Capturing and Querying Multiple Aspects of Semistructured Data

Curtis E. Dyreson
Department of Computer Science
James Cook University, Australia
curtis@cs.jcu.edu.au

Michael H. Böhlen Christian S. Jensen
Department of Computer Science
Aalborg University, Denmark
{boehlen,csj}@cs.auc.dk

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 web data. These models organize data in graphs where nodes denote objects or values and edges are labeled with single words or phrases. 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 meta-data *properties* in a semistructured data model. Properties such as temporal aspects of data, prices associated with data access, quality ratings associated with the data, and access restrictions on the data are considered. Specifically, the paper defines an extensible data model and an accompanying query language that provides new facilities for matching, slicing, collapsing, and coalescing properties. It also briefly introduces an implemented, SQL-like query language for the extended data model that includes additional constructs for the effective querying of graphs with properties.

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 [12, 4]. An XML web page can have a schema of how the data in the page is structured. XML will at best only provide some structure for data since the page-level schemas may (and likely will) vary from page to page. 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 [14, 26]. XML-like representation of web meta-data has also been proposed, cf. the RDF standard [21]. Somewhat unlike database meta-data, web meta-data is typically taken to mean additional information about a document, such as the author, subject, language, or URL. In this paper we use the term 'meta-data' to encompass both database and web meta-data.

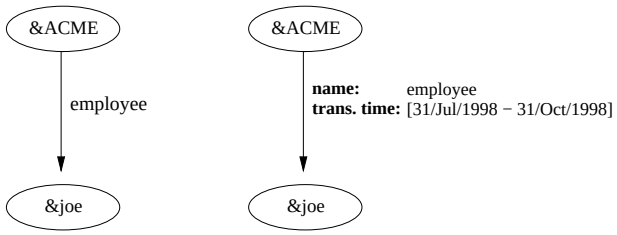
Semistructured data models organize data in graphs [8, 14] where each node represents an object or a value, and each edge represents a relationship between the objects or values represented by the edge's nodes. Edges are both directed and labeled. The labels are important because they make nodes *self-describing* in the sense that a node is described by the sequences of labels on paths through the graph that lead to the node [8].

This paper introduces an extensible, semistructured data model that generalizes existing semistructured models. In this model, each label is a set of descriptive *properties*. A property is a kind of meta-data. Typical properties are the name of the edge and the level of security that protects the edge, but any property can be used in a label to describe the nodes that are reachable through that edge.

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**. This 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 property **name** in Figure 1(b) could itself be transformed into a label with two properties: **name** and **language**, e.g., English, indicating

**Proceedings of the 25th VLDB Conference,
Edinburgh, Scotland, 1999.**



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

Figure 1: The New Kind of Edge Labels

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 although this framework could capture and query recursively nested properties, we focus exclusively on a single level of meta-data in this paper.

Previous research in semistructured and unstructured data models has focussed on basic issues such as query language design [6, 7, 25, 3, 20], restructuring of query results [13, 2], tools to help naive users query unknown semistructures [16, 17], techniques for improving implementation efficiency [25, 15, 23], and methods for extracting semistructured data from the web [18, 24]. Several well-designed languages have also been presented [6, 3, 20, 13, 14].

Our paper is different, in part, because it treats edge labels as something other than single words or strings. Buneman et al. also propose a semistructured model with complex labels [9]. In their model, key information from objects in the database is added to labels making each path in the database unique. We focus on adding meta-data rather than data to the labels and on the additional operations necessary to manipulate the meta-data in labels. Another paper with augmented labels presents the Chlorel query language for the DOEM data model [10]. 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 [5].

The paper is organized as follows. Section 2 motivates the extended semistructured model, arguing 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 [25, 22, 3], called AUCQL, for querying semistructured data with properties. The last section covers future work and summarizes 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, online 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; *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. 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 31/Jul/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* 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* 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.

