UNIVERSITY OF CALIFORNIA, SAN DIEGO

Structured, Unstructured, and Semistructured Search in Semistructured Databases

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy
in
Computer Science

by

Andrey Balmin

2006

Dedicated to my parents, Lev and Olga.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

ACKNOWLEDGEMENTS

Many people have inspired, guided, supported and otherwise helped me to reach this point. I am most indebted to Yannis Papakonstantinou, my long-time advisor, one-time employer, and the committee chair. His database systems class introduced me to this area of computer science and prompted me to focus on it in graduate school. The employment experience in the start-up company that he founded nudged me in the direction of query processing on semi-structured and XML data and laid the groundwork for this dissertation. He guided my interests towards the intersection of database and information retrieval areas, long before this subject became *in vogue* in the academic community. Finally, he introduced me to many people at the IBM Almaden Research Center, which eventually led to my summer and permanent employment there.

I would like to thank all my collaborators and co-authors. It has been a great pleasure working with Vagelis Hristidis, with whom I collaborated on the XKeyword and ObjectRank projects. He exemplified the drive needed to succeed in graduate school and academic life.

A number of people have contributed to the incremental update validation project. First, I'd like to thank my collaborator Victor Vianu for showing me how theoretical results ought to be written down in a precise yet crystal clear way. I am also grateful to Jayavel Shanmugasundaram for useful discussions on this problem. Finally, I'm are grateful to the anonymous ACM TODS reviewers for their constructive comments on the paper.

I would like to thank my colleagues at the IBM Almaden Research Center, especially my internship mentor Fatma Ozcan and managers Roberta Cochrane and Hamid Pirahesh. They guided me in my search for hard problems that have great practical implications.

Finally, I would like to thank my family. My parents, Olga and Lev, and my wife, Lucia, provided invaluable intellectual and emotional support, and never doubted my success, despite whatever they might have been saying at the time.

VITA

1997	B.A. in Computer Science, University of California, San Diego
1999	M.S. in Computer Science, University of California, San Diego
2006	Ph.D. in Computer Science, University of California, San Diego

PUBLICATIONS

A. Balmin, Y. Papakonstantinou: Storing and querying XML data using denormalized relational databases. VLDB Journal 14(1): pages 30-49 (2005)

A. Balmin, Y. Papakonstantinou, V. Vianu Incremental validation of XML documents. In ACM Transactions on Database Systems (TODS) 29(4), December 2004.

A. Balmin, V. Hristidis, Y. Papakonstantinou: Authority-Based Keyword Queries in Databases using ObjectRank. In VLDB 2004: pages 564-575

A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, H. Pirahesh: A Framework for Using Materialized XPath Views in XML Query Processing. In VLDB 2004: pages 60-71

A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, T. Wang: A System for Keyword Search on XML Databases. In VLDB Demo Session, 2003: pages 1069-1072

V. Hristidis, Y. Papakonstantinou, A. Balmin: Keyword Proximity Search on XML Graphs. In ICDE 2003: pages 367-378

A. Balmin, Y. Papakonstantinou, T. Papadimitriou: Optimization of Hypothetical Queries in an OLAP Environment. In ICDE Poster Session, 2000: page 311

A. Balmin, T. Papadimitriou, Y. Papakonstantinou: Hypothetical Queries in an OLAP Environment. In VLDB 2000: pages 220-231

ABSTRACT OF THE DISSERTATION

Structured, Unstructured, and Semistructured Search in Semistructured Databases

by

Andrey Balmin
Doctor of Philosophy in Computer Science
University of California, San Diego, 2006
Professor Yannis Papakonstantinou, Chair

A single framework for storing and querying XML data, using denormalized schema decompositions, can support both structured queries and unstructured searches, as well as serve as a foundation for combining the two forms of information access.

XML data format becomes increasingly popular in applications that mix structured data and unstructured text. These applications require integration of structured query and text search mechanisms to access XML data.

First, we introduce a framework for storing and querying XML data using denormalized schema decompositions. This framework was initially implemented in the XCacheDB XML database system, which uses XML schemas to shred XML data into relational storage. The XCacheDB supports a subset of XQuery language and emphasizes query optimization to reduce latency and output first results quickly.

The XCacheDB relies on XML schemas, which poses a novel challenge for validation XML updates. We investigate the incremental validation of XML documents with respect to DTDs and XML Schemas. We exhibit an $O(m \log n)$ algorithm using an auxiliary structure of size $O(n)$, where n is the size of the document and m is the number of updates. We exhibit a restricted class of DTDs called “local” that arise commonly in practice and for which incremental validation can be done in practically constant time by maintaining only a list of counters. We present implementations and experimental evaluations of both general incremental validation and local validation in the XCacheDB system.

We, then, present XKeyword system which uses a variation of XCacheDB of schema decompositions to support keyword proximity searches in XML databases. XKeyword decompositions include *ID relations* which store of IDs of *target objects*, and pre-compute common joins.

Finally, we present an architecture of the Semi-Structured Search System (S^4) designed to bridge the gap between traditional database and information retrieval systems. S^4QL query language combines features of structured queries and text search to facilitate information discovery without knowledge of schema. S^4 is based on the same schema decomposition framework of XCacheDB and XKeyword. However, the combination structured and unstructured query features pose novel challenges to efficient query processing. We outline these issues and possible ways of addressing them.

Chapter I

Introduction

I.A Search in Databases

Information discovery, also known as search, is one of the basic information access operations, which involves retrieval of records that satisfy certain conditions. In database systems, search queries are important in both On-Line Transaction Processing (OLTP) and On-Line Analytical Processing (OLAP) paradigms. An OLTP transaction often includes search queries to identify a small set of records which need to be updated and/or returned to the user. An OLAP cube may be built from a set of records that is narrowed down by a search query. With the advent of e-commerce sites efficient search access became important to facilitate web-based catalog browsing. Typically, industrial relational database management systems support two kinds of search queries: *structured search* where comparison or range predicates are used to restrict values of certain fields, or *unstructured search*, also known as keyword search, which filters out records that do not contain some form of the specified keyword in a certain field.

Example 1 *Consider a catalog of consumer electronics products stored in a relational database with the following schema.*

$$Product(Type, Brand, Model, ScreenSize, Price, Description)$$

The product table contains a row for each product in the catalog. For example, `('TV','Sony','KX-27',27,300,'27" Trinitron TV with S-Video and composite inputs')` *An e-commerce site that sells products from this catalog may allow structured search queries that filter television set models based on their screen size, brand, price, etc. The same site may allow unstructured search, that would limit products to only those that contain a keyword in the product description. It might even be possible to mix structured and unstructured predicates. For example, a user could be looking for a Sony TV with screen size from 25 to 32 inches and S-Video connection. Since relational schema does not have a separate field for connection types, the web application will translate the user inputs into the following SQL query:*

```
SELECT * FROM Product
WHERE Type = 'TV' AND Brand = 'Sony'
AND ScreenSize Between 25 And 32
AND Description LIKE '%S-Video%'
```

Alternatively, a "LIKE" operator could be replaced by a call to a full-text search extension. Such extensions are currently provided by all major database vendors.

Notice that the web application in the above example needs to know the schema of the relational database to accurately translate user requests into SQL queries. Even keyword search has to be invoked on a specific column (in this case `Product.Description`).

A number of projects such as DISCOVER [42], BANKS [16], and DB Explorer [3], have investigated the *keyword proximity search*, which allows running unstructured search queries on the entire database, instead of a particular column, and does not require any schema knowledge from the user. These systems take advantage of the relational schema information to find groups of connected tuples that contain all keywords of the query. They model the database as a graph, where tuples are nodes and primary key to foreign key relationships are edges. Each node is associated with a set of keywords, which are contained in the attributes of the corresponding tuple. The goal of the proximity search is to find the smallest subtrees of the database graph that cover all the keywords of the query. Thus, keywords of the query may occur in different fields of a tuple, or even different, yet related, tuples. For example, a query ("TV","Sony","S-Video") will have effect similar to that of the query in Example 1. However, a keyword query cannot be used to approximate the SQL "between" predicate in this example.

I.A.1 Schema Evolution in Relational Databases

No real world application ever remains static. Applications and their databases have to adapt to new kinds of data and new requirements. Since the applications' query translation requires tight coupling with the database schema, any change to this schema has to be reflected in the applications' translation algorithm.

Consider adding LCD TV's to the catalog of the previous example. LCD's have many new searchable parameters that are not applicable for the traditional CRT TV's, such as native resolution, brightness and contrast. There are three typical solutions to this schema evolution problem. The first solution is to add new columns to the existing `Product` table. For the old products, all new columns are set to `NULL`. The downside of this approach is that after a few iterations of schema evolution the `Product` table will become very wide and sparse, i.e., will contain relatively few non-null values. Wide and sparse tables waste space and make query processing, including search, less efficient. The second approach to this schema evolution problem is to create a separate table for each type of the product. This design avoids sparse tables, however it has another drawback – application queries often have to scan large unions of tables. For example, if information about LCD and CRT TV's is stored in two different tables, the query of Example 1 would have to scan both tables to find all matching TV's. The large number of tables complicates query processing especially for queries that include joins. Finally, the third option is to store all features in a single table with three column

$$ProductFeaturs(ProductID, FeatureName, FeatureValue)$$

In this schema, a product's Type, Brand, Model, ScreenSize, Price, and Description would each be represented by a separate tuple. While being the most flexible, this approach presents serious query processing challenges since each selection condition on a column of the `Product` table, now becomes a join with the `ProductFeaturs`.

Notice that keyword proximity search systems support schema evolution much better than the structured search. For instance, the in the first two approaches, keyword proximity search can find tuples that contain keywords different fields. And in the third approach keyword proximity search can find the keywords in multiple tuples of the `ProductFeaturs` connected by the same `ProductID` [1].

However, the keyword search alone, clearly cannot satisfy all search needs. Another solution is needed, that would support efficient structured and unstructured search in presence of schema evolution. One such solution that has been gaining momentum is based on the eXtensible Markup Language (XML) [96] data format. The problem of schema evolution has recently been called "the killer app" for the XML databases [77].

I.A.2 Search in XML Databases

The need for search queries, which became clear in the relational databases, only increases with the arrival of XML databases. XML data is less rigid than relational. XML documents are not required to have a schema. Even if an XML document has a schema, the schema declaration may contain elements with arbitrary structure. This feature is especially popular for XML documents that contain free text, possibly with some markup. Different XML documents describing the same type of information may have different schemas if they have different origins. Furthermore, schemas of XML documents tend to evolve, and documents in the same collection may correspond to different versions of the same schema.

[1] Assuming that the `ProductID` column is the foreign key to some central `Product` table.

```
<Product>
   <Type> TV </Type>
   <Brand> Sony </Brand>
   <Model> KX-27 </Model>
   <ScreenSize>27</screenSize>
   <Price>300</Price>
   <Features>  <S-Video/> <Composite/> </Features>
   <Description> 27" Trinitron TV with S-Video and composite inputs
   </Description>
</Product>
```

Figure I.1: XML representation of the Sony TV catalog entry of Example 1

Due to this flexibility XML is less likely to be used in traditional OLTP and OLAP applications such as payroll systems or retail sales analysis. However, structured search is still very important in XML databases. For example, catalog search applications are more likely to use XML databases then relational, due to their natural support for schema evolution. Catalogs take advantage of XML flexibility, where different products can have different sets of searchable fields, and addition of a new product feature does not trigger a schema redesign.

There have been a number of proposals for XML query languages, but the most prominent one, at this time, is the XQuery [101] standard proposed by the World Wide Web Consortium (W3C). XQuery is very complex language with many features designed to appeal to different communities interested in the processing of XML data. We will limit ourselves to a subset of XQuery relevant to structured search. This subset is formally defined in Section II.E. The following example illustrates the use of XQuery for structured search.

Example 2 *Consider the XML fragment of Figure I.1, which represents the catalog entry of Example 1: Notice that the "S-Video" feature, which is likely to be queried by the user, can now be represented in markup, and not only as textual description. Thus a structured search query can find TV's with S-Video inputs*

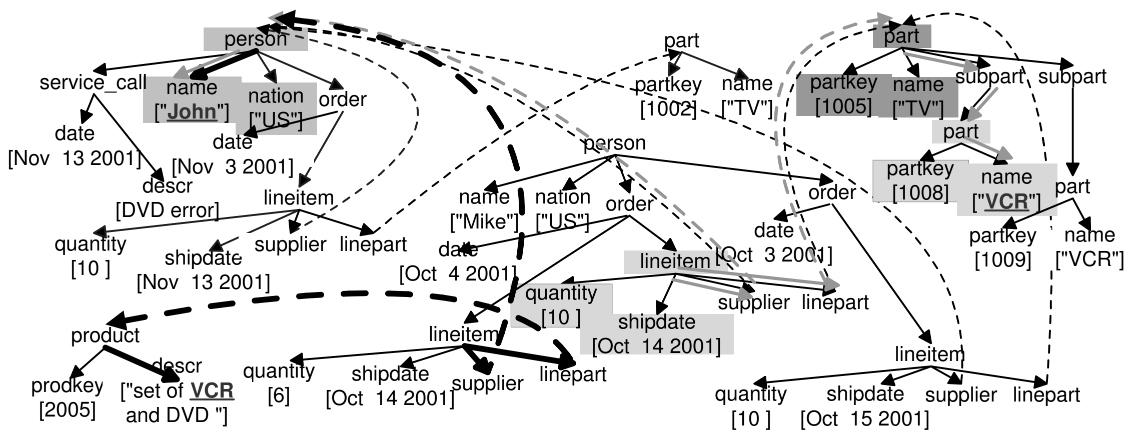


Figure I.2: Sample XML document

without using the keyword search capabilities. The following XQuery statement achieves the same effect as the SQL query of Example 1.

```
FOR $prod in collection(Products)/Product
WHERE Type = 'TV' AND Brand = 'Sony'
AND ScreenSize[. gt 25 AND . lt 32]
AND Features/S-Video
RETURN $prod
```

The fact that a relational text search predicate could be replaced by an XML structure predicate, does not mean that unstructured search in XML databases is less important than in relational systems. To the contrary, instead of a single, largely static relational schema, XML database users and application developers have to deal with a large number of evolving XML schemas. Thus, it is very important to be able to find information in XML databases without knowing exactly how this information is organized.

We investigate keyword proximity search in XML databases in the context of a XKeyword system described in Chapter IV. XKeyword modeled XML databases as labeled graphs, where edges correspond to element-subelement relationships and to IDREF pointers. The nodes of the graph are XML fragments (called *target objects*) that are semantically meaningful to the user, yet small enough to be easily displayed. Towards this direction, XKeyword associates a minimal piece of information, called *target object*, to each node and displays the target objects instead of the nodes in the results. A keyword proximity query is a set of keywords and the results are trees of target objects that contain all the keywords and are ranked according to the tree size. Trees of smaller sizes denote higher association between the keywords, which is generally true for reasonable schema designs.

Example 3 *Consider the data graph of Figure I.2, which combines product catalog information with customer orders. The keyword query "John, VCR" has two answers in this graph. The first highlighted tree (thick edges)* $name[John] \leftarrow person \leftarrow supplier \leftarrow lineitem \rightarrow linepart \rightarrow product \rightarrow descr[set\ of\ VCR\ and\ DVD]$ *on the source XML graph of Figure I.2 is a result of size 6. The second highlighted tree (gray arrows)* $name[John] \leftarrow person \leftarrow supplier \leftarrow lineitem \rightarrow linepart \rightarrow part \rightarrow subpart \rightarrow part \rightarrow name[VCR]$ *is a result of size 8. The highlighted boxes show the four* target objects *that cover the second result. For example, we display the target object* $part[partkey[1005], name[TV]]$ *in the place of the intermediate part node. We consider the first result to be "better" since the shorter distance corresponds to the closer connection between "John" and "VCR" in the first solution, where the "VCR" is the product that "John" supplied, as opposed to being a sub-part of another part supplied by "John". Notice that we allow edges to be followed in either direction.*

However, keyword search by itself does not satisfy all search needs. Even though, the users do not know the exact details of each XML schema used in the database, they may be aware of certain parts of the schemas and some logical entities and relationships that exist in the database. We argue that a new kind of search is needed in XML databases, which will combine elements of structured and unstructured search in a single query. This new kind of search should allow users to query the logical structure of the database and the known fragments of the schema using value predicates, joins, and other features of structured search, and at the same time, use keyword search and wildcards where the structure of the data is not known. We refer to this type of search as *semi-structured search* since it explores the space between structured search, which requires complete knowledge of the schema, and unstructured search, where users cannot take advantage of any schema information.

Example 4 *Consider the following semi-structured search query that looks for the Sony TV's with some connections to John.*

```
Product[Type = 'TV' AND Brand = 'Sony'
AND ScreenSize > 25 AND ScreenSize < 32 AND Features/S-Video
AND //*['John']]
```

Here the "//" is a wildcard, which implies a close semantic connection (but not necessarily ancestor-descendant) between the keyword "John" and the product we are searching for.*

Searching the mix of structured and unstructured XML

XML data is increasingly used to represent a mix of structured data and unstructured text. Such mixed data is yet another reason why semi-structured search is needed in XML databases. Providing only structured information discovery or only keyword search is not sufficient in modern enterprise information systems. It has long been the case that analytical applications can work only with clean and completely structured data. Any data without a strict structure can only be searched using keywords, or the data has to be cleaned up and normalized to fit into a fixed schema suitable for complex querying.

Transforming unstructured and semi-structured data to fit a fixed schema is an expensive, tedious, and error-prone process. This transformation process has to be constantly maintained as the underlying data and its structure evolves.

Due to high initial and maintenance costs, for most of the enterprise data that transformation is not practically feasible. Ability to analyze the data is critical to the success of the enterprise, and it is no longer acceptable that only a small fraction of data is available for analysis. Naturally, precise On-Line Analytical Processing (OLAP) style analysis is not possible on completely unstructured data. However, some middle-of-the-road approach is needed to investigate the trade-off between flexibility in data structure and complexity of the queries. Recently, there has been a strong interest in application of natural language processing (NLP) techniques to extract structured information from unstructured texts, even at the cost of some imprecision [86]. This illustrates the importance of unstructured data, and willingness of the users to accept a certain amount of error in analyzing this data.

Example 5 *For example, the product database of Figure I.2 could be extended to include customer complaints records, which could be analyzed by NLP tools to identify the types of the complaints and the defective parts of the products. A semi-structured search query could be used to find parts of Sony TV's that are related to John and had high severity complaints on them.*

```
Part[Product[Type = 'TV' AND Brand = 'Sony'
AND ScreenSize > 25 AND ScreenSize < 32 AND Features/S-Video]
AND //*['John'] AND //Complaint[Severity<2]]
```

I.B Structured Search in XML Databases

One approach towards building XML DBMS's is based on leveraging an underlying RDBMS for storing and querying the XML data. This approach allows the XML database to take advantage of mature relational technology, which provides reliability, scalability, high performance indices, concurrency control and other advanced functionality.

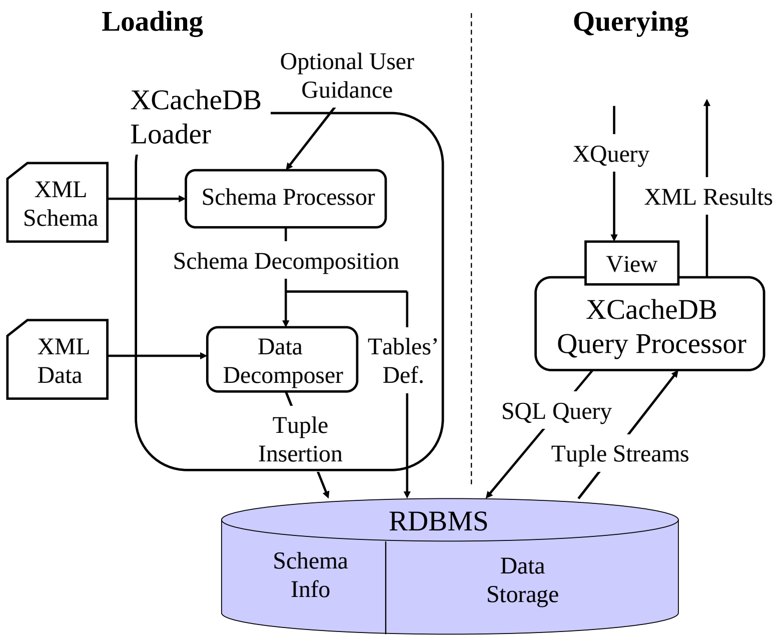


Figure I.3: The XML database architecture

We introduce a framework for schema-guided decomposition of XML data into relations. We identify a class of decompositions into non-normal form relations, called *inlined*, which allow us to trade storage space for query performance. The impact of inlined decompositions on query performance is especially notable in the reduction of *response time*, i.e. the time it takes to output the first results of a query, which makes them well suited for structured search.

