned} LabelSlice_\Gamma(\mathcal{L},\ \mathcal{S}) \triangleq\ & \{(p!\ PropertySlice_p(x,\ y)) \mid (p!\ x) \in \mathcal{L} \wedge ((p{:}\ y) \in \mathcal{S} \vee (p!\ y) \in \mathcal{S})\} \bigcup \\ & \{(p!\ PropertySlice_p(x,\ y)) \mid (p!\ y) \in \mathcal{S} \wedge ((p{:}\ x) \in \mathcal{L} \vee (p!\ x) \in \mathcal{L})\} \bigcup \\ & \{(p{:}\ PropertySlice_p(x,\ y)) \mid (p{:}\ x) \in \mathcal{L} \wedge (p{:}\ y) \in \mathcal{S}\} \bigcup \\ & \{(p!\ y) \mid (p!\ y) \in \mathcal{S} \wedge \neg\exists x[(p{:}\ x) \in \mathcal{L} \vee (p!\ x) \in \mathcal{L}]\} \bigcup \\ & \{(p!\ x) \mid (p!\ x) \in \mathcal{L} \wedge \neg\exists y[(p{:}\ y) \in \mathcal{L} \vee (p!\ y) \in \mathcal{L}]\} \bigcup \\ & \{(p{:}\ x) \mid (p{:}\ x) \in \mathcal{L} \wedge \neg\exists y[(p{:}y) \in \mathcal{S} \vee (p!\ y) \in \mathcal{S}]\} \end{aligned}$$ ∎

$PropertySlice_p$ is a property-specific constructor that slices a property. Appendix A.3 discusses the slicing of common properties.

Example 3.16 [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.

$\mathcal{L}_{movie}$:= {(**name!** movie)}
$\mathcal{L}_{stars}$:= {(**name!** stars)}
$\mathcal{L}_{name}$:= {(**name!** name)}
$\mathcal{L}_{now}$:= {(**trans-time:** [now - now])}
$Slice_\Gamma(\mathcal{L}_{now}, Match_{DB}(ROOTS,\ \mathcal{L}_{movie}.\mathcal{L}_{stars}.\mathcal{L}_{name}))$

Note that a $Slice_\Gamma$ 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. ∎

3.3.2 *Collapse*

In this section, the $PathCollapse_\Gamma$ 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 3.17 [$Collapse_\Gamma : 2^{PATHS} \rightarrow 2^{EDGES}$]
A set of paths, S, is collapsed by collapsing each path independently.

$$Collapse_\Gamma(S) \triangleq \{CollapsePath_\Gamma P \mid P \in S \ \wedge\ PathValid_\Gamma(P)\}$$ ∎

The utility of an operation like *Collapse* has been investigated in other semistructured query languages where it has been called "pull-up" [AKM94]. In Lorel, *Collapse* is not an operation at the query language level, rather it is used in the implementation to compute the value of a *path variable.*

3.3.3 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.

When several edges connect a pair of nodes, information about a single property may be distributed among multiple labels. 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 3.18 [$Coalesce_\Gamma : PROPERTY\ NAME \times 2^{EDGES} \rightarrow 2^{PROPERTY\ VALUES}$]
A set of edges, F, is coalesced for a *single* property as follows.

$$Coalesce_\Gamma(p, F) \triangleq$$
$$\{(v,\ (p\colon z),\ w) \mid z = PropertyCoalesce_p(\{x \mid ((p\colon x) \in \mathcal{L} \vee (p!\ x) \in \mathcal{L})\ \wedge\ v \xrightarrow{\mathcal{L}} w \in F\})\}$$ ∎

The *PropertyCoalesce* operation is a property-specific constructor. But, unlike the collapsing constructor, the coalescing constructor does not have to be a restrictor or mutator. Also, the result is not a label; rather, it is a list of both nodes and a single, coalesced property.