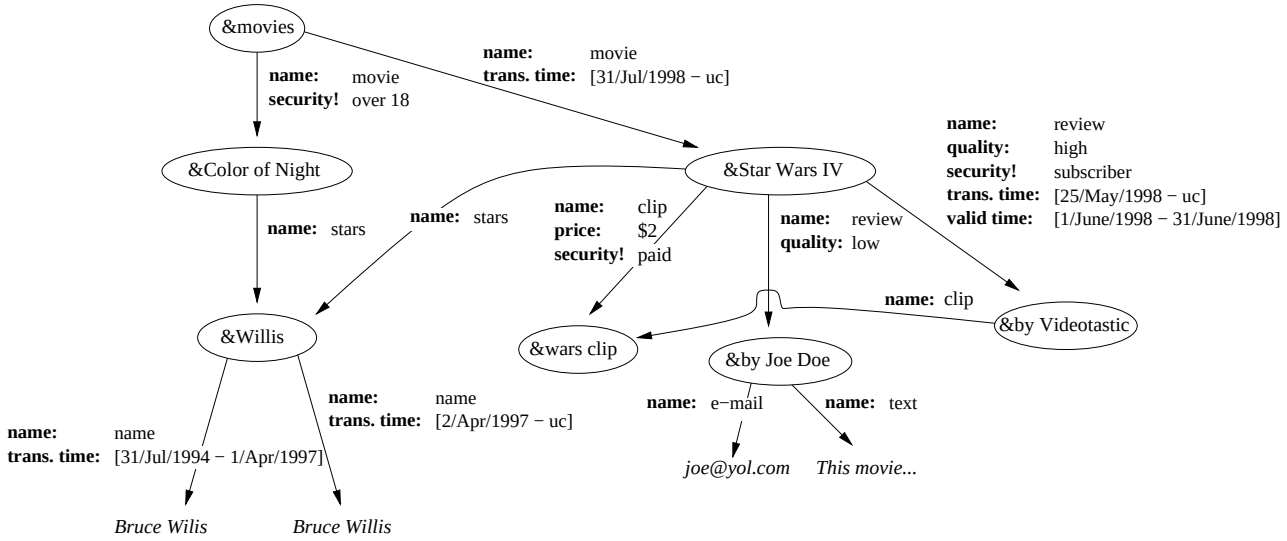


Figure 2: A Web Movie Database

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 in 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 mandatory. 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 is a set of names. For simplicity, in this paper our examples only use a single name.

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*. 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 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 either certificate alone would suffice. This is only one possible security property; the extensible data model can 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 [19]. 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.

valid time: The valid time of a database fact indicates when that fact is true in the modeled reality [19]. In our context, the valid-time property thus indicates when that edge reflects the real world. As for transaction time, 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 simply assume that the price is a U.S. 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 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.

The ability to accommodate the schema irregularities found in web data is an important feature of a semistructured (or unstructured) data model. In keeping with this requirement, the data model presented in the paper has several features worth mentioning.

One feature is that 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.

Another feature is that 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.[1] The **security** property on the edge to `&Color of Night` is a required property (indicated by affixing an '!' 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 [21]. 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.

This paper offers a collection of query language operators that support a more correct manipulation of 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. *Match* matches a so-called *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 a query that involves a *wildcard* (which matches any label) may unintentionally access meta-data rather than 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.