We have implemented the inlined decompositions in the presented *XCacheDB* XML DBMS designed to support structured search [10]. XCacheDB follows the typical architecture (see Figure I.3) of an XML database built on top of a RDBMS [58, 75, 80, 17, 25]. First, XML data, accompanied by their XML Schema [91], is loaded into the database using the XCacheDB *loader*, which consists of two modules: the *schema processor* and the *data decomposer*. The schema processor inputs the XML Schema and creates in the underlying relational database tables required to store any document conforming to the given XML schema. The conversion of the XML schema into relational may use optional user guidance. The mapping from the XML schema to the relational is called *schema decomposition*.[2] The data decomposer converts XML documents conforming to the XML schema into tuples that are inserted into the relational database.

XML data loaded into the relational database are queried by the XCacheDB *query processor*. The processor exports an XML view identical to the imported XML data. A client issues an XML query against the view. The processor translates the query into one or more SQL queries and combines the result tuples into the XML result. Notice that the underlying relational database is transparent to the query client.

The key challenges in XML databases built on relational systems are

1. how to decompose the XML data into relational data,
2. how to translate the XML query into a plan that sends one or more SQL queries to the underlying RDBMS and constructs an XML result from the relational tuple streams.

A number of decomposition schemes have been proposed [80, 17, 33, 25]. However all prior works have adhered to decomposing into normalized relational schemas. Normalized decompositions convert an XML document into a typically large number of tuples of different relations. Performance is hurt when an XML query that asks for some parts of the original XML document results into an SQL query (or SQL queries) that has to perform a large number of joins to retrieve and reconstruct all the necessary information.

We provide a formal framework that describes a wide space of XML Schema-driven denormalized decompositions and we explore this space to optimize query performance. Note that denormalized decompositions may involve a set of relational design anomalies; namely, non-atomic values, functional dependencies and multivalued dependencies. Such anomalies introduce redundancy and impede the correct maintenance of the database [36]. However, given that the decompo-

[2]XCacheDB stores it in the relational database as well.

sition is transparent to the user, the introduced anomalies are irrelevant from a maintenance point of view. Moreover, the XML databases today are mostly used in web-based query systems where datasets are updated relatively infrequently and the query performance is crucial. Thus, in our analysis of the schema decompositions we focus primarily on their repercussions on query performance and secondarily on storage space and update speed.

The XCacheDB employs the most effective of the described decompositions. It employs two techniques that trade space for query performance by denormalizing the relational data.

- *non-Normal Form (non-NF) tables* eliminate many joins, along with the particularly expensive join start-up time.

- *Binary Large OBjects (BLOBs)* are used to store pre-parsed XML fragments, hence facilitating the construction of XML results. BLOBs eliminate the joins and "order by" clauses that are needed for the efficient grouping of the flat relational data into nested XML structures, as it was previously shown in [79].

Overall, both of the techniques have a positive impact on *total query execution time* in most cases. The results are most impressive when we measure the *response time*, i.e. the time it takes to output the first few fragments of the result.

Our main contributions in the area of structured search are:

- We provide a framework that organizes and formalizes a wide spectrum of decompositions of the XML data into relational databases.

- We classify the schema decompositions based on the dependencies in the produced relational schemas. We identify a class of mappings called *inlined decompositions* that allow us to considerably improve query performance by reducing the number of joins in a query, without a significant increase in the size of the database.

- We describe data decomposition, conversion of an XML query into an SQL query to the underlying RDBMS, and composition of the relational result into the XML result.
- We have built in the XCacheDB system the most effective of the possible decompositions.
- Our experiments demonstrate that under typical conditions certain denormalized decompositions provide significant improvements in query performance and especially in query response time. In some cases, we observed up to 400% improvement in total time (Figure II.22, Q1 with selectivity 0.1%) and 2-100 times in response time (Figure II.22, Q1 with selectivity above 10%).

I.C Unstructured Search in XML

Despite the structural flexibility of XML data, one still needs sufficient knowledge of the structure, role of the requested objects and XQuery in order to formulate a meaningful query to XCacheDB or any other XML database. Keyword search does not present such requirements; it enables information discovery by providing a simple and intuitive interface.

The search engines available today provide keyword search on top of sets of documents. When a set of keywords is provided by the user the search engine returns all documents that are associated with these keywords. Typically, a set of keywords and a document are associated if the keywords are contained in the document. Their degree of associativity is often their distance from each other.

XKeyword follows a recent generation of information retrieval systems that provide keyword proximity search [39, 42, 16, 3] to structured and semistructured databases. XKeyword differs from prior systems for proximity search on labeled graphs in that it assumes the existence of a schema, similar to the XML Schema standard [91], to which the graph conforms. The schema facilitates the

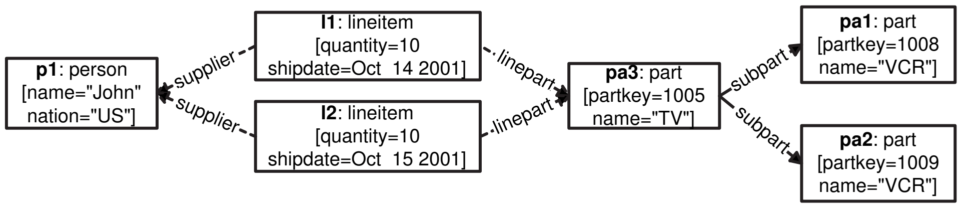


Figure I.4: Multivalued dependencies in results

presentation of the results and is also used in optimizing the performance of the system. Note that the end-user does not need to be aware of the schema.

The presentation of the results faces two key challenges that have not been addressed by prior systems. First, the results need to be semantically meaningful to the user. Towards this direction, XKeyword associates a minimal piece of information, called *target object*, to each node and displays the target objects instead of the nodes in the results. For example, in the query result of Figure I.4 we display the target object $part[partkey[1005], name[TV]]$ in the place of the intermediate *part* node $pa3$. Target objects are designated by the system administrator who splits the schema graph in minimal self-contained information pieces, which we call *Target Schema Segments (TSS)* and correspond to the target objects presented to the user. Furthermore, the edges connecting the target objects in the presentation graph are annotated with their semantic description, which is defined on the TSS graph. For example the $part \rightarrow part$ edge is named "*subpart*". The edge labels are displayed with the results to enhance the presentation.

The second challenge is to avoid overwhelming the user with a huge number of often trivial results, as is the case with DISCOVER [42] and DBXplorer [3][3]. Both of those systems present all trees that connect the keywords. In doing so they produce a large number of trees that contain the same pieces of information many times. For example, consider the result of the keyword query "US, VCR" shown in Figure I.4. This XML fragment contains four results:

[3]Both systems work on relational databases, but the presentation challenges are similar.

$$N_1 : p_1 \leftarrow l_1 \rightarrow pa_3 \rightarrow pa_1, \quad N_2 : p_1 \leftarrow l_2 \rightarrow pa_3 \rightarrow pa_2,$$
$$N_3 : p_1 \leftarrow l_2 \rightarrow pa_3 \rightarrow pa_1, \quad N_4 : p_1 \leftarrow l_1 \rightarrow pa_3 \rightarrow pa_2$$

The above results contain a form of redundancy similar to multivalued dependencies [36]: we can infer N_3 and N_4 from N_1 and N_2. In that sense, N_3 and N_4 are trivial, once N_1 and N_2 are given. Such trivial results penalize performance and overwhelm the user. XKeyword avoids producing "duplicate" results by employing a smart execution algorithm, and avoids the presentation "blow-up" by showing only a subset of the graph as it is formulated by various navigation actions of the user.

Two key challenges arise on the way to providing fast response times. First, the XML data has to be stored efficiently to allow the fast discovery of connections between the elements that contain the keywords. We employ XCacheDB database system which stores the XML data in a relational database (described in Chapter II), which we tune to provide the needed indexing and clustering. Then XKeyword builds a set of *connection relations*, which precompute particular path and tree connections on the TSS graph. Connection relations are similar to path indices [31] since they facilitate fast traversal of the database, but also different because they can connect more than two objects and they store the actual path between a set of target objects, which is needed in the answer of the keyword query. A core problem is the choice of the set of connection relations that are precomputed.

Second, the cost of computing the full presentation graph is very high. Hence XKeyword uses an on-demand execution method, where the execution is guided according to the user's navigation. We present an algorithm that generates a minimal set of queries to the underlying database in response to the user's navigation.

In summary, we make a number of contributions in the area of keyword proximity search:

- We present keyword proximity search semantics, extended to capture our novel result presentation method, which prevents information overflow and allows the user to navigate in the result.

- We present an architecture and framework that allows for choosing which connections between objects will be precomputed. We present rules to avoid generating any *useless* connection relation, i.e., connection relations that are not efficient to evaluate any CN. We show how to bound the number of joins needed to output a solution.
- We address the on-demand performance requirement of the presentation approach and we compare and analyze different decomposition schemes with respect to it. We also present an algorithm that efficiently generates the full list of results by caching partial results and avoiding to recompute the common result portions and show experimentally that it is up to 80% faster than the naive approach used in [42] and [3].

XKeyword has been implemented (Figure IV.2) and a demo is available at http://www.db.ucsd.edu/XKeyword, which operates on the XML data of the DBLP database.

I.D Semi-Structured Search System

To bridge the gap between traditional databases and information retrieval systems, we propose a Semi-Structured Search System (S^4) architecture. The goal is to enable a non-expert user to ask ad-hoc, information discovery queries without any knowledge of the dataset and its schema.

The power of S^4 comes from the combination of its three distinguishing features:

- XML datasets are modeled as graphs of entities and relationships. This conceptual model is much more intuitive to the user than a particular schema that implements the concept. It allows semantic querying without knowledge of element names or the structure of the hierarchy.

- The S^4 features efficient processing of tree pattern queries. A node of the query tree can be labeled with an entity name, a wildcard, a keyword, or a local predicate. Edges of the pattern are labeled with relationship names or wildcards.

- The results of the search are fragments of the XML dataset that satisfy the query conditions. The fragments are ordered by their query-specific rank, computed by a schema aware, “Page-Rank like” authority ranking mechanism.

We observe that many of the techniques designed to improve query processing performance in the XCacheDB and XKeyword systems, translate well into S^4 setting. The same framework of schema fragments that was used in the XCacheDB and XKeyword for XML data decomposition and storage can be used in a hybrid DB/IR system. The fragments are a convenient way to model S^4 materialized views, which are needed to provide reasonable query performance in S^4.

We do not claim to have solved the problem of bringing together the full power of database and information retrieval systems. Some serious technical challenges still need to be addressed before S^4 design can be successful in practice. For instance, the view containment problem is NP-hard. We propose an incomplete, but sound and efficient algorithm to decide view containment. However, there is more work to be done in identifying classes of queries and views where this (or other) algorithm will be successful. Also, the problems of query plan selection and view selection remain open. We outline these challenges and possible ways of addressing them.

I.E Supporting Updates in XML Databases

Up to now we have concentrated on query processing in XCacheDB, however, any database also has to support data updates. The XCacheDB implements XML updates by translating them into SQL update statements on the underlying relational database. However, the relational database cannot enforce XML structural integrity. External processing is needed to verify that an updated document still satisfies its schema. Doing this efficiently is a challenging problem. Brute-force validation from scratch is not practical when the data are large, because it may require reading and validating the entire database following each update. Instead, it is desirable to develop algorithms for incremental validation. However, this approach has been largely unexplored. We investigated the efficient incremental validation of updates to XML documents.

An XML document can be viewed abstractly as a tree of nested elements. The basic mechanism for specifying the type of XML documents is provided by Document Type Definitions (DTDs) [89]. DTDs can be abstracted as extended context-free grammars (CFGs). Unlike usual CFGs, the productions of extended CFGs have regular expressions on their right-hand sides. An XML document satisfies a DTD if its abstraction as a tree is a derivation tree of the extended CFG corresponding to the DTD.

Verifying that a word satisfies a regular expression[4] is the starting point in checking that an XML document satisfies a DTD. An obvious way to do this following an update is to verify it from scratch, i.e. run the updated sequence of labels through the non-deterministic finite automaton (NFA) corresponding to the regular expression. However, this requires $O(n)$ steps, under any reasonable set of unit operations, where n is the length of the sequence. We can do better by using incremental validation, relying on an appropriate auxiliary data structure. Indeed, we provide such a structure and corresponding incremental validation algorithm that, given a regular expression r, a string s of length n that satisfies r, and a se-

[4]A word *satisfies* a regular expression if it belongs to the corresponding language.

quence of m updates (inserts, deletes, label renamings) on s, checks[5] in $O(m \log n)$ whether the updated string satisfies r. The auxiliary structure we use materializes in advance relations that describe state transitions resulting from traversing certain substrings in s. These are placed in a balanced tree structure that is maintained similarly to B-trees and is well-behaved under insertions and deletions. The size of the auxiliary structure is $O(n)$. In addition, we provide an $O(m \log n)$ time algorithm that maintains the auxiliary structure, so that subsequent updates can also be incrementally validated.

Our approach to incremental validation of trees with respect to DTDs builds upon the incremental validation algorithm for strings. Incremental validation of m updates to a tree T with respect to a DTD can be done in time $O(m \log |T|)$ using an auxiliary structure of size $O(|T|)$ which can also be maintained in time $O(m \log |T|)$.

We then consider a restricted class of DTDs called "local", that arises very frequently in practice. Intuitively, these are DTDs using regular expressions for which membership after an update can be determined locally, by examining only substrings within bounded distance from the update position. This allows for a very efficient incremental validation algorithm. Although the theoretical worst-case data complexity of validating m updates is still $O(m \log |T|)$, and the auxiliary structure has size $O(i \log(|T|/i))$, where i is the number of internal nodes of T, the algorithm can be implemented very efficiently. Furthermore, if no internal nodes may be renamed, there is no need for an auxiliary data structure. In practice, the technique provides constant time validation, with space overhead of $O(i)$ counters that are represented very efficiently by usual 32-bit integers, which are sufficient when the maximum size of element lists in the database is 2^{32}. In addition to its simple implementation and efficiency, local validation is very frequent. We tested 60 DTDs from OASIS (see [20]), which contained 2141 complex regular expressions. Only 21 regular expressions in 10 DTDs were not local.

[5]For readability, we provide here the complexity with respect to the string and update sequence, for fixed DTD or regular expression. The combined complexity is spelled out in the Chapter III.

Finally, in the last part of the chapter, we evaluated the general DTD incremental validation algorithm, the validation algorithm for local DTDs, and a brute-force (re-)validation algorithm on an XML database that operates on top of a commercial RDBMS system. We describe the XML database and the implementation of the necessary data structures on top of an RDBMS. We discuss the applicability of the validation algorithm for local DTDs and a set of performance results that indicate its superiority over general incremental validation in the case of local DTDs. The experiments also quantify the significant superiority of both incremental validation algorithms over the brute-force technique. Finally, the experiments provide useful data for the optimization of various parameters of the data structures.

Chapter II

XCacheDB: RDBMS Backed Structured Querying of Semistructured Data

II.A Introduction

The acceptance and expansion of the XML model creates a need for XML database systems [80, 17, 25, 58, 75, 74, 48, 13, 63, 1, 38, 76, 29, 95]. Our approach towards building XML DBMS's is based on leveraging mature relational technology, which provides reliability, scalability, high performance indices, concurrency control and other advanced functionality.

We provide a formal framework for XML Schema-driven decompositions of the XML data into relational data. The described framework encompasses the decompositions described in prior work on XML Schema-driven decompositions [80, 17] and extends prior work with a wide range of decompositions that employ denormalized tables and binary-coded non-atomic XML fragments. The most effective among the set of the described decompositions have been implemented in the presented *XCacheDB*, an XML DBMS built on top of a commercial RDBMS [10].

A number of decomposition schemes have been proposed [80, 17, 33, 25]. However all prior works have adhered to decomposing into normalized relational schemas. Normalized decompositions convert an XML document into a typically large number of tuples of different relations. Performance is hurt when an XML query that asks for some parts of the original XML document results into an SQL query (or SQL queries) that has to perform a large number of joins to retrieve and reconstruct all the necessary information.

We provide a formal framework that describes a wide space of XML Schema-driven denormalized decompositions and we explore this space to optimize query performance. Note that denormalized decompositions may involve a set of relational design anomalies; namely, non-atomic values, functional dependencies and multivalued dependencies. Such anomalies introduce redundancy and impede the correct maintenance of the database [36]. However, given that the decomposition is transparent to the user, the introduced anomalies are irrelevant from a maintenance point of view. Moreover, the XCacheDB is designed primarily for use in web-based query systems where datasets are updated relatively infrequently and the query performance is crucial. Thus, in our analysis of the schema decompositions we focus primarily on their repercussions on query performance and secondarily on storage space and update speed.

The XCacheDB employs the most effective of the described decompositions. It employs two techniques that trade space for query performance by denormalizing the relational data.

- *non-Normal Form (non-NF) tables* eliminate many joins, along with the particularly expensive join start-up time.
- *BLOBs* are used to store pre-parsed XML fragments, hence facilitating the construction of XML results. BLOBs eliminate the joins and “order by” clauses that are needed for the efficient grouping of the flat relational data into nested XML structures, as it was previously shown in [79].

Overall, both of the techniques have a positive impact on *total query execution time* in most cases. The results are most impressive when we measure the *response time*, i.e. the time it takes to output the first few fragments of the result. Response time is important for web-based query systems where users tend to, first, issue under-constrained queries, for purposes of information discovery. They want to quickly, retrieve the first results and then issue a more precise query. At the same time, web interfaces do not need more than the first few results since the limited monitor space does not allow the display of too much data. Hence it is most important to produce the first few results quickly.

Our main contributions are:

- We provide a framework that organizes and formalizes a wide spectrum of decompositions of the XML data into relational databases.
- We classify the schema decompositions based on the dependencies in the produced relational schemas. We identify a class of mappings called *inlined decompositions* that allow us to considerably improve query performance by reducing the number of joins in a query, without a significant increase in the size of the database.
- We describe data decomposition, conversion of an XML query into an SQL query to the underlying RDBMS, and composition of the relational result into the XML result.
- We have built in the XCacheDB system the most effective of the possible decompositions.
- Our experiments demonstrate that under typical conditions certain denormalized decompositions provide significant improvements in query performance and especially in query response time. In some cases, we observed up to 400% improvement in total time (Figure II.22, Q1 with selectivity 0.1%) and 2-100 times in response time (Figure II.22, Q1 with selectivity above 10%).

The rest of this chapter is organized as follows. In Section II.B we discuss related work. In Section II.C, we present definitions and framework. Section II.D presents the decompositions of XML Schemas into sets of relations. In Section II.E, we present algorithms for translating the XML queries into SQL, and assembling the XML results. In Section II.F, we discuss the architecture of XCacheDB along with interesting implementation aspects. In Section II.G, we present the experiment results. We conclude and discuss directions for future work in Section II.H.

II.B Related Work

The use of relational databases for storing and querying XML has been advocated before by [17, 80, 33, 58, 25, 75]. Some of these works [33, 58, 25] did not assume knowledge of an XML schema. In particular, the Agora project employed a fixed relational schema, which stores a tuple per XML element. This approach is flexible but it is less competitive than the other approaches, because of the performance problems caused by the large number of joins in the resulting SQL queries. The STORED system [25] also employed a schema-less approach. However, STORED used data mining techniques to discover patterns in data and automatically generate XML-to-Relational mappings.

The works of [80] and [17] considered using DTD's and XML Schemas to guide mapping of XML documents into relations. [80] considered a number of decompositions leading to normalized tables. The "hybrid" approach, which provides the best performance, is identical to our "minimal 4NF decomposition". The other approaches of [80] can also be modeled by our framework. In one respect our model is more restrictive, as we only consider DAG schemas while [80] also takes into account cyclic schemas. It is possible to extend our approach to arbitrary schema graphs by utilizing their techniques. [17] studies horizontal and vertical partitioning of the minimal 4NF schemas. Their results are directly applicable in our case. However we chose not to experiment with those decompositions, since their effect, besides being already studied, tends to be less dramatic than the effect of producing denormalized relations. Note also that [17] uses a cost-based optimizer to find an optimal mapping for a given query mix. The query mix approach can benefit our work as well.