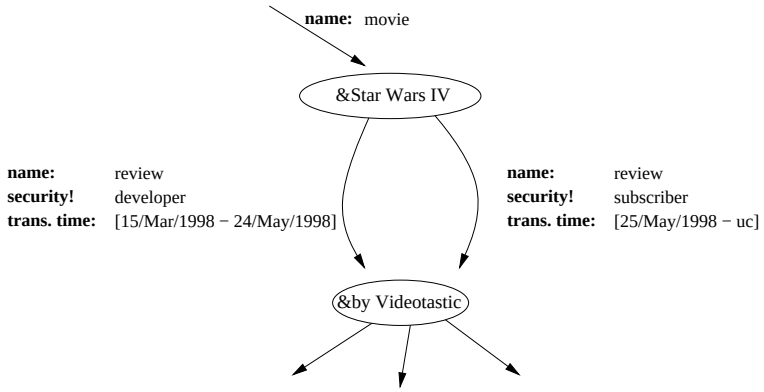


Figure 4: Evolving information about the *Videotastic* review of *Star Wars IV*

Example 3.19 [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.

$\mathcal{L}_{movie}$:= {(**name!** movie), (**security:** developer)}
$\mathcal{L}_{review}$:= {(**name!** review), (**security:** developer)}
$E := \{X \mid X = Collapse_{\Gamma}(Match_{DB}(ROOTS,\ \mathcal{L}_{movie}.\mathcal{L}_{review})\ \}$
$Coalesce_{\Gamma}$(**trans-time**, E)

The result is {(`&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. ∎

3.4 Updates

When transaction time is one of the supported properties, special semantics for update should 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. However, individual sites or even collections of pages within a site can be archived to correctly support transaction time. Because of the flexibility of our framework, information from pages that support transaction time can be freely mixed with information from pages that do not.

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. We discuss each kind of modification in detail.

3.4.1 Data updates

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

Definition 3.20 [Edge insertion]
Let T be the current transaction time. An edge is inserted into a database, $DB = (V,\ E,\ \&root,\ \Gamma)$, as follows.

$$Insert_{DB}(v \xrightarrow{\mathcal{L}} w) \triangleq (V \bigcup \{v,\ w\},\ E \bigcup \{v \xrightarrow{\mathcal{L}'} w\},\ \&root,\ \Gamma)$$

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

Redundant and overlapping labels 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 properties in labels).

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

Definition 3.21 [Edge deletion]
Let T be the current transaction time. An edge is deleted from a database, $DB = (V,\ E,\ \&root,\ \Gamma)$, as follows.

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

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

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

Otherwise, the transaction time property is missing from $\mathcal{L}$, so

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

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 3.22 [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`,
- $\mathcal{L}_1$:= {(**name:** review), (**security!** developer), (**trans-time:** [15/Mar/1998 - uc])}, and
- $\mathcal{L}_2$:= {(**name:** review), (**security!** paid subsriber), (**trans-time:** [25/May/1998 - uc])}.

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

3.4.2 Adding and removing properties

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

A property may be added to a label at any time. For all existing labels, the new property is simply missing. When a label is subsequently inserted or updated, the new property can be used as needed. Each property consists of a unique *name*, a *domain* or type, and four operations: $PropertyCoalesce_p$, $PropertyCollapse_p$, $PropertySlice_p$, and $PropertyMatch_p$. A database designer add this information to the semantics of properties, Γ, within *DB*. For most properties the default semantics for operations, given below, will suffice.

Definition 3.23 [Default property semantics]
Let t_1 and t_2 be any values for the property.