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 **ttime** edge from that node. As discussed above, a path is only valid if its edges are concurrent in the database—any other semantics is incorrect. 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.ttime INTERSECT
            S.ttime INTERSECT N.ttime)
```

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.

[1] A required property is similar to the **MUST** keyword in a proposal for privacy meta-data [27].

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 operations 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, an edge connects `&movies` to `&Color of Night`. The label is the set of properties {(**name:** movie), (**security!** over 18)}. The **security** property is a required 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, namely property collapse ($PrCl$), property match ($PrMa$), property coalesce ($PrCs$), and property slice ($PrSl$) (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 $domain(p)$.

- $PrCl_p : T \times T \rightarrow T \cup \{undefined\}$
- $PrMa_p : T \times T \rightarrow BOOLEAN$
- $PrCs_p : 2^{T \cup \{undefined\}} \rightarrow T \cup \{undefined\}$
- $PrSl_p : T \times T \rightarrow T \cup \{undefined\}$ □

These operations collapse, match, coalesce, and slice property values.

New properties may be introduced at any time by registering the appropriate operations with the database. Default semantics are available for the operations, as will be discussed in Section 3.4.2. Table 1 lists operations for one possible implementation of the properties discussed in this paper. The role of the property operations will become 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. These aspects are quite localized, since labels are used only in 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 *Valid* 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. 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 properties that share some "commonality." This commonality is computed by collapsing the labels on the path.

	name	valid time	security	transaction time	price	quality
$PrMa_p$	=	truth assignment	overlaps	overlaps	$\geq$	$\leq$
$PrCl_p$	concatenation	AND	intersection	intersection	sum	minimum
$PrCs_p$	union	OR	coalesce	coalesce	min	average
$PrSl_p$	semantic error	conjunct elimination	intersection	intersection	$>$ pruning	$<$ pruning

Table 1: Property Operations

Consider the case of a path to a movie star's name. One such path is shown in Figure 3, composed by the solid lines. Intuitively, the path forms a *virtual edge* from `&movies` to *Bruce Willis*. 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.

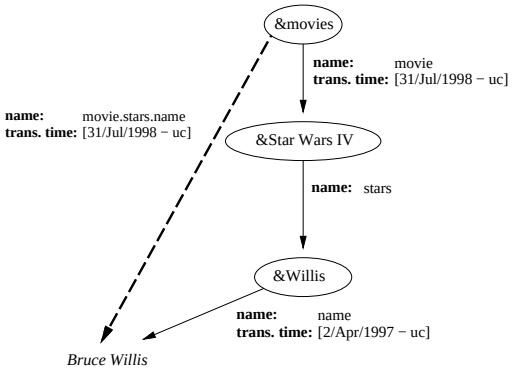


Figure 3: A (Virtual) Edge for the Name of a Movie Star

The operation described below collapses a path by recursively collapsing the 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. A missing property in a label is interpreted as "don't care information," meaning that any value of the missing 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. The path is collapsed backwards, that is, from the sink to the source, which effectively means that each collapsing constructor is left-associative.

Definition 3.4 $[ClPt_\Gamma : PATH \rightarrow EDGE]$
Collapse path ($ClPt_\Gamma$) 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 properties as given by Γ. Each constructor $PrCl_p$ 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.

$$ClPt_\Gamma(v \xrightarrow{\mathcal{L}} w) \stackrel{\triangle}{=} v \xrightarrow{\mathcal{L}} w$$

$$ClPt_\Gamma(v \xrightarrow{\mathcal{L}_1} u \xrightarrow{\mathcal{L}_2} w) \stackrel{\triangle}{=} v \xrightarrow{\mathcal{L}} w \text{ where}$$
$$\mathcal{L} = \{(p{:}\ PrCl_p(x,\ y) \mid (p{:}\ x) \in \mathcal{L}_1 \ \wedge\ (p{:}\ y) \in \mathcal{L}_2\} \cup$$
$$\{(p{:}\ x) \mid (p{:}\ x) \in \mathcal{L}_1 \ \wedge\ (p{:}\ y) \notin \mathcal{L}_2\} \cup$$
$$\{(p{:}\ y) \mid (p{:}\ x) \notin \mathcal{L}_1 \ \wedge\ (p{:}\ y) \in \mathcal{L}_2\}$$
$$ClPt_\Gamma(v \xrightarrow{\mathcal{L}_1} u \xrightarrow{\mathcal{L}_2} \ldots \xrightarrow{\mathcal{L}_m} w) \stackrel{\triangle}{=}$$