To the best of our knowledge, this is the first work to use denormalized decompositions to enhance query performance.

There are also other related works in the intersection of relational databases and XML. The construction of XML results from relational data was studied by [79, 32, 30]. [79] considered a variety of techniques for grouping and tagging results of the relational queries to produce the XML documents. It is interesting to note the comparison between the "sorted outer union" approach and BLOBs, which significantly improve query performance. The SilkRoute [32, 30] considered using multiple SQL queries to answer a single XML Query and specified the optimal approach for various situations, which are applicable in our case as well.

Oracle 8i/9i, IBM DB2, and Microsoft SQL Server provide some basic XML support [13, 74, 48]. None of these products support XQuery or any other full-featured XML query language.

Another approach towards storing and querying XML is based on native XML and OODB technologies [76, 63, 1, 38]. The BLOBs resemble the common object-oriented technique of clustering together objects that are likely to be queried and retrieved jointly [12]. Also, the non-normal form relations that we use are similar to path indices, such as the "access support relations" proposed by Kemper and Moerkotte [50]. An important difference is that we store data together with an index, similarly to Oracle's "index organized tables" [13].

A number of commercial XML databases are available. Some of these systems [51, 27, 59] only support API data access and are effectively persistent implementations of the Document Object Model [88]. However, most of the systems [22, 29, 98, 43, 45, 94, 64, 76, 66, 95, 6, 104] implement the XPath query language or its variations. Some vendors [76, 29, 64] have announced XQuery [101] support in the upcoming versions, however only X-Hive 3.0 XQuery processor [95] and Ipedo XML Database [45] were publically available at the time of writing.

The majority of the above systems use native XML storage, but some [29, 95, 94] are implemented on top of object-oriented databases. Besides the query processing some of the commercial XML databases support full text searches [45, 95, 104], transactional updates [22, 29, 98, 45, 94, 64] and document versioning [45, 94].

Even though XPath does not support heterogeneous joins, some systems [76, 66] recognize their importance for the data integration applications and provide facilities that enable this feature.

Our work concentrates on selection and join queries. Another important class of XML queries involve path expressions. A number of schemes [54, 46] have been proposed recently that employ various node numbering techniques to facilitate evaluation of path expressions. For instance, [54] proposes to use pairs of numbers (start position and sub-tree size) to identify nodes. The XSearch system [103] employs Dewey encoding of node IDs to quickly test for ancestor-descendant relationships. These techniques can be applied in the context of XCacheDB, since the only restriction that we place on node IDs is their uniqueness.

II.C Framework

We use the conventional labeled tree notation to represent XML data. The nodes of the tree correspond to XML elements, and are labeled with the tags of the corresponding elements. Tags that start with the "@" symbol stand for attributes. Leaf nodes may also be labeled with values that correspond to the string content.

Note that we treat XML as a database model that allows for rich structures that contain nesting, irregularities, and structural variance across the objects. We assume the presence of XML Schema, and expect the data to be accessed via an

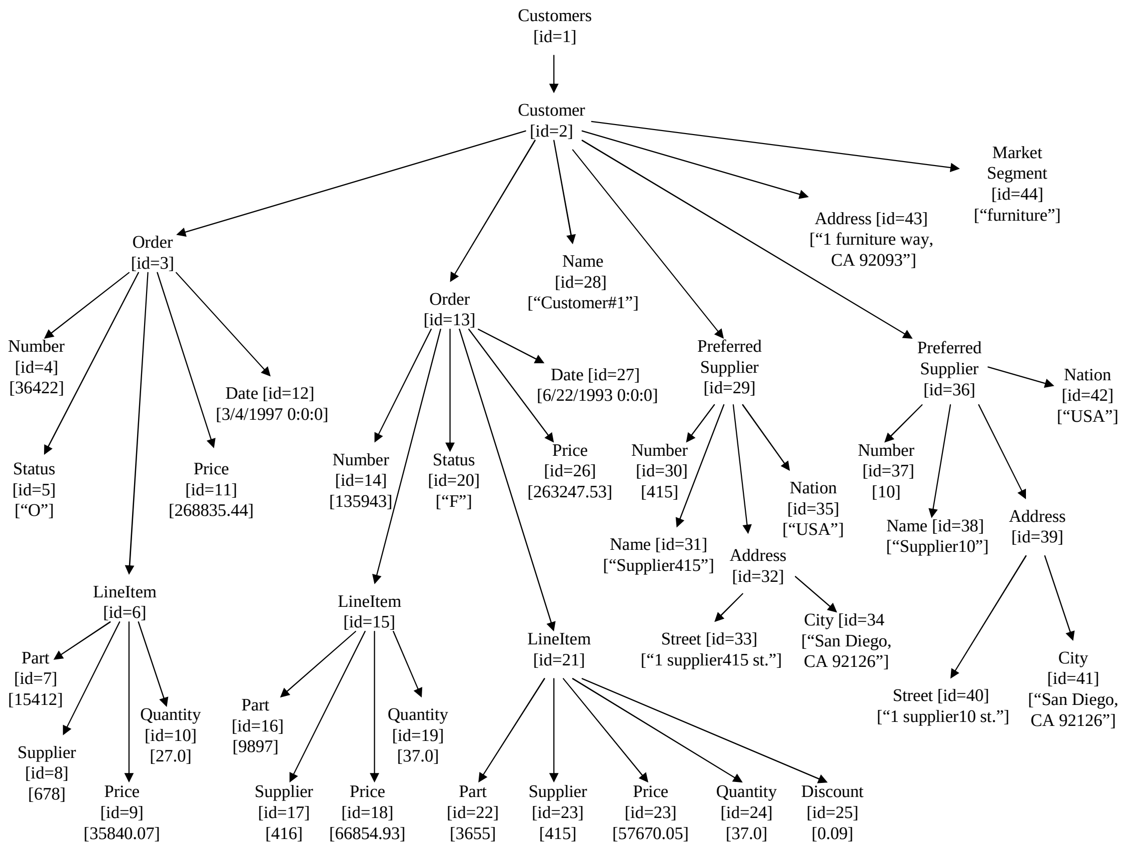


Figure II.1: A sample TPCH-like XML data set. Id’s and data values appear in brackets.

XML query language such as XQuery. We have excluded many document oriented features of XML such as mixed content, comments and processing instructions.

Every node has a unique id invented by the system. The id's play an important role in the conversion of the tree to relational data, as well as in the reconstruction of the XML fragments from the relational query results.

Definition 1 (XML document) *An XML document is a tree where*

1. *Every node has a label l coming from the set of element tags L*
2. *Every node has a unique id*
3. *Every atomic node has an additional label v coming from the set of values V. Atomic nodes can only be leafs of the document tree.* [1]

Figure II.1 shows an example of an XML document tree. We will use this tree as our running example. We consider only unordered trees. We can extend our approach to ordered trees because the node id's are assigned by a depth first traversal of the XML documents, and can be used to order sibling nodes.

II.C.1 XML Schema

We use *schema graphs* to abstract the syntax of XML Schema Definitions [91]. The following example illustrates the connection between XML Schemas and schema graphs.

Example 6 *Consider the XML Schema of Figure II.2 and the corresponding schema graph of Figure II.3. They both correspond to the TPC-H [85] data of Figure II.1. The schema indicates that the XML data set has a root element named* Customers*, which contains one or more* Customer *elements. Each* Customer *contains (in some order) all of the atomic elements* Name, Address*, and* MarketSegment*, as well as zero or more complex elements* Order *and* PreferedSupplier*. These complex elements in turn contain other sets of elements.*

[1] *However, not every leaf has to be an atomic node. Leafs can also be empty elements.*