- $PropertyCollapse_p(t_1 \times t_2) = \lambda t_1\ t_2.\ t_2$
- $PropertyMatch_p(t_1, t_2) = \lambda t_1\ t_2.\ \textbf{if}\ t_1 = t_2\ \textbf{then}\ True\ \textbf{otherwise}\ False$
- $PropertySlice_p(t_1, t_2)$ = Semantic Error
- $PropertyCoalesce_p(\{t_1, \ldots, t_n\})$ = Semantic Error ∎

Two properties are by default collapsed to the second since paths are collapsed top-down, from a root to a leaf. The "closest" or most recent property to a leaf is taken to be the relevant property. Consider a **URL** property that gives the URL at which a datum resides. The URL of the page that contains the datum is more relevant than the URL of a parent page, and this is exactly what is computed by the default collapse constructor. Two properties match only if they are equal. No defaults are provided for $PropertyCoalesce_p$ and $PropertySlice_p$ since no reasonable, general default exists. Furthermore, these operations are only invoked by mentioning the property name in an additional, specific query language operation (they are in some sense optional).

A property can be deleted by removing the property 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 property, 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 properties can be enhanced by maintaining a history of property insertions and deletions as meta-meta-data. This can be accomplished by using name and transaction time properties within each label in the meta-data. Then previous database states can be queried with the properties available as of that previous state, but this issue of transaction time support for property 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: <`www.cs.auc.dk/~curtis/AUCQL`>.

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 Name
FROM   movie.stars.name Name;
```

(This is not the shortest, or best possible query, but is adequate for the purposes of this discussion). This query sets up a variable `Name` 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.

$\mathcal{L}_{movie}$:= {(**name!** movie)}
$\mathcal{L}_{stars}$:= {(**name!** stars)}
$\mathcal{L}_{name}$:= {(**name!** name)}
$Name \in Nodes(Match_{DB}(ROOTS,\ \mathcal{L}_{movie}.\mathcal{L}_{stars}.\mathcal{L}_{name}))$

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

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

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 B. 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 Name
FROM   movie.stars Star,
       Star.(NAME! name, TRANSACTION_TIME: [now - now]) Name;
```

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

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

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

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

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

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

In the above query, the meaning of the variable `Name`, 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 = 'Bruce Willis' OR R.text = '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 labels. The new labels are sets 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 properties such as security, is also built into the framework. Several new operations are needed to manipulate the labels with properties. *Match* chooses a set of paths from the semistructure that match a user-given path regular expression. *Collapse* combines the properties in labels along a path to create a new label for the entire path. *Slice* slices a portion from each label on a path. Finally, *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 property 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 labels in one or more properties with respect to yet another property. 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 properties in bitemporal databases, but not, to our knowledge, for more properties.

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

We also need to research translating meta-data in XML, such as RDF [LS99] or P3P [W3C99], to a set of properties. The translation should be relatively straightforward since there is a clear mapping between paths in an XML data-set and properties: each path maps to a property, the labels along the path collapse to the property's name, while the terminal value of the path is the property's value.

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 properties in labels, 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

[AKM94] S. Aggarwal, I. Kim, and W. Meng. Database Exploration with Dynamic Abstractions. In *Proceedings of DEXA'94*, September 1994.

[AM98] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Proceedings of IEEE ICDE'98*, pp. 24–33, February 1998.

[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] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of ACM SIGMOD'96*, pp. 505–516, June 1996.

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

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

[BSS96] M. Böhlen, R. Snodgrass, and M. Soo. Coalescing in Temporal Databases. In *Proceedings of VLDB'96)*, pp. 180–191, September 1996.

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

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *Proceedings of IEEE ICDE'98*, pp. 4–13, February 1998.

[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, 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 IEEE ICDE'98*, pp. 14–23, February 1998.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of VLDB'97*, pp. 436–445, 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*, pp. 42–48, 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)*, June 1997.

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

[JD98] C. S. Jensen and C. E. Dyreson, editors. *A Consensus Glossary of Temporal Database Concepts - February 1998 Version.* In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*, LNCS 1399, pp. 367–405. Springer-Verlag, 1998.

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

[JS94] C. S. Jensen and R. Snodgrass. Temporal Specialization and Generalization. *IEEE TKDE*, 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.*

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

[LS99] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Technical Report, January 1999.

[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. To appear in *Proceedings of the International Conference on Database Theory*, 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'87*, pp. 467–478, May 1987.

[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)*, 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, pp. 427–449. Kluwer, 1995.