$$ClPt_\Gamma(v \xrightarrow{\mathcal{L}_1} ClPt_\Gamma(x \xrightarrow{\mathcal{L}_2} \ldots \xrightarrow{\mathcal{L}_m} w))$$ □

The collapsing constructor, $PrCl_p$, depends on the semantics of the property. Table 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** property in the collapsed path in Figure 3 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 $[Valid_\Gamma : PATH \rightarrow BOOLEAN]$
A path, P, is valid if after collapsing the path, there are no properties with *undefined* values.

$$Valid_\Gamma(P) \stackrel{\triangle}{=} \forall p\ [\ (p{:}undefined) \notin \mathcal{L} \wedge$$
$$(p!undefined) \notin \mathcal{L} \wedge$$
$$v \xrightarrow{\mathcal{L}} w = ClPt_\Gamma(P)\ ]$$ □

Example 3.7 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`. The **transaction time** property is *undefined*. The transaction times of the first and last edges in the path are disjoint, so their intersection does not produce a valid transaction time value. Consequently the path is invalid. □

The cost of checking path validity is $\mathcal{O}(n \cdot m)$, where n is the length of the path and m is the number of properties in a label. We expect that 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 first provide a means of determining whether a user-given *descriptor*, specified in a query, matches a label. The label matching operation is then incorporated into an *Match* operation to match a path regular expression to paths in the semistructure.

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 [3], which reuse the wildcard operator '%' from SQL. The descriptor `per%` would match any label that starts with 'per'.

The semantics of label matching is more involved in our model since each label is a set of properties. In addition, string comparison is insufficient because many properties are not strings. These complications are addressed in the label match operation *LaMa*, defined below. In general, operation *LaMa* succeeds if every individual property in the descriptor has a match in the label or is missing from the label. Extra properties in the label are ignored, and different $PrMa_p$ operations are used for different properties, p. Note that the descriptor is a label in the operator definition.

There are three cases to consider. (1) 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. (2) 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. (3) The property is present in both labels. The predicate, $PrMa_p$ 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 $[LaMa_\Gamma : LABEL \times LABEL \rightarrow BOOL]$ Label $\mathcal{L}$ is matched against label $\mathcal{S}$ as follows. *LaMa* depends on the semantics of the properties as specified in Γ, since properties in the labels are individually matched.

$$
\begin{aligned}
&LaMa_\Gamma(\mathcal{L},\ \mathcal{S}) \triangleq \\
&\quad \forall p, x[(p!\ x) \in \mathcal{L} \Rightarrow \exists y[(p\text{:}\ y) \in \mathcal{S} \wedge PrMa_p(x,y)]] \wedge \\
&\quad \forall p, y[(p!\ y) \in \mathcal{S} \Rightarrow \exists x[(p\text{:}\ x) \in \mathcal{L} \wedge PrMa_p(x,y)]] \wedge \\
&\quad \forall p, x, y[(p\text{:}\ x) \in \mathcal{L} \wedge (p\text{:}\ y) \in \mathcal{S} \Rightarrow PrMa_p(x,y)] \quad \square
\end{aligned}
$$

The property-specific predicate $PrMa_p$ matches two property values. For example, equality may be used for **name**, and time interval overlaps may be used for **transaction time**. See Table 1.

Example 3.9 The label that follows 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`.

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

These labels are matched as follows.

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

Operation *LaMa* 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 regular expression operations (+, *, ?, |, and . for sequencing) have their usual meaning. 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 an extended path matches the regular expression and the entire path is valid.

Definition 3.10 $[Match_{DB} : 2^{PATHS} \times REG \rightarrow 2^{PATHS}]$ Let S be a set of starting paths (typically the roots of the semistructure) and X be a regular expression over an alphabet of (extended) labels. Then X is said to match a path in $DB = (V,\ E,\ \&root,\ \Gamma)$ by extending a path in S as follows.