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema">
    <xsd:element name = "customers">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref = "customer" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "customer">
        <xsd:complexType>
            <xsd:all>
                <xsd:element ref = "number"/>
                <xsd:element ref = "name"/>
                <xsd:element ref = "address"/>
                <xsd:element ref = "market"/>
                <xsd:element ref = "orders" minOccurs = "0" maxOccurs = "unbounded"/>
                <xsd:element ref = "preferred_supplier" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:all>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "number" type = "xsd:integer"/>
    <xsd:element name = "name" type = "xsd:string"/>
    <xsd:element name = "address" type = "xsd:string"/>
    <xsd:element name = "market" type = "xsd:string"
    <xsd:element name = "orders">
        <xsd:complexType>
            <xsd:all>
                <xsd:element ref = "number"/>
                <xsd:element ref = "status"/>
                <xsd:element ref = "price"/>
                <xsd:element ref = "date"/>
                <xsd:element ref = "lineitem" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:all>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "preferred_supplier">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref = "number"/>
                <xsd:element ref = "name"/>
                <xsd:element ref = "address"/>
                <xsd:element ref = "nation"/>
                <xsd:element ref = "balance"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "status" type = "xsd:string"/>
    <xsd:element name = "price" type = "xsd:float"/>
    <xsd:element name = "date " type = "xsd:string"/>
    <xsd:element name = "lineitem">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref = "part"/>
                <xsd:element ref = "supplier"/>
                <xsd:element ref = "quantity"/>
                <xsd:element ref = "price"/>
                <xsd:element ref = "discount" minOccurs = "0"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "part" type = "xsd:integer"/>
    <xsd:element name = "supplier" type = "xsd:integer"/>
    <xsd:element name = "quantity" type = "xsd:float"/>
    <xsd:element name = "discount" type = "xsd:float"/>
    <xsd:element name = "nation" type = "xsd:string"/>
    <xsd:element name = "balance" type = "xsd:float"/>
</xsd:schema>
```

Figure II.2: The TPCH XML Schema

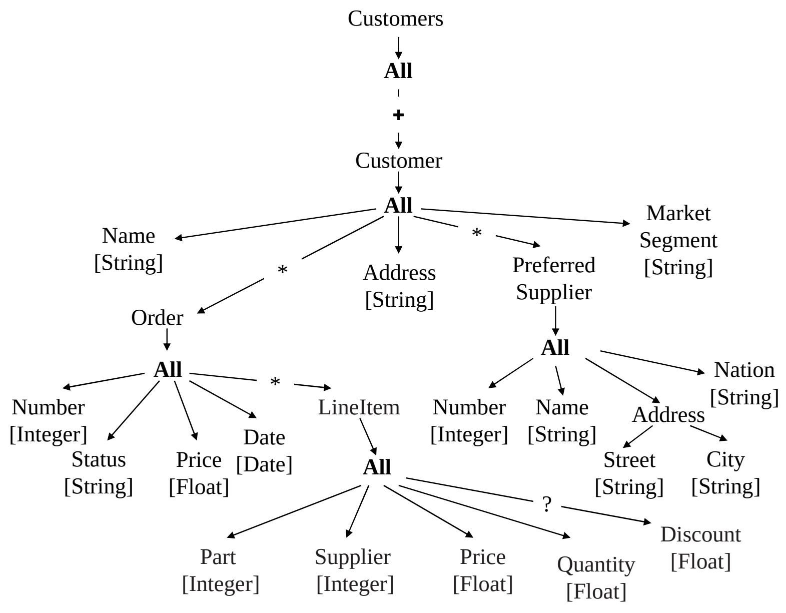


Figure II.3: Schema Graph notation

Notice that XML schemas and schema graphs are in some respect more powerful than DTDs [89]. For example, in the schema graph of Figure II.3 both *Customer* and *Supplier* have *Address* subelements, but the customer's address is simply a string, while the supplier's address consists of *Street* and *City* elements. DTD's cannot contain elements with the same name, but different content types.

Definition 2 (Schema Graph) *A schema is a directed graph G where:*

1. *Every node has a label l that is one of "all", or "choice", or is coming from the set of element tags L. Nodes labeled "all" and "choice" have at least two children.*

2. *Every leaf node has a label t coming from the set of types T.*

3. *Every edge is annotated with "minOccurs" and "maxOccurs" labels, which can be a non-negative integer or "unbounded".*

4. *A single node r is identified as the "root". Every node of G is reachable from r.*

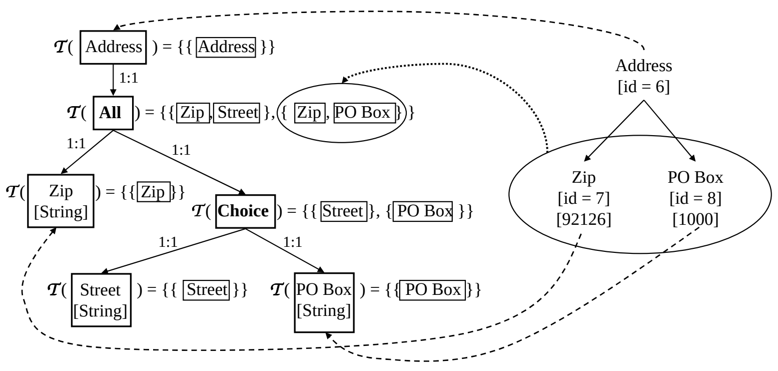

T(Address) = {{ Address }}
1:1
T(All) = {{ Zip , Street }, { Zip , PO Box }}
1:1
1:1
T(Zip [String]) = {{ Zip }}
T(Choice) = {{ Street }, { PO Box }}
1:1
1:1
T(Street [String]) = {{ Street }}
T(PO Box [String]) = {{ PO Box }}
Address [id = 6]
Zip [id = 7] [92126]
PO Box [id = 8] [1000]

Figure II.4: Content Types and Document Tree Validation

Schema graph nodes labeled with element tags are called *tag nodes*; the rest of the nodes are called *link nodes*.

Since we use an unordered data model, we do not include “sequence” nodes in the schema graphs. Their treatment is identical to that of “all” nodes. We also modify the usual definition of a *valid* document to account for the unordered model. To do that, we, first, define the *content type* of a schema node, which defines bags of sibling XML elements that are valid with respect to the schema node.

Definition 3 (Content Type) *Every node g of a schema graph G is assigned a* content type $\mathcal{T}(g)$, *which is set of bags of schema nodes, defined by the following recursive rules.*

- *If g is a tag node, $\mathcal{T}(g) = \{\{g\}\}$*
- *If g is a “choice” node $g = choice(g_1, \ldots, g_n)$, with min/maxOccur labels of the $g \to g_i$ edge denoted min_i and max_i, then $\mathcal{T}(g) = \bigcup_{i=1}^{n} \mathcal{T}_{min_i}^{max_i}(g_i)$, where $\mathcal{T}_{min_i}^{max_i}(g_i)$ is a union of all bags obtained by concatenation of k, not necessarily distinct, bags from $\mathcal{T}(g_i)$, where $min_i \le k \le max_i$, or $min_i \le k$ if $max_i =$ “unbounded”. If $min_i = 0$, $\mathcal{T}_{min_i}^{max_i}(g_i)$ also includes an empty bag.*
- *If g is an “all” node $g = all(g_1, \ldots, g_n)$, then $\mathcal{T}(g)$ is a union of all bags obtained by concatenation of n bags – one from each $\mathcal{T}_{min_i}^{max_i}(g_i)$.*

Definition 4 (Document Tree Valid wrt Schema Graph) *We say that a document tree T is valid with respect to schema graph G, if there exist a total mapping $\mathcal{M}$ of nodes of T to the tag nodes of G, such that $root(T)$ maps to $root(G)$, and for every pair $(t, g) \in \mathcal{M}$, the following holds:*

1. *$label(t) = label(g)$*
2. *A bag of schema nodes to which the children of t map is a member of $\mathcal{T}_{min}^{max}(g_c)$, where g_c is the child of g, and min and max are min/maxOccur labels of the $g \rightarrow g_c$ edge.*

Figure II.4 illustrates how the content types are assigned and used in the document validation. The *Address* element on the right is valid with respect to the schema graph on the left. Each schema node is annotated with its content type. For example, the type of the "choice" node is {{*Street*},{*PO Box*}}. The document validation is done by mapping XML tree nodes to the tag nodes of the schema graph (mappings are shown by the dashed lines), in such a way that the bag of types corresponding to the children of every XML node is a member of the content type of the child of the corresponding schema node. For example, the children of the *Address* element belong to the content type of the "all" node.

Normalized Schema Graphs To simplify the presentation we only consider *normalized* schema graphs, where all incoming edges of *link* nodes have $maxOccurs = 1$. Any schema graph can be converted into, a possibly less restrictive, normalized schema graph by a top-down breadth-first traversal of the schema graph that applies the following rules. For every link node N that has an incoming edge with $minOccurs = inMin$ and $maxOccurs = inMax$, where $inMax > 1$, the $maxOccurs$ is set to 1 and the $maxOccurs$ of every outgoing edge of N is multiplied by $inMax$. The result of the product is "unbounded" if at least one parameter is "unbounded". Similarly, if $inMin > 1$, the $minOccurs$ is set to 1 and the $minOccurs$ of every outgoing edge of N is multiplied by $inMin$. Also, if N is a

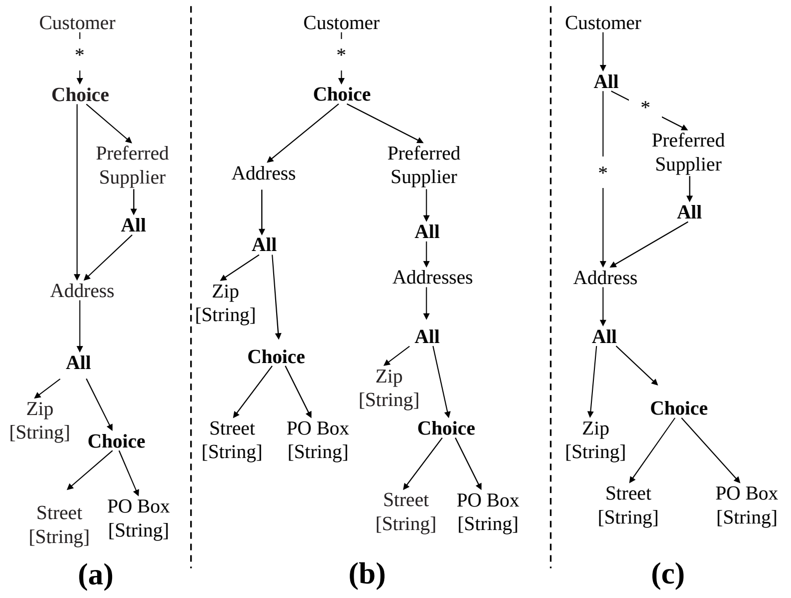


Figure II.5: Schema graphs (a) and (b) are equivalent. Graph (c) is normalization of graph (a).

"choice", it gets replaced with an "all" node with the same set of children, and for every outgoing edge the *minOccur* is set to 0. For example, the schema graph of Figure II.5(a) will be normalized into the graph of Figure II.5(c). Notice that the topmost "choice" node is replaced by "all", since a customer may contain multiple addresses and preferred supplier records.

Without loss of generality to the decomposition algorithms described next, we only consider schemas where $minOccurs \in \{0, 1\}$ and $manOccurs$ is either 1 or *unbounded*. We use the following symbols: "1", "*", "?", "+", to encode the "minOccurs"/"maxOccurs" pairs. For brevity, we omit "1" annotations in the figures. We also omit "all" nodes if their incoming edges are labeled "1", whenever this doesn't cause an ambiguity.

We only consider acyclic schema graphs. Schema graph nodes that are pointed by a "*" or a "+" will be called *repeatable*.

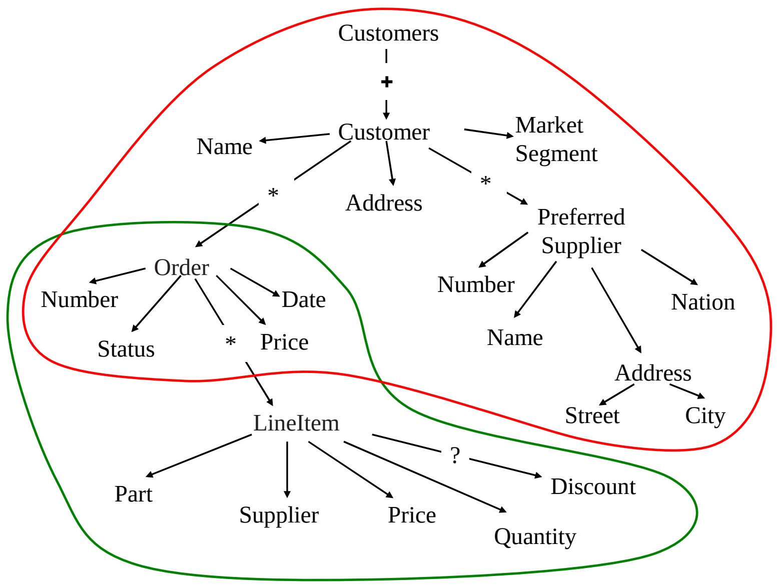


Figure II.6: An XML Schema decomposition

II.D XML Decompositions

We describe next the steps of decomposing an XML document into a relational database. First, we produce a schema decomposition, i.e., we use the schema graph to create a relational schema. Second, we decompose the XML data and load it into the corresponding tables. We use the schema decomposition to guide the data load.

The generation of an equivalent relational schema proceeds in two steps. First, we decompose the schema graph into fragments. Second, we generate a relational table definition for each fragment.

Definition 5 (Schema Decomposition) *A schema decomposition of a schema graph G is a set of fragments $F_1, \ldots, F_n$, where each fragment is a subset of nodes of G that form a connected DAG. Every tag node of G has to be member of at least one fragment.*

Due to acyclicity of the schema graphs, each fragment has at least one *fragment root* node, i.e., a node that does not have incoming edges from any other node of the fragment. Similarly, *fragment leaf* nodes are the nodes that do not have outgoing edges that lead to other nodes of the fragment. Note that a schema decomposition is not necessarily a partition of the schema graph – a node may be included in multiple fragments (Figure II.6).

Some fragments may contain only "choice" and "all" nodes. We call these fragments *trivial*, since they correspond to empty data fragments. We only consider decompositions which contain connected, *non-trivial* fragments, where all fragment leafs are tag nodes.

DAG schemas offer an extra degree of freedom, since an equivalent schema can be obtained by "splitting" some of the nodes that have more than one ancestor. For example, the schema of Figure II.5(b), can be obtained from the schema of Figure II.5(a) by splitting at element *Address*. Such a split corresponds to a *derived horizontal partitioning* of a relational schema [67].

Similarly, element nodes may also be eliminated by "combining" nodes. For example, an $all(a*, b, a*)$ may be reduced to $all(a*, b)$ if types of both a's are equal [2]. Since we consider an unordered data model, the queries cannot distinguish between "first" and "second" a's in the data. Thus, we do not need to differentiate between them. A similar DTD reduction process was used in [80]. However, unlike [80] our decompositions do not require reduction and offer flexibility needed to support the document order. Similar functionality is included in LegoDB [17].

Definition 6 (Path Set, Equivalent SchemaGraphs) *A path set of a schema graph G, denoted $PS(G)$, is the set of all possible paths in G that originate at the root of G. Two schema graphs G_1 and G_2 are* equivalent *if $PS(G_1) = PS(G_2)$.*

We define the set of *generalized schema decompositions* of a graph G to be the set of schema decompositions of all graphs G' that are equivalent to G (including the schema decompositions of G itself.) Whenever it is obvious from the context we will say "set of schema decompositions" implying the set of generalized

[2]We say that types A and B are equal, if every element that is valid wrt A is also valid wrt B, and vice versa.

schema decompositions.

Definition 7 (Root Fragments, Parent Fragments) *A* root fragment *is a fragment that contains the root of the schema graph. For each non-root fragment F we define its* Parent Fragments *in the following way: Let R be a root node of F, and let P be a parent of R in the schema graph. Any fragment that contains P is called a parent fragment of F.* [3]

Definition 8 (Fragment Table) *A* Fragment Table *T corresponds to every fragment F. T has an attribute $A_{N_{ID}}$ of the special "ID" datatype*[4] *for every tag node N of the fragment. If N is an atomic node the schema tree T also has an attribute A_N of the same datatype as N. If F is not a root fragment, T also includes a parent reference column, of type ID, for each distinct path that leads to a root of F from a repeatable ancestor A and does not include any intermediate repeatable ancestors. The parent reference columns store the value of the ID attribute of A.*

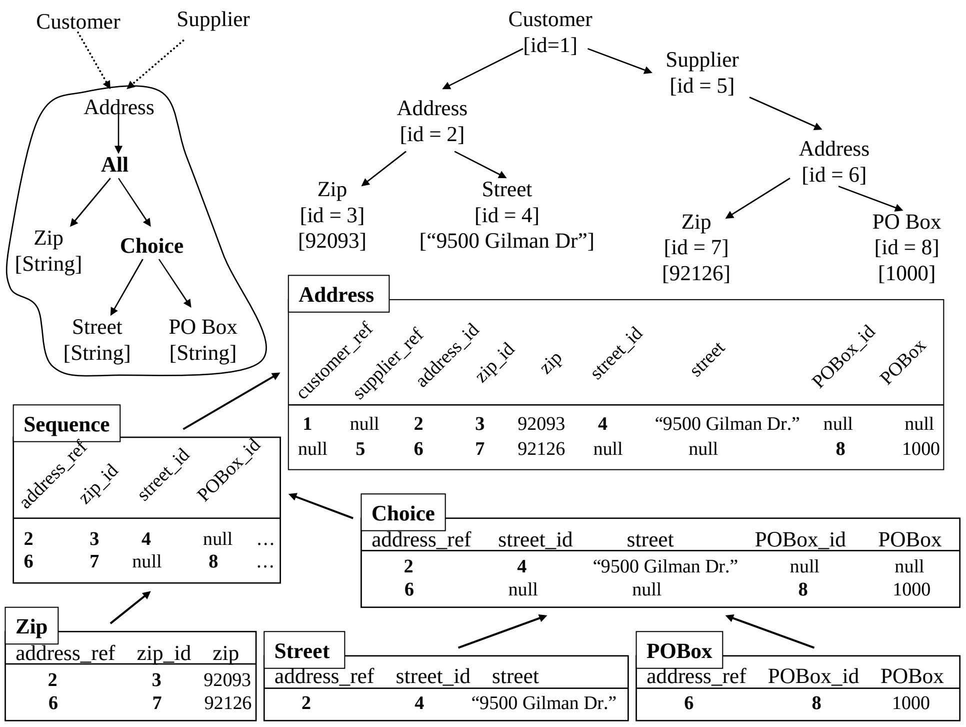
For example, consider the `Address` fragment table of Figure II.7. Regardless of other fragments present in the decomposition, the `Address` table will have two parent reference columns. One column will refer to the *Customer* element and another to the *Supplier*. Since we consider only tree data, every tuple of the `Address` table will have exactly one non-null parent reference.

A fragment table is named after the left-most root of the corresponding fragment. Since multiple schema nodes can have the same name, name collisions are resolved by appending a unique integer.

We use null values in ID columns to represent missing optional elements. For example, the null value in the `POBox_id` of the first tuple of the `Address` table indicates that the *Address* element with id=2 does not have a *POBox* subelement. An empty XML element N is denoted by a non-null value in $A_{N_{ID}}$ and a null in A_N.

[3] *Note that a decomposition can have multiple root fragments, and a fragment can have multiple parent fragments.*

[4] *In RDBMS's we use the "integer" type to represent the "ID" datatype.*



Address

customer_ref	supplier_ref	address_id	zip_id	zip	street_id	street	POBox_id	POBox
1	null	**2**	**3**	92093	**4**	“9500 Gilman Dr.”	null	null
null	**5**	**6**	**7**	92126	null	null	**8**	1000

Sequence

address_ref	zip_id	street_id	POBox_id	
2	**3**	**4**	null	…
6	**7**	null	**8**	…

Choice

address_ref	street_id	street	POBox_id	POBox
2	**4**	“9500 Gilman Dr.”	null	null
6	null	null	**8**	1000

Zip

address_ref	zip_id	zip
2	**3**	92093
6	**7**	92126

Street

address_ref	street_id	street
2	**4**	“9500 Gilman Dr.”

POBox

address_ref	POBox_id	POBox
6	**8**	1000

Figure II.7: Loading data into fragment tables

Data Load We use the following inductive definition of fragment tables' content. First, we define the data content of a fragment consisting of a single tag node N. The fragment table T_N, called *node table*, contains an ID attribute $A_{N_{ID}}$, a value attribute A_N, and one or more parent attributes. Let us consider a *Typed Document Tree* D', where each node of D is mapped to a node of the schema graph. A tuple is stored in T_N for each node $d \in D$, such that $(d \rightarrow N) \in D'$. Assume that d is a child of the node $p \in D$, such that $(p \rightarrow P) \in D'$. The table T_N will be populated with the following tuple: $\langle A_{P_{ID}} = p_{id}, A_{N_{ID}} = d_{id}, A_N = d \rangle$. If T_N contains parent attributes other than $A_{P_{ID}}$, they are set to null.

A table T corresponding to an internal node N is populated depending on the type of the node.

- If N is an “all”, then T is the result of a join of all children tables on parent reference attributes.

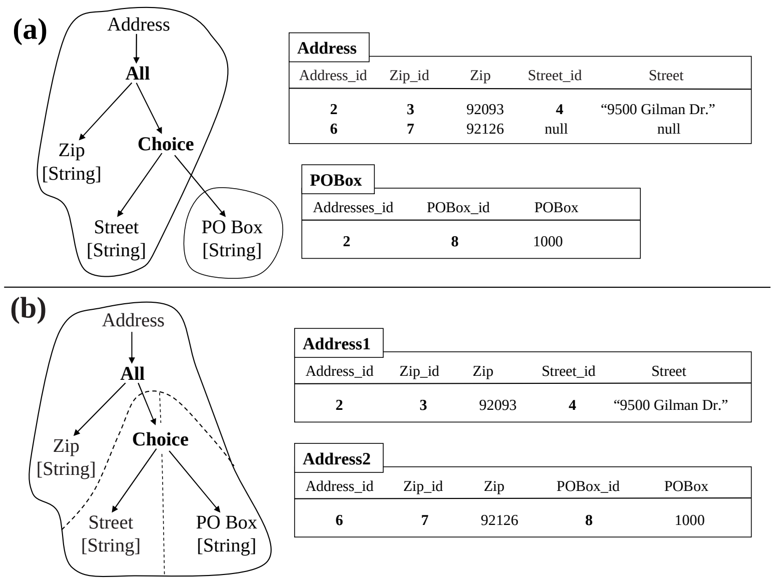
- If N is a "choice", then T is the result of an outer union [5] of all children tables.
- If N is a tag node, which by definition has exactly one child node with a corresponding table T_C, then $T = T_N ⟕ T_C$

The following example illustrates the above definition. Notice that the XCacheDB Loader does not use the brute force implementation suggested in the example. We employ optimizations that eliminate the majority of the joins.

Example 7 *Consider the schema graph fragment, and the corresponding data fragment of Figure II.7. The* `Address` *fragment table is built from node tables* `Zip`*,* `Street`*, and* `POBox`*, according to the algorithm described above. A table corresponding to the "choice" node in the schema graph is built by taking an outer union of* `Street` *and* `POBox`*. The result is joined with* `Zip` *to obtain the table corresponding to the "all" node. The result of the join is, in turn, joined with the* `Address` *node table (not shown) which contains three attributes "customer_ref", "supplier_ref", and "address_id".*

Alternatively, the "Address" fragment of Figure II.7 can be split in two as shown in Figure II.8(a) and (b). The dashed lines in Figure II.8(b) indicates that a *horizontal partitioning* of the fragment should occur along the "choice" node. This line indicates that the fragment table should be split into two. Each table projects out attributes corresponding to one side of the "choice". The tuples of the original table are partitioned into the two tables based on the null values of the projected attributes. This operation is similar to the "union distribution" discussed in [17]. Horizontal partitioning improves the performance of queries that access either side of the union (e.g., either *Street* or *POBox* elements). However, performance may degrade for queries that access only *Zip* elements. Since we assume no knowledge

[5] Outer union of two tables P and Q is a table T, with a set of attributes $attr(T) = attr(P) \cup attr(Q)$. The table T contains all tuples of P and Q extended with nulls in all the attributes that were not present in the original.



Address

Address_id	Zip_id	Zip	Street_id	Street
2	**3**	92093	**4**	"9500 Gilman Dr."
6	**7**	92126	null	null

POBox

Addresses_id	POBox_id	POBox
2	**8**	1000

Address1

Address_id	Zip_id	Zip	Street_id	Street
2	**3**	92093	**4**	"9500 Gilman Dr."

Address2

Address_id	Zip_id	Zip	POBox_id	POBox
6	**7**	92126	**8**	1000

Figure II.8: Alternative fragmentations of data of Figure II.7

of the query workload, we do not perform horizontal partitioning automatically, but leave it as an option to the system administrator.

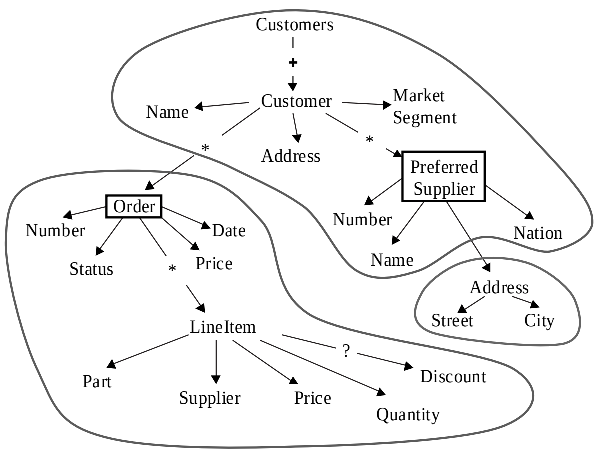
The following example illustrates decomposing TPCH-like XML schema of Figure II.3 and loading it with data of Figure II.1.

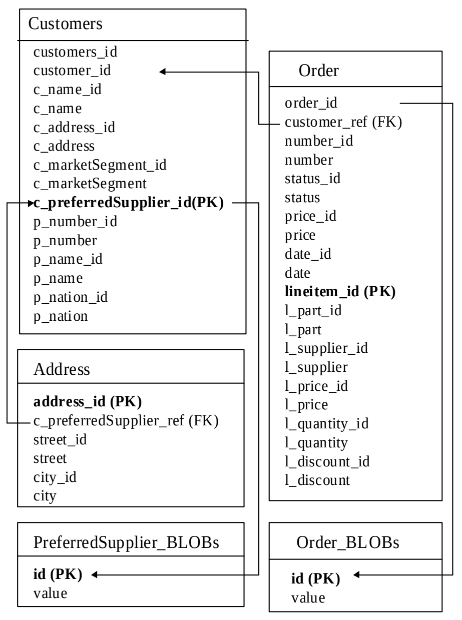
Example 8 *Consider the schema decomposition of Figure II.9. The decomposition consists of three fragments rooted at the elements* Customers*,* Order*, and* Address*. Hence the corresponding relational schema has tables* `Customers`*,* `Order`*, and* `Address`*. The bottom part of Figure II.9 illustrates the contents of each table for the dataset of Figure II.1. Notice that the tables* `Customers` *and* `Order` *are not in BCNF.*