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

[Suc98] D. Suciu. Semistructured Data and XML. In *Proceedings of the International Conference on the Foundations of Data Organization*, 1998.

[W3C99] W3C. Platform for Privacy Preferences (P3P1.0). W3C Technical Report, January 1999.

A Property specific operations

A.1 Collapsing of common properties

The $PropertyCollapse_p$ operation is used for collapsing properties when the properties of virtual edges are derived. The operation depends on the type of property, p. Below are given some suggested definitions of this operation for common properties.

- **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 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.

A.2 Matching of common properties

The $PropertyMatch_p$ predicate is used for matching properties in queries. Below are given some suggested definitions of this predicate for common properties.

- **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 label is the set of certificates owned by a user, and the security in the second label is the set of certificates required (expressed as a boolean formula). All certificates present in the first label 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 label 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 label should be equal to or larger than the price on the second label.
- **quality** ($\leq$) The quality of the first label must be less than or equal to that of the second label.

A.3 Slicing of common properties

Below are given some suggested definitions of the $PropertySlice_p$ operator for common properties.

- **name** (semantic error) Slicing does not have to be implemented for every property. Alternatively a substring operation could be used.
- **security** (conjunct elimination) This would limit the security certificates in a label 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 labels that have a price of $1 or less.
- **quality** (less-than pruning) Slicing can be used to eliminate low quality edges.

A.4 Coalescing of common properties

Typically a property is coalesced from a set of collapsed paths. The coalescing constructor $PropertyCoalesce_p$ depends on the semantics of the property.

- **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.

Coalescing, like slicing, is an optional operation for a property, and not all properties need to support it.

B 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*⟩ | ⟨*property 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 (` ⟨*property name*⟩ `,` ⟨*collapsed path*⟩ `)`
⟨*descriptor regexp*⟩ ::= regular expression over ⟨*descriptor*⟩s
⟨*descriptor*⟩ ::= `(` ⟨*property list*⟩ `)`
⟨*property list*⟩ ::= ⟨*property*⟩ [`,` ⟨*property list*⟩]
⟨*property*⟩ ::= ⟨*property name*⟩ `:` ⟨*literal*⟩ | ⟨*property name*⟩ `!` ⟨*literal*⟩
⟨*property name*⟩ ::= `NAME` | `TRANSACTION_TIME` | . . . | `VALID_TIME`
⟨*property value*⟩ ::= `PROPERTY (` ⟨*property name*⟩ `,` ⟨*path*⟩ `)`
⟨*literal*⟩ ::= ⟨*string literal*⟩ | ⟨*integer literal*⟩ | ⟨*time literal*⟩ | . . . | ⟨*identifier*⟩

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 $Value_p$ projects the property value for property p.

⟦ `MATCH(`⟨*path*⟩`,` ⟨*descriptor regexp*⟩`)` ⟧	$\triangleq Match_{DB}(⟦ \langle path \rangle ⟧, \langle descriptor\ regexp \rangle)$
⟦ `NODES(`⟨*path*⟩`)` ⟧	$\triangleq Nodes(⟦ \langle path \rangle ⟧)$
⟦ `SLICE(`⟨*descriptor*⟩`,` ⟨*path*⟩`)` ⟧	$\triangleq Slice_\Gamma(\langle descriptor \rangle, ⟦ \langle path \rangle ⟧)$
⟦ `COALESCE(`⟨*property name*⟩, ⟨*collapsed path*⟩`)` ⟧	$\triangleq Coalesce_\Gamma(\langle property\ name \rangle, ⟦ \langle collapsed\ path \rangle ⟧)$
⟦ `COLLAPSE(`⟨*path*⟩`)` ⟧	$\triangleq Collapse_\Gamma(⟦ \langle path \rangle ⟧)$
⟦ `PROPERTY((`⟨*property name*⟩`,`⟨*path*⟩`)` ⟧	$\triangleq Value_{\langle propertyname \rangle}(⟦ \langle path \rangle ⟧)$
⟦ `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).