$Match_{DB}(S,\ X) \triangleq \{x \mid x \in M(S,X) \wedge Valid_\Gamma(x)\}$, where the matcher, M, is defined as follows.

$$
\begin{aligned}
M(S,\mathcal{L}) = \{&v_1 \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_m} v_{m+1} \mid \\
&v_1 \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_{m-1}} v_m \in S \wedge \\
&v_m \xrightarrow{\mathcal{L}_m} v_{m+1} \in E \wedge LaMa_\Gamma(\mathcal{L},\mathcal{L}_m)\} \\
M(S,X.Y) &= M(M(S,X),Y) \\
M(S,X*) &= S \cup M(S,X+) \\
M(S,X+) &= M(S,X.X*) \\
M(S,X?) &= S \cup M(X) \\
M(S,X|Y) &= M(S,X) \cup M(S,Y) \quad \square
\end{aligned}
$$

In the definition, the matcher M extends a path in S by recursively decomposing a path regular expression (the expression unifies with the second argument). The matcher extends a standard semistructured database matcher to use $LaMa_\Gamma$ to match individual labels, as discussed above.

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 $\mathcal{O}(m)$, where m is the number of properties in a label. $LaMa$ 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 3.11 $[Nodes : 2^{PATHS} \rightarrow 2^{NODES}]$
Let P be a set of paths.

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

Example 3.12 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.

It is instructive to consider four paths in Figure 2. (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. $\square$

3.2.3 Backwards Compatibility

Compatibility with current semistructured models is achieved by assuming that the string labels in those models 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, 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 also 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 path regular 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 use the new type of labels.

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 that carves a portion from each label on a path. Next, the previously defined *ClPt* operation 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 3.13 $[Slice_\Gamma : LABEL \times 2^{PATHS} \rightarrow 2^{PATHS}]$
A descriptor, $\mathcal{L}$, *slices* the labels along each path in a set of paths, P, as follows.

$$\begin{aligned} Slice_\Gamma(\mathcal{L},\ P) \stackrel{\triangle}{=} \{v \xrightarrow{\mathcal{L}'_1} \ldots \xrightarrow{\mathcal{L}'_m} w \mid \\ v \xrightarrow{\mathcal{L}_1} \ldots \xrightarrow{\mathcal{L}_m} w \in P \ \wedge \\ \mathcal{L}'_1 = LaSl_\Gamma(\mathcal{L}, \mathcal{L}_1) \wedge \ldots \wedge \mathcal{L}'_m = LaSl_\Gamma(\mathcal{L}, \mathcal{L}_m) \wedge \\ Valid_\Gamma(v \xrightarrow{\mathcal{L}'_1} \ldots \xrightarrow{\mathcal{L}'_m} w) \end{aligned} \qquad \square$$

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

$$
\begin{array}{l}
LaSl_\Gamma(\mathcal{L},\ \mathcal{S}) \triangleq \\
\quad \{(p!\ PrSl_p(x,\ y)) \mid \\
\qquad\qquad (p!\ x) \in \mathcal{L} \wedge ((p{:}\ y) \in \mathcal{S} \vee (p!\ y) \in \mathcal{S})\} \cup \\
\quad \{(p!\ PrSl_p(x,\ y)) \mid \\
\qquad\qquad (p!\ y) \in \mathcal{S} \wedge ((p{:}\ x) \in \mathcal{L} \vee (p!\ x) \in \mathcal{L})\} \cup \\
\quad \{(p{:}\ PrSl_p(x,\ y)) \mid (p{:}\ x) \in \mathcal{L} \wedge (p{:}\ y) \in \mathcal{S}\} \cup \\
\quad \{(p!\ y) \mid (p!\ y) \in \mathcal{S} \wedge \neg\exists x[(p{:}\ x) \in \mathcal{L} \vee (p!\ x) \in \mathcal{L}]\} \cup \\
\quad \{(p!\ x) \mid (p!\ x) \in \mathcal{L} \wedge \neg\exists y[(p{:}\ y) \in \mathcal{L} \vee (p!\ y) \in \mathcal{L}]\} \cup \\
\quad \{(p{:}\ x) \mid (p{:}\ x) \in \mathcal{L} \wedge \neg\exists y[(p{:}y) \in \mathcal{S} \vee (p!\ y) \in \mathcal{S}]\} \quad \square
\end{array}
$$

Recall that $PrSl_p$ is a property-specific constructor that slices a property. Table 1 shows the slicing operators.

Example 3.15 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.

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

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. $\square$

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

$$Collapse_\Gamma(S) \triangleq \{ClPt_\Gamma(P) \mid P \in S \ \wedge\ Valid_\Gamma(P)\} \quad \square$$