For example, the table `Order` *has the non-key functional dependency "order_id → number_id", which introduces redundancy.*

We use "(FK)" labels in Figure II.9 to indicate parent references. Technically these references are not foreign keys since they do not necessarily refer to a primary key.

Alternatively one could have decomposed the example schema as shown





PreferedSupplier_BLOBs

id	value
29	PreferredSupplier(29)[Number(30)[415],...
36	PreferredSupplier(36)[Number(37)[10],...

Order_BLOBs

id	value
3	Order(3)[Number(4)[36422],Status...
13	Order(13)[Number(14)[135943],...

Customers

customers_id	customer_id	c_name_id	c_name	c_address_id	c_address	c_market Segment_id	c_market Segment	c_preferred Supplier_id	p_number_id	p_number	p_name_id	p_name	p_nation_id	p_nation
1	**2**	28	"Customer1"	43	"1 Furniture..."	44	"furniture"	**29**	30	415	31	"Supplier415"	35	"USA"
1	**2**	28	"Customer1"	43	"1 Furniture..."	44	"furniture"	**36**	37	10	38	"Supplier10"	42	"USA"

Order

order_id	customer_ref	number_id	number	status_id	status	price_id	price	date_id	date	lineitem_id	l_part_id	l_part	l_supplier_id	l_supplier	l_price_id	l_price	l_quantity_id	l_quantity	l_discount_id	l_discount
3	**2**	4	36422	5	"O"	11	268835.44	12	3/4/1997 0:0:0	**6**	7	15412	8	678	9	35840.07	10	27.0	Null	Null
13	**2**	14	135943	20	"F"	26	263224.53	27	6/22/1993 0:0:0	**15**	16	9897	17	416	18	66854.93	19	37.0	Null	Null
13	**2**	14	135943	20	"F"	26	263224.53	27	6/22/1993 0:0:0	**21**	22	3655	23	415	23	57670.05	24	37.0	25	0.09

Address

address_id	c_preferredSupplier_ref	street_id	street	city_id	city
32	**29**	33	"1 supplier10 St."	34	"San Diego, CA 92126"
39	**36**	40	"1 supplier415 St."	41	"San Diego, CA 92126"

Figure II.9: XML Schema and Data decomposition

in Figure II.6. In this case there is a non-FD multi-valued dependency (MVD) in the `Customers` *table, i.e., an MVD that is not implied by a functional dependency. Orders and preferred suppliers of every customer are independent of each other:*

$$\begin{aligned}
&customers\_id, customer\_id, c\_name\_id, c\_address\_id,\\
&c\_marketSegment\_id, c\_name, c\_address,\\
&c\_marketSegment \quad \twoheadrightarrow \quad c\_preferredSupplier\_id,\\
&p\_name\_id, p\_number\_id, p\_nation\_id, p\_name,\\
&p\_number, p\_nation, p\_address\_id, a\_street\_id,\\
&a\_city\_id, a\_street, a\_city
\end{aligned}$$

The decompositions that contain non-FD MVD's are called *MVD decompositions.*

Vertical Partitioning In the schema of Figure II.9 the *Address* element is not repeatable, which means that there is at most one address per supplier. Using a separate `Address` table is an example of *vertical partitioning* because there is a one-to-one relationship between the `Address` table and its parent table `Customers`. The vertical partitioning of XML data was studied in [17], which suggests that partitioning can improve performance if the query workload is known in advance. Knowing the groups of attributes that get accessed together, the vertical partitioning can be used to reduce table width without incurring a big penalty from the extra joins. We do not consider vertical partitioning, but the results of [17] can be carried over to our approach. We use the term *minimal* to refer to decompositions without vertical partitioning.

Definition 9 (Minimal Decompositions) *A decomposition is* minimal *if all edges connecting nodes of different fragments are labeled with "*" or "+".*

Figure II.6 and Figure II.10 show two different minimal decompositions of the same schema. We call the decomposition of Figure II.10 a *4NF decomposition*

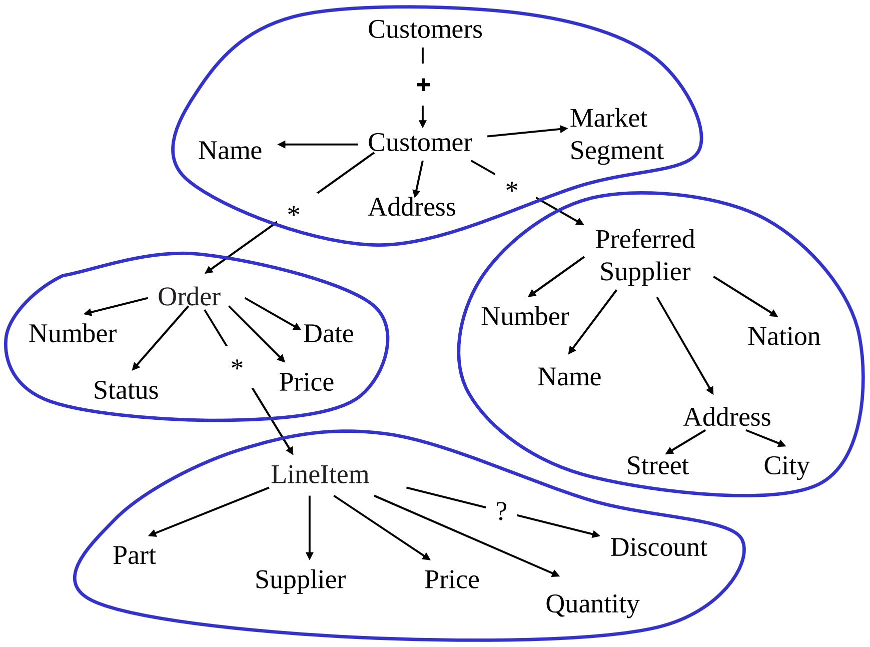


Figure II.10: Minimal 4NF XML Schema decomposition

because all its fragments are *4NF fragments* (i.e. the fragment tables are in 4NF). Note that a fragment is 4NF if and only if it does not include any "*" or "+" labeled edges, i.e. no two nodes of the fragment are connected by a "*" or "+" labeled edge. We assume that the only dependencies present are those derived by the decomposition.

Every XML Schema tree has exactly one minimal 4NF decomposition, which minimizes the space requirements. From here on, we only consider minimal decompositions.

Prior work [80, 17] considers only 4NF decompositions. However we employ denormalized decompositions to improve query execution time as well as response time. Particularly important for performance purposes is the class of inlined decompositions described below. The inlined decompositions improve query performance by reducing the number of joins, and (unlike MVD decompositions) the space overhead that they introduce depends only on the schema and not on the dataset.

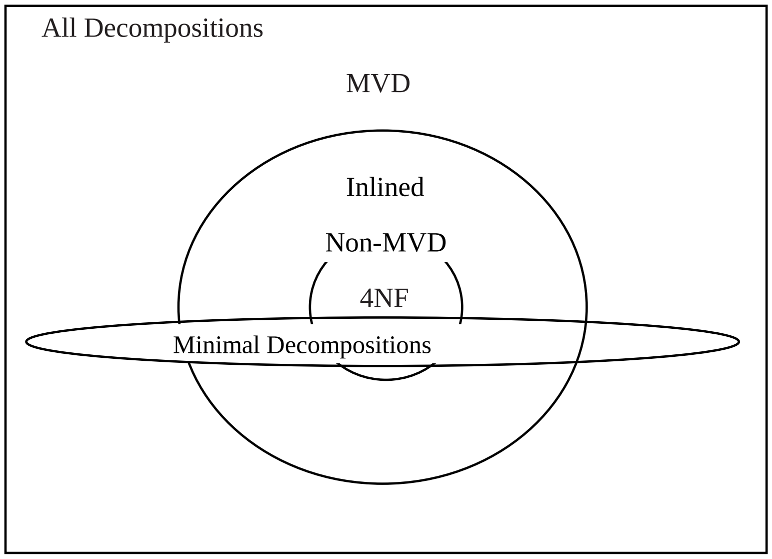


Figure II.11: Classification of Schema decompositions

Definition 10 (Non-MVD Decompositions and Inlined Decompositions) *A non-MVD fragment is one where all "*" and "+" labeled edges appear in a single path. A non-MVD decomposition is one that has only non-MVD fragments. An* inlined *fragment is a non-MVD fragment that is not a 4NF fragment. An inlined decomposition is a non-MVD decompositions that is not a 4NF decomposition.*

The non-MVD fragment tables may have functional dependencies (FD's) that violate the BCNF condition (and also the 3NF condition [36]), but they have no non-FD MVD's. For example, the *Customers* table of Figure II.9 contains the FD

$$customer\_ID \rightarrow c\_name$$

that breaks the BCNF condition, since the key is "c_preferredSupplier_id". However, the table has no non-FD MVD's.

From the point of view of the relational data, an *inlined fragment* table is the join of fragment tables that correspond to a line of two or more 4NF fragments. For example, the fragment table *Customers* of Figure II.9 is the join of the fragment tables that correspond to the 4NF fragments *Customers* and *PreferredSupplier* of

Figure II.10. The tables that correspond to inlined fragments are very useful because they reduce the number of joins while they keep the number of tuples in the fragment tables low.

Lemma 1 (Space Overhead as a Function of Schema Size) *Consider two non-MVD fragments F_1 and F_2 such that when unioned, they result in an inlined fragment F.* [6] *For every XML data set, the number of tuples of F is less than the total number of tuples of F_1 and F_2.*

Proof: Let's consider the following three cases. First, if the schema tree edge that connects F_1 and F_2 is labeled with "1" or "?", the tuples of F_2 will be inlined with F_1. Thus F will have the same number of tuples as F_1.
Second, if the edge is labeled with "+", F will have the same number of tuples as F_2, since F will be the result of the join of F_1 and F_2, and the schema implies that for every tuple in F_2, there is exactly one matching tuple, but no more in F_1.
Third, if the edge is labeled with "*", F will have fewer tuples than the total of F_1 and F_2, since F will be the result of the left outer join of F_1 and F_2. $\square$

We found that the inlined decompositions can provide significant query performance improvement. Noticeably, the storage space overhead of such decompositions is limited, even if the decomposition include all possible non-MVD fragments.

Definition 11 (Complete Non-MVD Decompositions) *A complete non-MVD decomposition,* complete *for short, is one that contains all possible non-MVD fragments.*

The complete non-MVD decompositions are only intended for the illustrative purpose, and we are not advocating their practical use.

Note that a complete non-MVD decomposition includes all fragments of the 4NF decomposition. The other fragments of the complete decomposition

[6] *A fragment consisting of two non-MVD fragments connected together, is not guaranteed to be non-MVD .*

consist of fragments of the 4NF decomposition connected together. In fact, a 4NF decomposition can be viewed as a tree of 4NF fragments, called *4NF fragment tree.* The fragments of a complete minimal non-MVD decomposition correspond to the set of paths in this tree. The space overhead of a complete decompositions is a function of the size of the 4NF fragment tree.

Lemma 2 (Space Overhead of a Complete Decomposition) *Consider a schema graph G, its complete decomposition $D_C(G) = \{F_1, \dots, F_k\}$, and a 4NF decomposition $D_{4NF}(G)$. For every XML data set, the number of tuples of the complete decomposition is*

$$|D_C(G)| = \sum_{i=1}^{k} |F_i| < |D_{4NF}(G)| * h * n$$

where h is the height of the 4NF fragment tree of G, and n is the number of fragments in $D_{4NF}(G)$.

Proof: Consider a *record tree* R constructed from an XML document tree T in the following fashion. A node of the record tree is created for every tuple of the 4NF data decomposition $D_{4NF}(T)$. Edges of the record tree denote child-parent relationships between tuples. There is a one to one mapping from paths in the record tree to paths in its 4NF fragment tree, and the height of the record tree h equals to the height of the 4NF fragment tree. Since any fragment of $D_C(G)$ maps to a path in the 4NF fragment tree, every tuple of $D_C(T)$ maps to a path in the record tree. The number of path's in the record tree $P(R)$ can be computed by the following recursive expression: $P(R) = N(R) + P(R_1) + \cdots + P(R_n)$, where $N(R)$ is the number of nodes in the record tree and stands for all the paths that start at the root. R_i's denote subtrees rooted at the children of the root. The maximum depth of the recursion is h. At each level of the recursion, after the first one, the total number of added paths is less than N. Thus $P(R) < hN$.

Multiple tuples of $D_C(T)$ may map to the same path in the record tree, because each tuple of $D_C(T)$ is a result of some outerjoin of tuples of $D_{4NF}(T)$, and the same tuple may be a result of multiple outer joins (e.g. $A ⟕ B = A ⟕ B ⟕ C$, if C is empty.) However the same tuple cannot be a result of more than n distinct left outerjoins. Thus $|D_C(G)| \leq P(R) * n$. By definition $|D_{4NF}(G)| = N$; hence $|D_C(G)| < |D_{4NF}(G)| * h * n$. $\square$

II.D.1 BLOBs

To speed up construction of the XML results from the relational result-sets XCacheDB stores a binary image of pre-parsed XML subtrees as Binary Large OBjects (BLOBs). The binary format is optimized for efficient navigation and printing of the XML fragments. The fragments are stored in special BLOBs tables that use node IDs as foreign keys to associate the XML fragments to the appropriate data elements.

By default, every subtree of the document except the trivial ones (the entire document and separate leaf elements) is stored in the Blobs table. This approach may have unnecessarily high space overhead because the data gets replicated up to $H - 2$ times, where H is the depth of the schema tree. We reduce the overhead by providing a graphical utility, the *XCacheDB Loader*, which allows the user to control which schema nodes get "BLOB-ed", by annotating the XML Schema. The user should BLOB only those elements that are likely to be returned by the queries. For example, in the decomposition of Figure II.9 only `Order` and `PreferredSupplier` elements were chosen to be BLOB-ed, as indicated by the boxes. `Customer` elements may be too large and too infrequently requested by a query, while `LineItem` is small and can be constructed quickly and efficiently without BLOB's.

We chose not to store Blobs in the same tables as data to avoid unnecessary increase in table size, since Blob structures can be fairly large. In fact, a Blob has similar size to the XML subtree that it encodes. The size of an XML document (without the header and whitespace) can be computed as

$$XML_{Size} = E_N * (2E_{Size} + 5) + T_N * T_{Size}$$

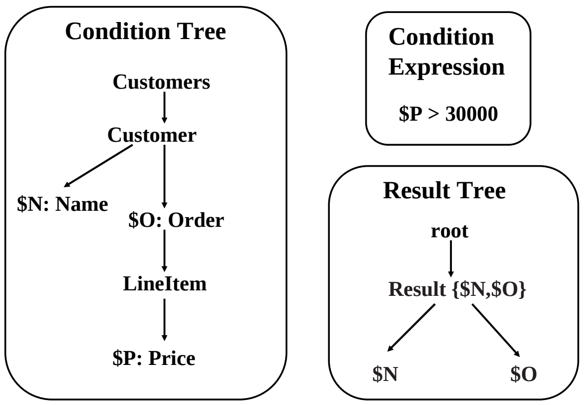


Figure II.12: XML Query notation

where E_N is the number of elements, E_{Size} is the average size of the element tag, T_N is how many elements contain text (i.e. leafs) and T_{Size} is the average text size. The size of a BLOB is:

$$BLOB_{Size} = E_N * (E_{Size} + 10) + T_N * (T_{Size} + 3)$$

The separate Blobs table also gives us an option of using a separate SQL query to retrieve Blobs which improves the query response time.

II.E XML Query Processing

We represent XML queries with a tree notation similar to *loto-ql* [69]. The query notation facilitates explanation of query processing and corresponds to FOR-WHERE-RETURN queries of the XQuery standard [101].

Definition 12 (Query) *A query is a tuple $\langle C, E, R\rangle$, where C is called* condition tree*, E is called* condition expression*, and R is called* result tree*.*

C is a labeled tree that consists of:

- *Element nodes that are labeled with labels from L. Each element node n may also be labeled with a variable $Var(n)$.*

- *Union nodes. The same set of variables must occur in all children subtrees of a Union node. Two nodes cannot be labeled with the same variable, unless their lowest common ancestor is a Union node.*

E *is a logical expression involving logical predicates, logical connectives, constants, and variables that occur in* C*.*

R *is a tree where internal nodes are labeled with constants and leaf nodes are labeled either with variables that occur in* C *or with constants. Some nodes may also have "group-by" labels consisting of one or more variables that occur in* C*. If a variable* V *labels a leaf* $l \in R$ *then* V *is in the group-by label of* l *or the group-by label of an ancestor of* l*.*

The query semantics are based on first matching the condition tree with the XML data to obtain bindings and then using the result tree to structure the bindings into the XML result.

The semantics of the condition tree are defined in two steps. First, we remove Union nodes and produce a forest of *conjunctive condition trees*, by traversing the condition tree bottom-up and replacing each Union node non-deterministically by one of its children. This process is similar to producing a disjunctive normal form of a logical expression. Set of bindings produced by the condition tree is defined as a union of sets of bindings produced by each of the conjunctive condition trees.

Formally, let C be a condition tree of a query and t be the XML document tree. conjunctive Let $Var(C)$ be the set of variables in C. Let $C_1 ... C_l$ be a set of all conjunctive condition trees of C. Note that $Var(C) = Var(C_i), \forall i \in [1, l]$. A *variable binding* $\hat{\beta}$ maps each variable of $Var(C)$ to a node of t. The set of variable bindings is computed based on the set of condition tree bindings. A condition tree binding β maps each node n of some conjunctive condition tree C_i to a node of t. The condition tree binding is valid if $\beta(root(C_i)) = root(t)$ and recursively, traversing C depth-first left-to-right, for each child c_j of a node $c \in C_i$, assuming

c is mapped to $x \in t$, there exists a child x_j of x such that $\beta(c_j)) = x_j$ and $label(c_j) = label(x_j)$.

The set of variable bindings consists of all bindings $\hat{\beta} = [V_1 \mapsto x_1, \ldots, V_n \mapsto x_n]$ such that there is a condition tree binding $\beta = [c_1 \mapsto x_1, \ldots, c_n \mapsto x_n, \ldots]$, such that $V_1 = Var(c_1), \ldots, V_n = Var(c_n)$.

The condition expression E is evaluated using the binding values and if it evaluates to true, the variable binding is qualified. Notice that the variables bind to XML elements and not to their content values. In order to evaluate the condition expression, all variables are coerced to the content values of the elements to which they bind. For example, in Figure II.12 the variable P binds to an XML element "price". However, when evaluating the condition expression we use the integer value of "price".

Once a set of qualified bindings is identified, the resulting XML document tree is constructed by structural recursion on the result tree R as follows. The recursion starts at the root of R with the full set of qualified bindings B. Traversing R top-down, for each sub-tree $R(n)$ rooted at node n, given a partial set of bindings B' (we explain how B' gets constructed next) we construct a forest $F(n, B')$ following one of the cases below:

Label: If n consists of a tag label L without a group-by label, the result is an XML tree with root labeled L. The list of children of the root is the concatenation $F(n_1, B'')\# \ldots \# F(n_m, B'')$, where $n_1, n_2, \ldots, n_m$ are the children of n. For each of the children, the partial set of bindings is $B'' = B'$.

Group-By: If n is of the form $L\{V_1, \ldots, V_k\}$, where $V_1, \ldots, V_k$ are group-by variables, $F(n, B')$ contains an XML tree $T_{v_1,\ldots,v_k}$ for each distinct set $v_1, \ldots, v_k$ of values of $V_1, \ldots, V_k$ in B'. Each $T_{v_1,\ldots,v_k}$ has its root labeled L. The list of children of the root is the concatenation $F(n_1, B'_1)\# \ldots \# F(n_m, B'_m)$, where $n_1, n_2, \ldots, n_m$ are the children of n. For $T_{v_1,\ldots,v_k}$ and n_i the partial set of bindings is

$B'_i = \Pi_{V(n_i)}(\sigma_{V_1=v_1 AND \ldots AND V_k=v_k} B')$, where $V(n_i)$ is the set of variables that occur in the tree rooted at n_i.

Leaf Group-By: If n is a leaf node of form $V\{V_1, \ldots, V_k\}$, the result is a list of values of V, for each distinct set $v_1, \ldots, v_k$ of values of $V_1, \ldots, V_k$ in B'.

Leaf Variable: If n is a single variable V, and V binds to an element E in B', the result is E. If the query plan is valid, B' will contain only a single tuple.

The result of the query is the forest $F(r, B)$, where r is the root of the result tree and B is the set of bindings delivered by the condition tree and condition expression. However, since in our work we want to enforce that the result is a single XML tree, we require that r does not have a "group-by" label.

Example 9 *The condition tree and expression of the query of Figure II.12 retrieve tuples $\langle N, O \rangle$ where N is the* Name *element of a* Customer *element with an* Order *O that has at least one* LineItem *that has* Price *greater than 30000. For each tuple $\langle N, O \rangle$ a* Result *element is produced that contains the N and the O. This is essentially query number 18 of the TPC-H benchmark suite [85], modified not to aggregate across lineitems of the order. It is equivalent to the XQuery of Figure II.13.*

For example, if the query is executed on data of Figure II.1, the following set of bindings is produced, assuming that the Order *elements are BLOB-ed.*

$\langle \$N/Name_{29}[\text{“}Customer1\text{”}],$
$\$O/Order_3, \$P/Price_9[35840.07]\rangle$
$\langle \$N/Name_{29}[\text{“}Customer1\text{”}],$
$\$O/Order_{13}, \$P/Price_{18}[66854.93]\rangle$
$\langle \$N/Name_{29}[\text{“}Customer1\text{”}],$
$\$O/Order_{13}, \$P/Price_{24}[57670.05]\rangle$

```
</root>
     FOR $C IN document(‘‘customers.xml")/
               Customers/Customer
         $N IN $c/Name
         $O IN $c/Order
     WHERE not empty(
         FOR $P IN $o/LineItem/Price
         WHERE $P > 30000
         RETURN $P
     )
     RETURN
         <result>
             {$N}
             {$O}
         </result>
</root>
```

Figure II.13: The XQuery equivalent to the query of Figure II.12

Numbers in subscript indicate node ID's of the elements; square brackets denote values of atomic elements and subelements of complex elements. First, a single root *element is created. Then, the group-by on the* Result *node partitions the bindings into two groups (for $Order_3$ and $Order_{13}$), and creates a* Result *element for each group. The second group-by creates two* Order *elements from the following two sets of bindings.*

$\langle \$O/Order_3, \$P/Price_9[35840.07]\rangle$
and
$\langle \$O/Order_{13}, \$P/Price_{18}[66854.93]\rangle$
$\langle \$O/Order_{13}, \$P/Price_{24}[57670.05]\rangle$

The final result of the query is the following document tree:

```
root₁₀₀[
   Result₁₀₁[
      Name₂₉[‘‘Customer1"],
      Order₃[...],
   Result₁₀₂[
      Name₂₉[‘‘Customer1"],
      Order₁₃[...]
   ]
]
```

We can extend our query semantics to ordered XML model. To support order-preserving XML semantics, group-by operators will produce lists, given sorted lists of source bindings. In particular the group-by operator will order the output elements according to the node ID's of the bindings of the group-by variables. For example, the group-by in query of Figure II.12 will produces lists of pairs of names and orders, sorted by name ID and order ID.

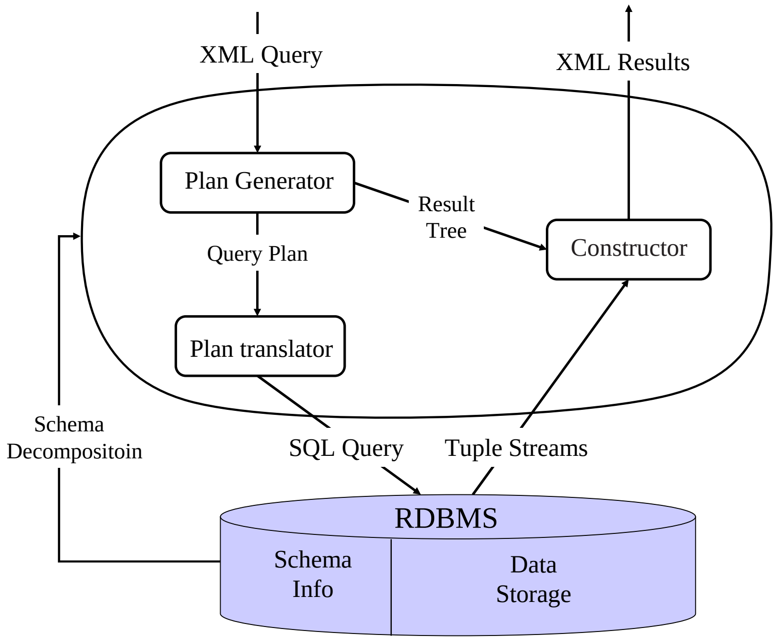


Figure II.14: Query Processing Architecture

II.E.1 Query Processing

Figure II.14 illustrates the typical query processing steps followed by XML databases built on relational databases; the architecture of XCacheDB is indeed based on the one of Figure II.14. The *plan generator* receives an XML query and a schema decomposition. It produces a plan, which consists of the *condition tree*, the *condition expression*, the *plan decomposition*, and the *result tree*. The *plan translator* turns the query plan into an SQL query. Plan result trees outline how the qualified data of fragments are composed into the XML result. The *constructor* receives the tuples in the SQL results and structures them into the XML result following the plan result tree. Formally a query plan is defined as follows.

Definition 13 ((Valid) Query Plan) *A query plan wrt a schema decomposition D, is a tuple $\langle C', P', E, R' \rangle$, where C' is a plan condition tree, P' is a* plan decomposition, *and R' is a* plan result tree.

C' has the structure of a query condition, except that some edges may be labeled

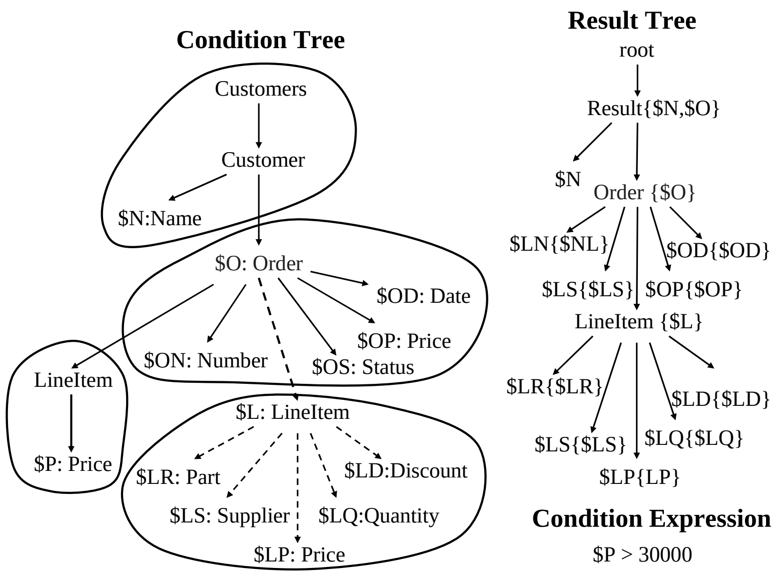


Figure II.15: Query Plan

as "soft". However, no path may contain a non-soft edge after a soft one. That is, all the edges below a soft edge have to be soft.

P' *is a pair* $\langle P, f\rangle$*, where* P *is a partition of* C' *into fragments* $P_1, \ldots, P_n$*, and* f *is a mapping from* P *into the fragments of* D*. Every* P_i *has to be covered by the fragment* $f(P_i)$ *in the sense that for every node in* P_i *there is a corresponding schema node in* $f(P_i)$*.*

R' *is a tree that has the same structure as a query result tree. All variables that appear in* R' *outside the group-by labels, must bind to atomic elements*[7] *or bind to elements that are BLOB-ed in* D*.*

C' and R' are constructed from C, R and the schema decomposition D by the following nondeterministic algorithm. For every variable V, that occurs in R on node N_R and in C on node N_C, find the schema node S that corresponds to N_C, i.e. the path from the root of C to N_C and the path from the schema root to S have the same sequence of node labels. If S exists and is not atomic, there are two options:

[7] *It is easy to verify this property using the schema graph.*

1. Do not perform any transformations. In this case V will bind to BLOBs assuming that S is BLOB-ed in D.

2. Extend N_C with all the children of S. Label every new edge as "soft" if the corresponding schema edge has a "*" or a "?" label, or if the incoming edge of N_C is soft. Label every new node with a new unique variable V_i. If S is not repeatable, remove label V from N_C; otherwise, V will be used by a "group-by" label in R'. For every V_i that was added to N_C, extend N_R with a new child node labeled V_i. If S is repeatable, add a group-by label $\{V\}$ to N_R.

The above procedure is applied recursively to all the nodes of C'. For example, Figure II.15 shows one of the query plans for the query of Figure II.12. First, the *Order* is extended with *Number*, *Status*, *LineItem*, *Price* and *Date*. Then the *LineItem* is extended with all its attributes. The edge between the *Order* and the *LineItem* is soft (indicated by the dotted line) because, according to the schema, *LineItem* is an optional child of *Order*. Since the incoming edge of the *LineItem* is soft, all its outgoing edges are also soft. Group-by labels on *Order* and *LineItem* indicate that nested structures will be constructed for these elements. Given the decomposition of Figure II.16 which includes BLOBs of *Order* elements, another valid plan for the query of Figure II.12 will be identical to the query itself, with a plan decomposition consisting of a single fragment.

We illustrate the translation of query plans into the SQL queries with the following example.

Example 10 *Consider the valid query plan of Figure II.15, which assumes the 4NF decomposition without BLOBs of Figure II.10. This plan will be translated into SQL by the following process. First, we identify the tables that should appear in the SQL* `FROM` *clause. Since the condition tree is partitioned into four fragments, the* `FROM` *clause will feature four tables:*

```
FROM Customer C, Order O,
LineItem L1, LineItem L2
```

Second, for each fragment of the condition tree, we identify variables defined in this fragment that also appear in the result tree. For every such variable, the corresponding fragment table attribute is added to the `SELECT` *clause. In our case, the result includes all variables, with the exception of* $\$P$:

```
SELECT DISTINCT C.name, O.order_id, O.number,
O.status, O.price, O.date,L1.lineitem_id,
L1.part_number, L1.supplier_number,
L1.price, L1.quantity, L1.discount
```

Third, we construct a `WHERE` *clause from the plan condition expression and by inspecting the edges that connect the fragments of the plan decomposition. If the edge that connects a parent fragment* P *with a child fragment* C *is a regular edge, then we introduce the condition*
`tbl_P.parent_attr = tbl_C.parent_ref`
If the edge is "soft", the condition
`tbl_P.parent_attr =* tbl_C.parent_ref`, *where "=*" denotes a left outerjoin. An outerjoin is needed to ensure accurate reconstruction of the original document. For example, an order can appear in the result even if it does not have any lineitems. In our case, the* `WHERE` *clause contains the following conditions:*

```
WHERE C.cust_id = O.cust_ref
AND O.order_id = L2.order_ref
AND O.order_id =* L1.order_ref
AND L2.price > 30000
```

Notice, that the above where clause can be optimized by replacing the outerjoin with a natural join because the selection condition on L2 implies that the order O will have at least one lineitem.

Finally, the clause `ORDER BY O.order_id` *is appended to the query to facilitate the grouping of lines of the same order, which allows the XML result to be constructed by a constant space tagger [79].*

The resulting SQL query is:

```
SELECT DISTINCT C.name, O.order_id, O.number,
O.status, O.price, O.date,L1.lineitem_id,
L1.part_number, L1.supplier_number,
L1.price, L1.quantity, L1.discount
FROM Customer C, Order O,
LineItem L1, LineItem L2
WHERE C.cust_id = O.cust_ref
AND O.order_id = L2.order_ref
AND O.order_id = L1.order_ref
AND L2.price > 30000
ORDER BY O.order_id
```

Now consider a complete decomposition without BLOBs. Recall, that a complete decomposition consists of all possible non-MVD fragments. This decomposition, for instance, includes a non-MVD fragment Customer-Order-LineItem (COL for short) that contains all Customer, Order and LineItem information. This fragment is illustrated in Figure II.16. The COL fragment can be used to answer the above query with only one join, using the query plan illustrated in Figure II.17.

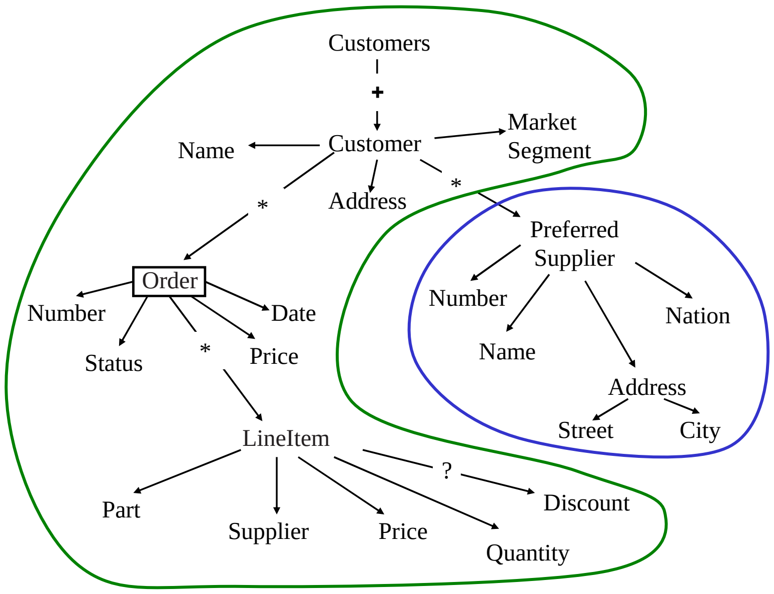


Figure II.16: Inlined schema decomposition used for the experiments

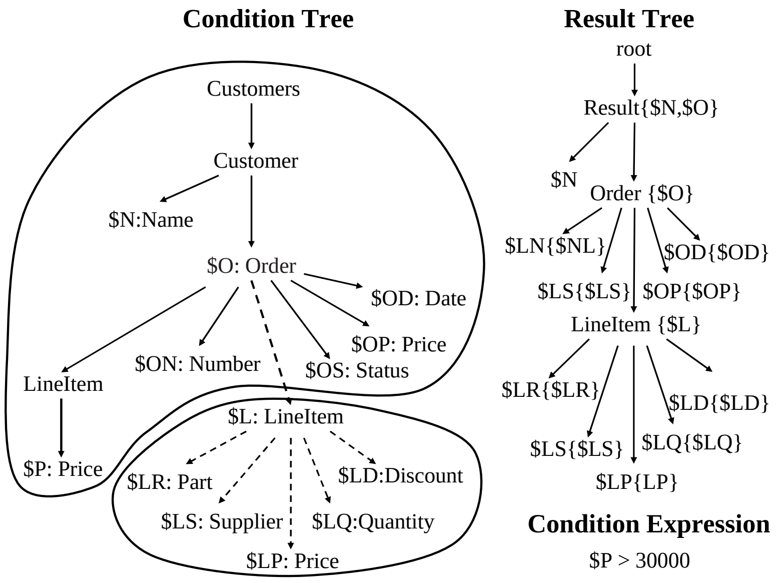


Figure II.17: A Possible Query Plan

```
SELECT DISTINCT COL.name, COL.order_id,
COL.number, COL.status, COL.price, COL.date,
L1.lineitem_id, L1.part_number, L1.price,
L1.supplier_number, L1.quantity, L1.discount
FROM COL, LineItem L1
WHERE COL.order_id = L1.order_ref
AND COL.line_price > 30000
ORDER BY COL.order_id
```

Finally, consider the same complete decomposition that also features Order *BLOBs. We can use the query plan identical to the query itself (Figure II.12), with a single fragment plan decomposition. Again a single join is needed (with Blobs table), but the result does not have to be tagged afterwards. This also means that the ORDER BY clause is not needed.*

```
SELECT DISTINCT COL.cust_name, Blobs.value
FROM COL, Blobs
WHERE COL.line_price > 30000
AND COL.order_id = Blobs.id
```

The XCacheDB also has an option of retrieving BLOB values with a separate query, in order to improve the query response time. Using this option we eliminate the join with the Blobs table. The query becomes

```
SELECT DISTINCT COL.cust_name, COL.order_id
FROM COL WHERE COL.line_price > 30000
```

The BLOB values are retrieved by the following prepared query:

```
SELECT value FROM Blobs WHERE id = ?
```

The above example demonstrates that the BLOBs can be used to facilitate construction of the results, while the non-4NF materialized views can reduce the number of joins and simplify the final query. The BLOBs and inlined decompositions are two independent techniques that trade space for performance. Both of the techniques have their pros and cons.

Effects of the BLOBs

Positive: Use of BLOBs may replace a number of joins with a single join with the Blobs table, which, as our experiments show, typically improves performance. BLOBs eliminate the need for the order-by clause, which improves query performance, especially the response time. BLOBs do not require tagging, which also saves time. BLOBs can be retrieved by a separate query which significantly improves the response time.

Negative: The BLOBs introduce significant space overhead. The join with the Blobs table can be expensive especially when the query results are large.

Effects of the Inlined decomposition

Positive: The denormalized decompositions reduce number of joins, which may lead to better performance. For instance, eliminating high start-up costs of some joins (e.g. hash join), improves query response time. Since the query has fewer joins, it is simpler to process; as the result, query performance is much less dependant on the relational optimizer. During our experiments with the normalized decompositions, we encountered cases when a plan produced by the relational optimizer simply could not be executed by our server. For example, one of such plans called for a Cartesian product of 5000 tuples with 600000. We never encountered such problems while experimenting with the inlined decompositions.

Negative: The scans of denormalized tables take longer because of the increased width. The inlining, also introduces space overhead.

II.E.2 Minimal Plans

Out of the multiple possible valid plans we are interested in the ones that minimize the number of joins.

Definition 14 (Minimal Plan) *A valid plan is* minimal *if its plan decomposition P' contains the smallest possible number of partitions P_i.*

Still, there may be situations where there are multiple minimal plans. In this case the plan generator uses the following heuristic algorithm, which is linear in the size of the query and the schema decomposition. When the algorithm is applied on a minimal non-MVD decomposition it is guaranteed to produce a minimal plan.

- 1. Pick any leaf node N of the query.
- 2. Find the fragment F that covers N and goes as far up as possible (covers the most remote ancestor of N)
- 3. Remove from the query tree, the subtree covered by F
- 4. Repeat the above steps until all nodes of the query are covered.

The advantage of this algorithm is that it avoids joins at the lower levels of the query – where most of the data is usually located. For example, in the TPC-H dataset we used for the experiments (it conforms to the schema of Figure II.3, the `Order` $\bowtie$ `LineItem` join is 40 times bigger (and potentially more expensive) than the `Customer` $\bowtie$ `Order` join.

II.F Implementation

The XCacheDB system [10] of Enosys Software, Inc., is an XML database built on top of commercial JDBC-compliant relational database systems. The ab-

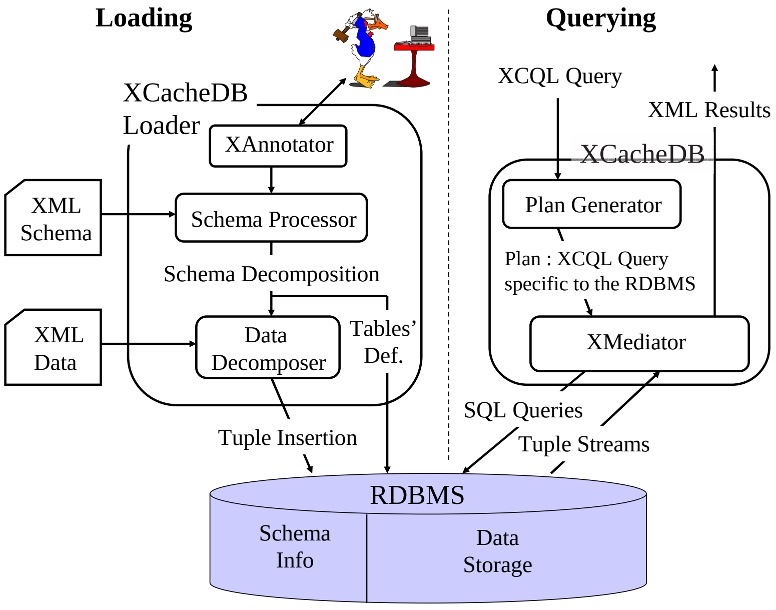
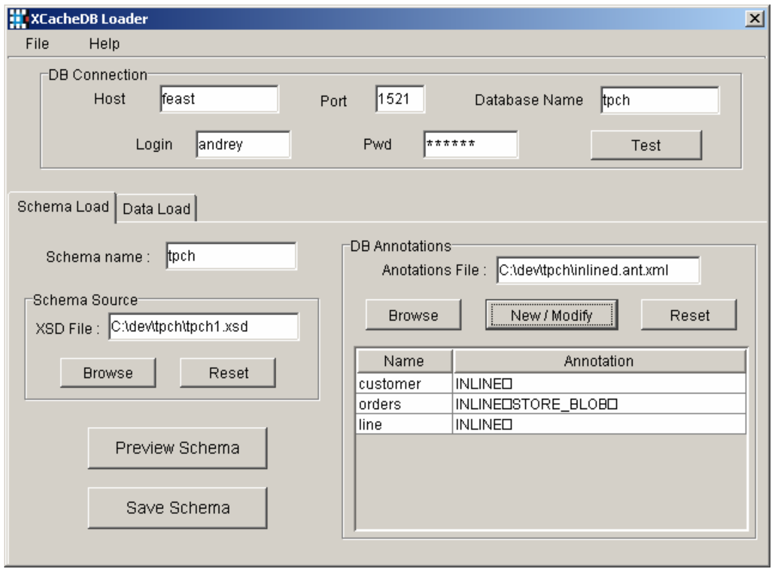


Figure II.18: The XCacheDB architecture



DB Connection

Host: feast Port: 1521 Database Name: tpch

Login: andrey Pwd: ******

Schema name : tpch

Schema Source

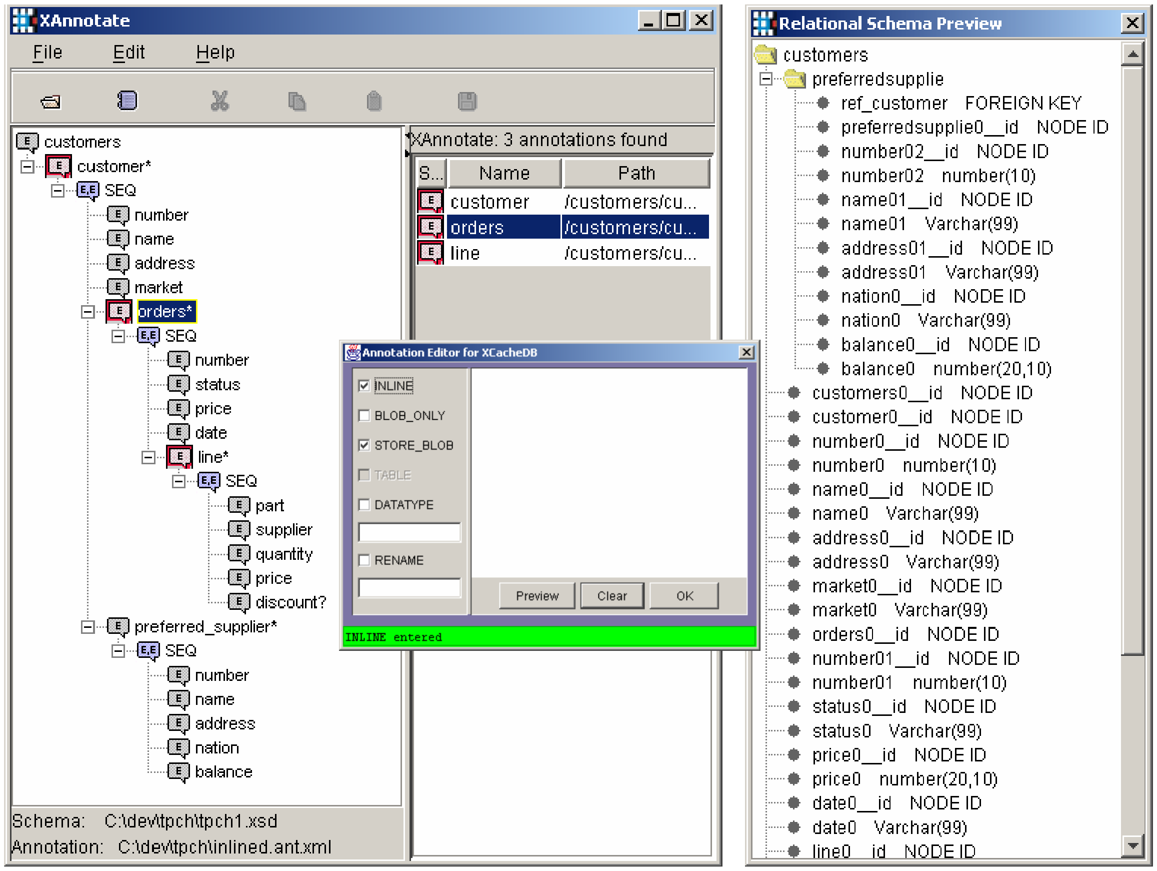
XSD File : C:\dev\tpch\tpch1.xsd

DB Annotations

Anotations File : C:\dev\tpch\inlined.ant.xml

Name	Annotation
customer	INLINE□
orders	INLINE□STORE_BLOB□
line	INLINE□

Figure II.19: The XCacheDB Loader utility



```
XAnnotate
File   Edit   Help

customers
  customer*
    SEQ
      number
      name
      address
      market
      orders*
        SEQ
          number
          status
          price
          date
          line*
            SEQ
              part
              supplier
              quantity
              price
              discount?
      preferred_supplier*
        SEQ
          number
          name
          address
          nation
          balance

Schema:     C:\dev\tpch\tpch1.xsd
Annotation: C:\dev\tpch\inlined.ant.xml
```

XAnnotate: 3 annotations found

S...	Name	Path
E	customer	/customers/cu...
E	orders	/customers/cu...
E	line	/customers/cu...