The utility of an operation like *Collapse* has been investigated in other semistructured query languages where it has been called "pull-up" [1]. 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 *Coalesce*

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.

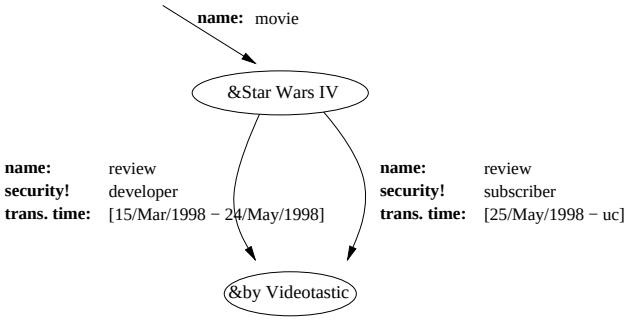


Figure 4: Evolving Information About a Review

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 3.17 [$Coalesce_\Gamma : NAME \times 2^{EDGES} \rightarrow VALUE$]
Assume that a set of edges, F, connects the same pair of nodes. F is coalesced for a *single* property, p, as follows.

$$
\begin{array}{l}
Coalesce_\Gamma(p, F) \triangleq PrCs_p(\\
\quad \{x \mid ((p{:}\ x) \in \mathcal{L} \ \vee\ (p!\ x) \in \mathcal{L} \ \wedge\ v \xrightarrow{\mathcal{L}} w \in F)\} \cup \\
\quad \{undefined \mid ((p{:}\ x) \notin \mathcal{L} \ \wedge\ (p!\ x) \notin \mathcal{L}) \ \wedge \\
\qquad\qquad v \xrightarrow{\mathcal{L}} w \in F\}) \quad \square
\end{array}
$$

The $PrCs_p$ operation is a property-specific constructor. Unlike the collapsing constructor, the coalescing constructor does not have to be a restrictor or mutator. Also, the result is not a label, but a single, coalesced value.

Example 3.18 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**s of the virtual edges.

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

3.4.1 Data Updates

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

Definition 3.19 [Edge insertion]
Let T be the current time. An edge is inserted into a semistructure, $DB = (V,\ E,\ \&root,\ \Gamma)$, as follows.

$$Insert_{DB}(T, v \xrightarrow{\mathcal{L}} w) \triangleq$$
$$(V \cup \{v,\ w\},\ E \cup \{v \xrightarrow{\mathcal{L}'} w\},\ \&root,\ \Gamma),$$
where $\mathcal{L}' = \mathcal{L} \cup \{(\textbf{transaction time:}\ [T - \text{uc}])\}$. □

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

$$Delete_{DB}(T,\ v \xrightarrow{\mathcal{L}} w) \triangleq$$
$$(V,\ (E - \{v \xrightarrow{\mathcal{L}} w\}) \cup \{v \xrightarrow{\mathcal{L}'} w\},\ \&root,\ \Gamma),$$
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 (**transaction time:** x), then

$$\mathcal{L}' = \mathcal{L} - \{(\textbf{transaction time:}\ x)\} \cup$$
$$\{(\textbf{transaction time:}\ (x \cap [\text{beginning} - T]))\}.$$

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

$$\mathcal{L}' = \mathcal{L} \cup \{(\textbf{transaction time:}\ [\text{beginning} - T])\}.$$ □

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 3.21 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)}, and
$\mathcal{L}_2 :=$ {(**name:** review), (**security!** paid subscriber)}.

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 time, properties can also be added and (logically) deleted.

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: $PrCl_p$ (collapse), $PrMa_p$ (match), $PrSl_p$ (slice), and $PrCs_p$ (coalesce). A database designer adds this information to the semantics of properties, Γ, within *DB*. For most properties, the default semantics for operations given below will suffice.

Definition 3.22 [Default property semantics]
Let t_1 and t_2 be any values for the property.

$PrCl_p(t_1, t_2) = \lambda t_1\ t_2.\ t_2$
$PrMa_p(t_1, t_2) = \lambda t_1\ t_2.\ t_1 = t_2$
$PrSl_p(t_1, t_2)$ = Semantic Error
$PrCs_p(\{t_1, \ldots, t_n\})$ = 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 and $PrSl_p$ and $PrCs_p$ since no reasonable, general defaults 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 offers a brief overview of an SQL-like query language, AUCQL, for querying a semistructured database that has been extended with properties. AUCQL is like Lorel [3], 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. The reader is encouraged to interactively try the AUCQL queries given here, 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.

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

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.

4.2 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 semistructure.

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

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