```
Annotation Editor for XCacheDB
[✓] INLINE
[ ] BLOB_ONLY
[✓] STORE_BLOB
[ ] TABLE
[ ] DATATYPE
[ ] RENAME
Preview   Clear   OK
INLINE entered
```

```
Relational Schema Preview
customers
  preferredsupplie
    ref_customer   FOREIGN KEY
    preferredsupplie0__id   NODE ID
    number02__id   NODE ID
    number02   number(10)
    name01__id   NODE ID
    name01   Varchar(99)
    address01__id   NODE ID
    address01   Varchar(99)
    nation0__id   NODE ID
    nation0   Varchar(99)
    balance0__id   NODE ID
    balance0   number(20,10)
  customers0__id   NODE ID
  customer0__id   NODE ID
  number0__id   NODE ID
  number0   number(10)
  name0__id   NODE ID
  name0   Varchar(99)
  address0__id   NODE ID
  address0   Varchar(99)
  market0__id   NODE ID
  market0   Varchar(99)
  orders0__id   NODE ID
  number01__id   NODE ID
  number01   number(10)
  status0__id   NODE ID
  status0   Varchar(99)
  price0__id   NODE ID
  price0   number(20,10)
  date0__id   NODE ID
  date0   Varchar(99)
  line0__id   NODE ID
```

Figure II.20: Annotating the XML schema and resulting relational schema

stract architecture of Figure I.3 has been reduced to the one of Figure II.18, where the plan translation and construction functions of the query processor are provided by the XMediator [28] product of Enosys Software, Inc. Finally, the "optional user guidance" of Figure I.3 is provided via the *XAnnotator* user interface, which produces a set of *schema annotations* that affect decomposition.

The XCacheDB loader supports acyclic schemas, which by default are transformed into tree schemas. By default, the XCacheDB loader creates the minimal 4NF decomposition. However, the user can control the decomposition using the schema annotations and can instruct the XCacheDB what to inline and what to BLOB. In particular, the XAnnotator (Figure II.20) displays the XML Schema and allows the user to associate a set of annotation keywords with nodes of the schema graph. The following six annotation keywords are supported. We provide a brief informal description of their meaning:

`INLINE`: placed on a schema node n it forces the fragment rooted at n to be merged with the fragment of the parent of n.

`TABLE`: placed on a schema node n directs the loader to create a new fragment rooted at n.

`STORE_BLOB`: placed on a schema node n it indicates that a BLOB should be created for elements that correspond to this node.

`BLOB_ONLY`: implies that the elements that correspond to the annotated schema node should be BLOB-ed and not decomposed any further.

`RENAME`, `DATATYPE`: those annotations enable the user to change names of the tables and columns in the database, and data types of the columns respectively.

A single schema node can have more than one annotation. The only exception is that `INLINE` and `TABLE` annotations cannot appear together, as they contradict each other.

The XCacheDB loader automatically creates a set of indices for each table that it loads. By default, an index is created for every data column to improve performance of selection conditions, but it can be switched off. An index is also created for a parent reference column, and for every node-ID column that gets referenced by another table. These indices facilitate efficient joins between fragments.

Query processing in XCacheDB leverages the XMediator, which can export an XML view of a relational database and allow queries on it. The plan generator takes an XML query, which was XCQL [68] and is now becoming XQuery, and produces a query algebra plan that refers directly to the tables of the underlying relational database. This plan can be run directly by the XMediator's engine, since it is expressed in the algebra that the mediator uses internally.

II.G Experimentation

This section evaluates the impact of BLOBs and different schema decompositions on query performance. All experiments are done using an “TPC-H like” XML dataset that conforms to the schema of Figure II.3. The dataset contains 10000 customers, 150000 orders, ∼120000 suppliers and ∼600000 lineitems. The size of the XML file is 160 MB. Unless otherwise noted, the following system configuration is used. The XCacheDB is running on a Pentium 3 333MHz system with 384MB of RAM. The underlying relational database resides on a dual Pentium 3 300MHz system with 512MB of RAM and 10000RPM hard drives connected to a 40MBps SCSI controller. The database server is configured to use 64MB of RAM for buffer space. We flush the buffers between runs, to ensure the independence of the experiments. Statistics are collected for all tables and the relational database is set to use the cost-based optimizer, since the underlying database allows both cost-based and rule-based optimization. The XCacheDB connects to the database through a 100Mb switched Ethernet network. We also provide experiments with 11Mb wireless Ethernet connection between the systems, and show the effects of a lower-bandwidth, high-latency connection.

All previous work on XML query processing, concentrated on a single performance metric – *total time*, i.e. time to execute the entire query and output the complete result. However, we are also interested in *response time*. We define the response time as the time it takes to output the first ten results.

Queries We use the following three queries (see Figure II.21):

Q1 The selection query of Example 9 returns pairs of customer names and their orders, if the order contains at least one lineitem with *price* $> P$, where P is a parameter that ranges from 75000 (qualifies about 15% of tuples) to 96000 (qualifies no tuples).

Q2 also has a range condition on the supplier. The parameter of the supplier

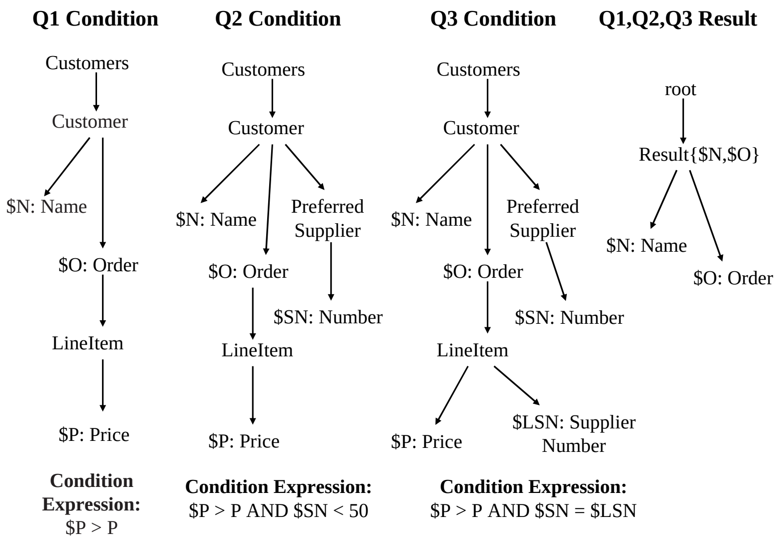


Figure II.21: Three queries used for the experiments

condition is chosen to filter out about 50% of customers on the average. Notice that since this query refers to both orders and suppliers, it cannot be answered using a single non-MVD fragment.

Q3 This query finds customers that have placed expensive orders with preferred suppliers (i.e. customer contains a prefered supplier and an order with an expensive item from this supplier.) Notice the join between `Supplier` and `LineItem`.

Testing various decompositions We compare the following query decompositions:

1. 4NF schema decomposition without BLOBs, which consists of the following four tables: `Customer`, `Order`, `LineItem`, and `PrefSupplier` (Figure II.10). The above four tables occupy 64 MB of disk space. This case corresponds to a typical decomposition considered in the previous work [80, 17].

2. Same 4NF decomposition as above with the addition of a BLOBs table that stores Order subtrees. This table takes up 150 MB.

3. Inlined decomposition of Figure II.16, which includes two non-MVD fragments: `Supplier` and `Customer-Order-Line`. These two tables occupy 137.5 MB. This decomposition also includes Order BLOBs.

We also consider a decomposition that contains an MVD fragment `Customer-Order-Supplier` and a separate table for `LineItem`. However, the experiments show that this approach is not competitive. The space overhead (the two tables take up almost 600 MB) translates into poor query performance.

Discussion The left side of Figure II.22 shows the total execution time of the three queries plotted against the selectivity of the condition on price, which essentially controls the size of the result. For Q1, 1% selectivity translates into a 1.5 MB result XML file. For Q2 and Q3 the rates are about 0.75 MB and 0.4 MB respectively. The right side of Figure II.22 shows the response time of the same queries. Recall, in the response time experiments the queries return top ten top-level objects, i.e. the result size is constantly around 10 KB.

All the "total time" graphs exhibit the same trend. The "4NF" line starts higher than the "inlined" one, because of the time it takes the database to initiate and execute multiway joins. However, the slope of the "inlined" line is steeper because of the space and I/O overhead derived from the denormalization. Table scans take longer on the "inlined" tables.

BLOBs improve performance of the small queries, but their effects also diminish as the result sizes grow. For smaller results (less than 2 MB of XML) 4NF with BLOBs consistently outperforms 4NF without BLOBs by 200% to 300%. As the result sizes increase, join with the BLOBs table becomes more expensive in comparison to the extra joins needed to reconstruct the result fragments.

The main advantage of the XCacheDB is its response time. Both inlining and BLOBs significantly simplify the SQL query which is sent to the relational database, which allows the server to create the result cursor, in some cases, almost instantaneously.

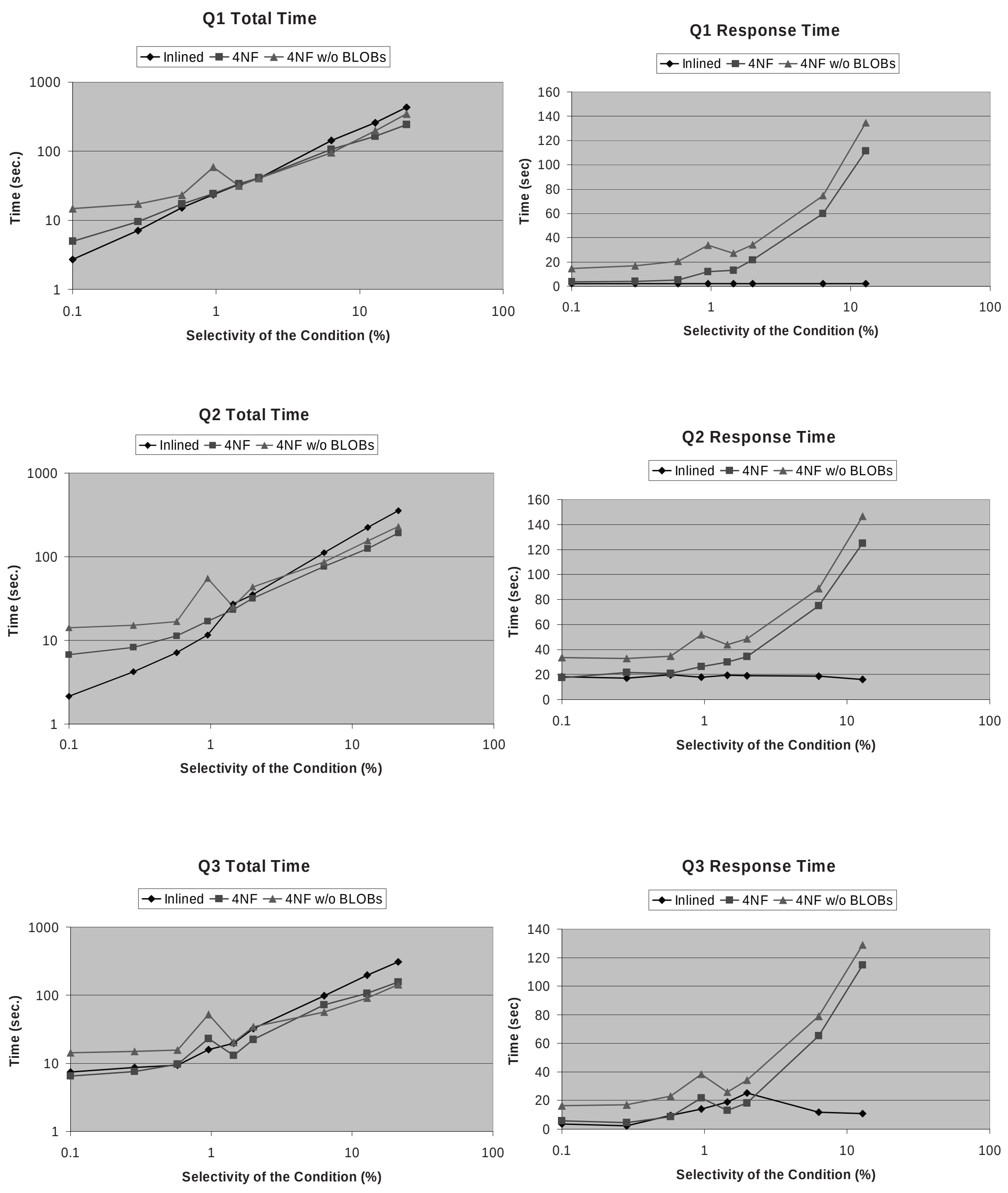


Figure II.22: Experimental results

The irregularities that are be observed in the graphs (e.g. a notch on the "4NF without BLOBs" line of all three "total time" graphs around the 1% selectivity) are mostly due to the different plans picked by the relational optimizer for different values of the parameter. Notice that on the Q3 response time graph the "Inlined" line uncharacteristically dips between the 6-th and 7-th points (selectivity values 1.6% and 4.3%). It turns out that at that point the optimizer reversed the sides of the hash join of `COL` and `Supplier` tables, which improved performance.

II.G.1 Effects of higher CPU/bandwidth ratio

Query processing on the inlined schema requires less CPU resources than on the 4NF schema, since fewer joins need to be performed. However, pre-joined data needs to be read from the disk, which requires more I/O operations than reading data required for the join. If the database optimizer correctly picks join ordering and join strategies, a table will not be scanned more than twice for a join, and most of the time, a single scan will be sufficient [36]. This tradeoff was observed when the database was installed on a 500MHz system with a slow (4200RPM) IDE disk. In this setting, the 4NF decomposition with BLOBs often provided for faster querying than the inlined one. For example, in a fast disk setup a Q1-type query with result size 2 MB according to Figure II.22 takes about 7.5 sec on both 4NF and Inlined schemas. On a server with slower disk the same query took 8.2 sec with the 4NF decomposition and 11.6 sec with the Inlined decomposition.

BLOBs are sensitive to interconnect speeds between the database server and the XCacheDB, since they include tags and structure information in addition to the data itself. BLOB-ed query results are somewhat larger than those containing only atomic values, and on slower, high-latency links, network speed can become the bottleneck. For example, Q1 with BLOBs takes 34.2 seconds to complete on a 11Mb wireless network. The same query in the same setup, but on a 100Mb Ethernet takes only 7.5 sec.

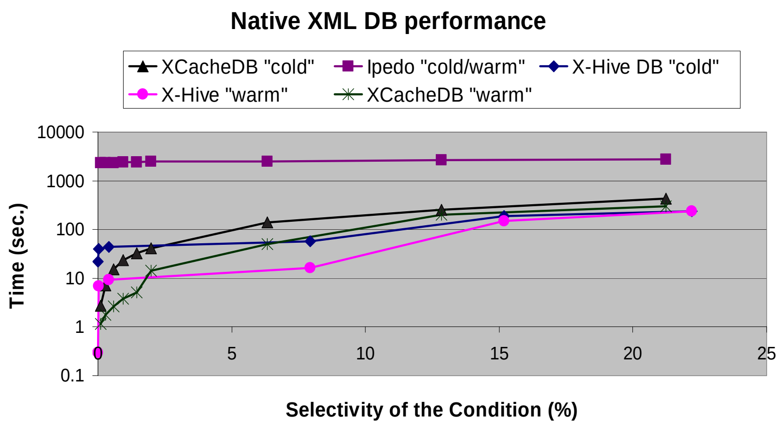


Figure II.23: The total execution time of Q1 on native XML databases vs. XCacheDB

II.G.2 Comparison with a Commercial XML Database

We compared the performance of XCacheDB against two commercial native XML database systems: X-Hive 4.0 [95] and Ipedo 3.1 [45]. For this set of experiments we only measured total execution time, because these two databases could not compete with the XCacheDB in response time, since they are unable to return the first result object before the query execution is completed.

Both systems support subsets of XQuery which include the query Q1, as it appears in Example 9 and as it was used in the XCacheDB experiments above. However, we did not use Q1 because the X-Hive was not able to use the value index to speed-up range queries. Thus, we replaced range conditions on price elements with equality conditions on “part”, “supplier”, and “quantity” elements, which have different selectivities.

For Ipedo we were not able to rewrite the query in a way that would enable the system to take advantage of the value indices. As a result, the performance of the Ipedo database was not competitive (Figure II.23), since a full scan of the database was needed every time to answer the query.

In all previous experiments we measured and reported “cold-start” execution times, which for X-Hive were significantly slower than when the query ran on “warm” cache. For instance, the first execution of a query that used a value index, generated more disk traffic than the second one. It may be the case that X-Hive reads from disk the entire index used by the query. This would explain relatively long (22 seconds) execution time for the query that returned only four results. The second execution of the same query took 0.3 seconds. For the less selective queries the difference was barely noticeable as the “warm” line of Figure II.23 indicates.

We do not report results for the Q3 query, since both X-Hive and Ipedo where able to answer it only by a full scan of the database, and hence they were not competitive.

II.H Conclusions and Future Work

Our approach towards building XML DBMS’s is based on leveraging an underlying RDBMS for storing and querying the XML data in the presence of XML Schemas. We provide a formal framework for schema-driven decompositions of the XML data into relational data. The framework encompasses the decompositions described in prior work and takes advantage of two novel techniques that employ denormalized tables and binary-coded XML fragments suitable for fast navigation and output. The new spectrum of decompositions allows us to trade storage space for query performance.

We classify the decompositions based on the dependencies in the produced relational schemas. We notice that non-MVD relational schemas that feature inlined repeatable elements, provide a significant improvement in the query performance (and especially in response time) by reducing the number of joins in a query, with a limited increase in the size of the database.

We implemented the two novel techniques in XCacheDB – an XML DBMS built on top of a commercial RDBMS. Our performance study indicates that XCacheDB can deliver significant (up to 400%) improvement in query execution time. Most importantly, the XCacheDB can provide orders of magnitude improvement in query response time, which is critical for typical web-based applications.

We identify the following directions for future work:

- Extend to more complex queries.
- Extend our schema model from DAG's to arbitrary graphs. This extension will increase the query processing complexity, since it will allow recursive queries which cannot be evaluated in standard SQL.
- Consider a cost-based approach for determining a schema decomposition given a query mix, along the lines of [17].
- Enhance the query processing to consider plans where some of the joins may be evaluated by the XCacheDB. Similar work was done by [30], however, they focused on materializing large XML results, whereas our first priority is minimizing the response time.

Chapter III

Incremental Validation of XML Data Under Updates

We investigate the incremental validation of XML documents with respect to DTDs [89], under updates consisting of element tag renamings, insertions and deletions. We exhibit an $O(m \log n)$ incremental validation algorithm using an auxiliary structure of size $O(n)$, where n is the size of the document and m is the number of updates. This is a significant improvement over brute-force re-validation from scratch.

We exhibit a restricted class of DTDs called "local" that arise commonly in practice and for which incremental validation can be done in practically constant time by maintaining only a list of counters. We present implementations of both general incremental validation and local validation on an XML database built on top of a relational database.

Our experimentation includes a study of the applicability of local validation in practice, results on the calibration of parameters of the auxiliary data structure, and results on the performance comparison between the general incremental validation technique, the local validation technique, and brute-force validation from scratch.

Both general and local algorithms are also applicable to validation of

XML documents with respect to *XML Schemas* [91], under insertions and deletions. Incremental validation of XML Schema wrt renaming of internal nodes, is more involved and requires a "specialized DTDs" incremental validation algorithm, which can be found in [11]

III.A Background

The emergence of XML as a standard representation format for data on the Web has led to a proliferation of databases that store, query, and update XML data. Typically, valid XML documents must conform to a specified type that places structural constraints on the document. When an XML document is updated, it has to be verified that the new document still satisfies its type. Doing this efficiently is a challenging problem that is central to many applications. Brute-force validation from scratch is not practical when the data are large, because it requires reading and validating the entire database following each update. Instead, it is desirable to develop algorithms for incremental validation. However, this approach has been largely unexplored. In this chapter we investigate the efficient incremental validation of updates to XML documents.

We model XML documents as trees of nested elements. Document Type Definitions (DTDs) [89], which are the basic mechanism for specifying the type of XML documents, can be abstracted as extended context-free grammars (CFGs). Unlike usual CFGs, the productions of extended CFGs have regular expressions on their right-hand sides. An XML document satisfies a DTD if its abstraction as a tree is a derivation tree of the extended CFG corresponding to the DTD. XML Schemas [91] provide XML typing mechanisms that extend DTDs in several ways. Most notable is the ability to decouple the type of an element from its label. In this chapter we use *specialized* DTDs [70], that capture the decoupling of element tags from types, by allowing the type of an element to depend on the full set of node labels (tags) of the XML tree. XML Schema is abstracted as a restriction of specialized DTDs where the type of an element only depends on its label and the type of its parent.

Verifying that a word satisfies a regular expression[1] is the starting point in checking that an XML document satisfies a DTD. An obvious way to do this following an update is to verify it from scratch, i.e. run the updated sequence of labels through the non-deterministic finite automaton (NFA) corresponding to the regular expression. However, this requires $O(n)$ steps, under any reasonable set of unit operations, where n is the length of the sequence (note that, in complexity-theoretic terms, membership of a word in a regular language is complete in NC^1 under DLOGTIME reductions [87].) We can do better by using incremental validation, relying on an appropriate auxiliary data structure. Indeed, we provide such a structure and corresponding incremental validation algorithm that, given a regular expression r, a string s of length n that satisfies r, and a sequence of m updates (inserts, deletes, label renamings) on s, checks[2] in $O(m \log n)$ whether the updated string satisfies r. The auxiliary structure we use materializes in advance relations that describe state transitions resulting from traversing certain substrings in s. These are placed in a balanced tree structure that is maintained similarly to B-trees and is well-behaved under insertions and deletions. The size of the auxiliary structure is $O(n)$. In addition, we provide an $O(m \log n)$ time algorithm that maintains the auxiliary structure, so that subsequent updates can also be incrementally validated.

Our approach to incremental validation of trees with respect to DTDs, specialized DTDs and XML Schemas builds upon the incremental validation algorithm for strings. DTDs turn out to be easier to validate than specialized DTDs, whereas XML Schemas fall between specialized DTDs and DTDs in difficulty. Indeed, based on the algorithm for string validation, incremental validation of m updates to a tree T with respect to a DTD can be done in time $O(m \log |T|)$

[1]A word *satisfies* a regular expression if it belongs to the corresponding language.

[2]For readability, we provide here the complexity with respect to the string and update sequence, for fixed (specialized) DTD or regular expression. The combined complexity is spelled out in the paper.

using an auxiliary structure of size $O(|T|)$ which can also be maintained in time $O(m \log |T|)$. The same complexity results apply to XML Schemas for all update operations, except internal node renamings, which require extension of the algorithm to the specialized DTDs that can be found in [11].

We then consider a restricted class of DTDs called “local”, that arises very frequently in practice. Although we did not pursue this, we expect that “local” XML Schemas are equally frequent and their incremental validation can benefit in the same way. Intuitively, these are DTDs using regular expressions for which membership after an update can be determined locally, by examining only substrings within bounded distance from the update position. This allows for a very efficient incremental validation algorithm. Although the theoretical worst-case data complexity of validating m updates is still $O(m \log |T|)$, and the auxiliary structure has size $O(i \log(|T|/i))$, where i is the number of internal nodes of T, the algorithm can be implemented very efficiently. Furthermore, if no internal nodes may be renamed, there is no need for an auxiliary data structure. In practice, the technique provides constant time validation, with space overhead of $O(i)$ counters that are represented very efficiently by usual 32-bit integers, which are sufficient when the maximum size of element lists in the database is 2^{32}. In addition to its simple implementation and efficiency, local validation is very frequent. We tested 60 DTDs from OASIS (see [20]) and only 10 DTDs were not local.

Finally, in the last part of the paper, we evaluated the general DTD incremental validation algorithm, the validation algorithm for local DTDs, and a brute-force (re-)validation algorithm on an XML database that operates on top of a commercial RDBMS system. We describe the XML database and the implementation of the necessary data structures on top of an RDBMS. We discuss the applicability of the validation algorithm for local DTDs and a set of performance results that indicate its superiority over general incremental validation in the case of local DTDs. The experiments also quantify the significant superiority of both incremental validation algorithms over the brute-force technique. Finally, the experiments provide useful data for the optimization of various parameters of the data structures.

Related Work As mentioned earlier, XML databases need to efficiently validate updates on their content. Ipedo's XML database [45] validates update commands with respect to XML Schemas; however, to our knowledge no technical information is publicly available on the underlying structures and algorithms. Another application where efficient validation is useful is XML editors (see [97] for a survey of available products). Some XML editors like XMLMind [99] and XMLSpy [100] feature incremental validation of DTDs. Recently, XMLSpy also included validation of XML Schemas [100]. No information is provided on their incremental validation algorithms.

Note that our abstraction of the content models of DTDs [89] by arbitrary regular expressions removes the requirement for 1-unambiguous regular expressions. The incremental validation algorithm of [14] utilizes the fact that 1-unambiguousness leads to deterministic Glushkov automata for the regular expressions of DTDs. Consequently [14] use the Glushkov automata to develop a local incremental validation algorithm. Our definition of "locality" is more general in two aspects. First, the "CF" concept of locality in [14] corresponds to particular cases of 1-local of our development. We define a more general concept of "k-local", whose significance is that an update can be validated by inspecting only the siblings within distance k from the update. Second, our definition of locality is based on the locality of the minimal automata of the regular expressions, while [14] base the definition on the Glushkov automata. We prove that if an automaton (including potentially a Glushkov automaton) recognizes a regular expression is local then the minimal automaton is also local, but the converse does not necessarily hold. Hence detecting locality using the minimal automata provides a wider definition. Our locality property is orthogonal to the "1,2 CF" property of [14], which was designed exclusively to validate individual atomic updates. Instead, our validation algorithm supports transactions consisting of multiple updates.

Closely related to incremental validation is incremental parsing, which is key to incremental program compilation. Research on incremental parsing has focused on LR parsing [37, 92, 47, 52, 72] and LL (recursive descent parsing) [62, 55, 56], since programming languages are typically described by LR(0), LR(1), LL(1), LALR(1) and LL(1) grammars. All techniques start by parsing the input text and producing a parse tree, which is typically annotated with auxiliary information. The parse tree is updated as a result of the updates to the input text. A typical theme of the incremental parsing techniques is identifying minimal structural units of the parse tree that are affected by the modifications (see [37] for LR(0) parsing and [52] for a generalization to LR(k).) However, the performance of the incremental parsing algorithms is hard to compare to our validation algorithm because of the differences in settings and goals, which typically involve minimization of the changes on the parse tree. Indeed, the best-case performance of incremental parsers will generally beat the one of our regular expression validation algorithm, which always takes $O(\log n)$ steps for a single update. This is because incremental parsers take advantage of natural "termination points" used in programming languages syntax [56], that typically occur close to the update. Logarithmic complexity in the size of the string is achieved for LALR grammars by [92] but only if the grammar is such that its parse trees have depth $O(\log n)$ for a string of length n. One can easily see that there are LALR grammars that do not meet this property, and neither do the CFGs corresponding to DTDs. Furthermore, [92] require that the interpretation of iterative sequences be independent of the context. In particular, [92] provide the following "bad grammar", which recognizes the regular expression $(a|b)x^*$

$$S \rightarrow aC^+|bD^+$$
$$C \rightarrow x$$
$$D \rightarrow x$$

This grammar is problematic for their algorithm because the reduction of an x to either a C or a D is determined by the initial symbol in the sentence, which is arbitrarily distant. In this case their algorithm needs $O(n)$ recomputation, where n is the size of the string. Notice that our divide-and-conquer algorithm for the incremental validation of regular expressions does not pose any restriction on the regular expression.

The complexity of validation is related to that of membership of a word in a regular language, and of a tree in a regular tree language. The problem of word membership in a regular language is known to be complete in uniform NC^1 under DLOGTIME reductions [87] and acceptance of a tree over a ranked alphabet by a tree automaton is complete in uniform NC^1 under DLOGTIME reductions if the tree is presented in prefix notation [57], and complete in LOGSPACE if the tree is presented as a list of its edges [78]. To our knowledge, no complexity results exist on the incremental variants of these problems, with the exception of a result of [71] discussed below.

Incremental evaluation of queries by first-order means is studied by [26] using the notion of first-order incremental evaluation systems (FOIES) A related descriptive complexity approach to incremental computation is developed by Patnaik and Immerman in [71]. They define the dynamic complexity class Dyn-FO (equivalent to FOIES), consisting of properties that can be incrementally verified by first-order means. They exhibit various problems in Dyn-FO, such as multiplication, graph connectivity, and bipartiteness. Most relevant to our work, they show that membership of a word in a regular language is in Dyn-FO. For label renamings, they sketch an approach similar to ours. The incremental algorithm and auxiliary structure for node insertions and deletions that modify the length of the string are not spelled out. Also, no extension to regular tree languages is discussed. The study in [71] is pursued in [41], where an extension of Dyn-FO is introduced and it is shown that the single-step version of the circuit value problem is complete in Dyn-FO under certain reductions. Complexity models of incremental computation are considered in [61]. The focus is on the classes *incr*-POLYLOGTIME (*incr*-POLYLOGSPACE) of properties that can be incrementally verified in polylogarithmic time (space). Interesting connections to parallel complexity classes are exhibited, as well as complete problems for classical complexity classes under reductions in the above incremental complexity classes.

Organization The chapter is organized as follows. Section III.B presents our abstraction of XML documents and DTDs. It also presents specialized DTDs, their restriction capturing XML Schemas, and their connection to tree automata. We also spell out formally the incremental validation problem and the assumptions made in our complexity analysis. In Section III.C we examine the incremental validation of strings with respect to regular expressions and develop the core divide-and-conquer strategy used later for DTD and XML Schema validation. Section III.D presents an $O(m \log |T|)$ validation algorithm for DTDs and an $O(m)$ algorithm for local DTDs. Those algorithms are also applicable to the incremental validation of XML Schemas wrt insertions, deletions and leaf node renamings. Section III.E describes our implementation of incremental validation for DTDs. Section III.F presents an evaluation of the applicability of local validation, and experimental results comparing our general incremental algorithm for DTDs, the algorithm for local DTDs, and a brute-force revalidation algorithm. Section IV.H contains concluding remarks and future work.

III.B Basic Framework

We introduce here the basic formalism used throughout the paper, including our abstractions of XML documents, DTDs, and XML Schemas. We also recall basic definitions relating to tree automata.

Labeled ordered trees We abstract XML documents as labeled ordered trees. Our abstraction ignores data values present in XML documents, because their validation with respect to an XML Schema is trivial. For example, an XML document holding ads for used cars and new cars is shown in Figure III.1 (left), together with its abstraction as a labeled tree.

An *ordered labeled tree* over finite alphabet Σ is a pair $T = \langle t, \lambda \rangle$, where t is an ordered tree and λ is a mapping associating to each node n of t a label $\lambda(n) \in \Sigma$. Trees are assumed by default to be unranked, i.e. there is no fixed bound on the number of children each node may have. The set of all labeled ordered trees over Σ is denoted by $\mathcal{T}_\Sigma$. We sometimes denote a tree consisting of a root v with subtrees $T_1 \ldots T_k$ by $v(T_1 \ldots T_k)$. We will also consider binary trees, where each node has at most two children. If every internal node has *exactly* two children, the binary tree is called *complete*.

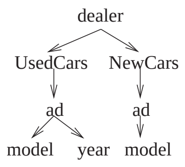
We assume that finding (i) the label, (ii) the parent, (iii) the immediate left (right) sibling, and (iv) the first child of a specified node, are unit operations, i.e., they can be accomplished in $O(1)$.

Types and DTDs As usual, we define XML document types in terms of the document's structure alone, ignoring data values. The basic specification method is (an abstraction of) DTDs. A DTD consists of an extended context-free grammar over alphabet Σ (we make no distinction between terminal and non-terminal symbols). In an extended CFG, the right-hand sides of productions are regular expressions over Σ. An ordered labeled tree $\langle t, \lambda \rangle$ over Σ satisfies a DTD d if the tree $\langle t, \lambda \rangle$ is a derivation tree of the grammar. For example, the tree is valid with respect to the DTD in Figure III.1.

The start symbol of a DTD d is denoted by $root(d)$. We can assume without loss of generality that for each $a \in \Sigma$ the DTD has a single rule $a \ \rightarrow r_a$ with a on the left-hand side. and we denote by N_a a standard non-deterministic finite-state automaton (NFA) recognizing the language r_a. The set of labeled trees satisfying a DTD d is denoted by $sat(d)$.

We use the following notation for NFA. An NFA is a 5-tuple