```
SET DEFAULT PROPERTY
    (SECURITY: over 18 AND subscriber);
```

5 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 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, an implemented, Lorel-like query language, which is briefly described in this paper.

This work may be extended in a number of directions. 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 [21] or P3P [27], 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, contract no. FMRX-CT96-0056.

References

[1] S. Aggarwal, I. Kim, and W. Meng. Database Exploration with Dynamic Abstractions. In *DEXA'94*, Sep. 1994.

[2] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *ICDE'98*, pp. 24–33, Feb. 1998.

[3] 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.

[4] T. Berners-Lee. Keynote Address. In *WWW7*, Apr. 1998.

[5] M. Böhlen, R. Snodgrass, and M. Soo. Coalescing in Temporal Databases. In *VLDB'96*, pp. 180–191, Sep. 1996.

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

[7] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *SIGMOD'96*, pp. 505–516, Jun. 1996.

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

[9] P. Buneman, A. Deutsch, and W.-C. Tan. A Deterministic Model for Semi-structured Data. In *ICDT'99 Workshop on the Web Query Languages*, Jan. 1999.

[10] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *ICDE'98*, pp. 4–13, Feb. 1998.

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

[12] 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.

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

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

[15] M. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *ICDE'98*, pp. 14–23, Feb. 1998.

[16] R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB'97*, pp. 436–445, Sep. 1997.

[17] R. Goldman and J. Widom. Interactive Query and Search in Semistructured Databases. In *the First International Workshop on the Web and Databases*, pp. 42–48, Mar. 1998.

[18] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web. In *the Workshop on the Management of Semistructured Data* (in association with *SIGMOD'97*), Jun. 1997.

[19] C. S. Jensen and C. E. Dyreson (eds.). *A Consensus Glossary of Temporal Database Concepts - February 1998 Version*. In O. Etzion et al. (eds.), *Temporal Databases: Research and Practice*, LNCS 1399, pp. 367–405. Springer-Verlag, 1998.

[20] 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*.

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

[22] 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, Sep. 1997.

[23] T. Milo and D. D. Suciu. Index Structures for Path Expressions. In *ICDT '99*, Jan. 1999.

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

[25] 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.

[26] D. Suciu. Semistructured Data and XML. In *FODO'98*, 1998.

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