```
<dealer>
 <UsedCars>
  <ad>
   <model>Honda</model>
   <year>92</year>
  </ad>
 </UsedCars>
 <NewCars>
  <ad>
   <model>BMW</model>
  </ad>
 </NewCars>
</dealer>
```



```
<!DOCTYPE dealer>
<!ELEMENT dealer
 (UsedCars, NewCars)>
<!ELEMENT UsedCars (ad*)>
<!ELEMENT NewCars (ad*)>
<!ELEMENT ad (model, year?)>
<!ELEMENT model PCDATA>
<!ELEMENT year PCDATA>
```

$root$	:	$dealer$
$dealer$	$\rightarrow$	$UC\ NC$
UC	$\rightarrow$	ad^*
NC	$\rightarrow$	ad^*
ad	$\rightarrow$	$model\ (year \mid \epsilon)$
$model$	$\rightarrow$	ϵ
$year$	$\rightarrow$	ϵ

$root : d^t$	
$d^t \rightarrow UC^t\ NC^t$	$\mu(d^t) = dealer$
$UC^t \rightarrow (ad^u)^*$	$\mu(UC^t) = UC$
$NC^t \rightarrow (ad^n)^*$	$\mu(NC^t) = NC$
$ad^u \rightarrow m^t\ y^t$	$\mu(ad^u) = ad$
$ad^n \rightarrow m^t$	$\mu(ad^n) = ad$
$m^t \rightarrow \epsilon$	$\mu(m^t) = model$
$y^t \rightarrow \epsilon$	$\mu(y^t) = year$

Figure III.1: XML, DTD and specialized DTD (UC and NC stand for `UsedCars` and `NewCars`)

$N = \langle \Sigma, Q, q_0, F, \delta \rangle$ where Σ is a finite alphabet, Q is a finite set of *states*, $q_0 \in Q$ is the *start state*, $F \subseteq Q$ is the set of *final states*, and δ is a mapping from $\Sigma \times Q$ to $\mathcal{P}(Q)$. A string $a_1 \ldots a_n$ is accepted by N iff there exists a mapping $\sigma : \{1, \ldots, n\} \to Q$ such that $\sigma(a_1) \in \delta(a_1, q_0)$, $\sigma(a_n) \in F$, and for each $i, 1 \le i < n$, $\sigma(a_{i+1}) \in \delta(a_{i+1}, \sigma(a_i))$. The set of strings accepted by N is denoted $L(N)$. N is a *deterministic finite-state automaton* (DFA) iff δ returns singletons on each input. Recall that for each regular expression r there exists an NFA N whose number of states is linear in r, such that N accepts the regular language r. In general, a DFA accepting r requires exponentially many states wrt r. However, for certain classes of regular expressions, the corresponding DFA remains linear in the expression. One such class consists of the 1-unambiguous regular languages [19]. This is relevant in the context of XML types, since DTDs and XML Schemas require the regular expressions used to specify the contents of elements to be 1-unambiguous.

An important limitation of DTDs is the inability to separate the *type* of an element from its *name*. For example, consider the dealer document in Figure III.1. Used cars have model and year while new cars have model only. There is no mechanism to specify this using DTDs, since rules depend only on the name of elements, and not on its context. To overcome this limitation, XML Schema provides a mechanism to decouple element names from their types and thus allow context-dependent definitions of their structure. We abstract and extend this mechanism using the notion of *specialized DTD* (studied in [70] and equivalent to formalisms proposed in [15, 21]).

Definition 15 (Specialized DTD) *A* specialized DTD *is a 4-tuple $\langle \Sigma, \Sigma^t, d, \mu \rangle$ where Σ is a finite alphabet of labels, Σ^t is a finite alphabet of* types, *d is a DTD over Σ^t and μ is a mapping from Σ^t to Σ.*

Intuitively, Σ^t provides, for each $a \in \Sigma$, a set of types associated to a, namely those $a^t \in \Sigma^t$ for which $\mu(a^t) = a$. In our specialized DTD example (lower right corner of Figure III.1) we create two types for the element *ad*: a type ad^n whose content is just a "model" type, and a type ad^u whose content is "model" and "year". Note that μ induces a homomorphism on words over Σ^t, and also on trees over Σ^t (yielding trees over Σ). We also denote by μ the induced homomorphisms.

Let $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$ be a specialized DTD. A tree t over Σ satisfies τ (or is *valid* wrt τ) if $t \in \mu(sat(d))$. Thus, t is a homomorphic image under μ of a derivation tree in d. Equivalently, a labeled tree over Σ is valid if it can be "specialized" to a tree that is valid with respect to the DTD over the alphabet of types. The set of all trees over Σ that are valid w.r.t. τ is denoted $sat(\tau)$. When τ is clear from the context, we simply say that a tree is *valid*.

An XML Schema is abstracted as a specialized DTD $\langle \Sigma, \Sigma^t, d, \mu \rangle$ where two additional constraints apply: For every rule $a \rightarrow r_a$ of d, the regular expression r_a does not contain two types a^t and $a^{t'}$ such that $\mu(a^t) = \mu(a^{t'})$. Intuitively, the constraint implies that for every node v of a tree t that satisfies an XML Schema, the type of v is a function of its label and of the type of its parent.

The incremental validation problem Given a (specialized) DTD τ, a tree $T \in sat(\tau)$, and a sequence s of updates to T yielding another tree T', we wish to efficiently check if $T' \in sat(\tau)$.[3] In particular, the cost should be less than re-validation of T' from scratch. The individual updates are the following:

(a) replace the current label of a specified node by another label,

(b) insert a new leaf node after a specified node,

(c) insert a new leaf node as the first child of a specified node, and

(d) delete a specified node; if the node is an internal one, the subtree rooted at the node is also deleted.

We allow some cost-free one-time pre-processing to initialize incremental validation, such as computing the NFA corresponding to the regular expressions

[3]Notice that a subsequence (prefix) of s may produce a tree $T'' \notin sat(\tau)$, while the complete sequence produces a consistent tree $T' \in sat(\tau)$.

used by the DTDs. We will also initialize and then maintain an auxiliary structure $\mathcal{A}(T)$ to help in the validation. The cost of the incremental validation algorithm is evaluated with respect to:

(a) the time needed to validate T' using T and $\mathcal{A}(T)$, as a function of $|T|$ and $|s|$

(b) the time needed to compute $\mathcal{A}(T')$ from T, s, and $\mathcal{A}(T)$,

(c) the size of the auxiliary structure $\mathcal{A}(T)$ as a function of $|T|$.

The complexity analysis is provided in terms of the number of update operations and will also make explicit the combined complexity taking into account the specialized DTD.

The algorithms can be trivially extended to accomodate insertions of subtrees. In this case the provided algorithmic complexity results are modified to account for the straightforward non-incremental validation of the subtree.

III.C Warmup: Incremental Validation of Strings

As warmup to the validation problem, we consider in this section the incremental validation of *strings* with respect to a regular language specified by an NFA. We first consider the case when all updates consist of label renamings. We discuss inserts and deletes later.

Consider an NFA $N = \langle \Sigma, Q, q_0, F, \delta \rangle$, and a string $a_1 \ldots a_n \in L(N)$. For compatibility with our tree formalism, we view a string as a sequence of nodes (or elements) each of which has a label. When there is no confusion we sometimes blur the distinction between an element and its label.

Consider a sequence of element renamings $u(a_{i_1}, b_1), \ldots, u(a_{i_m}, b_m)$, where $i_1 < i_2 < \ldots < i_m$. The renaming $u(a_{i_j}, b_j)$ requires that the label of element a_{i_j} be renamed to b_j. We would like to efficiently check whether the updated string

$$a_1 \ldots a_{i_1 - 1} b_1 a_{i_1 + 1} \ldots a_{i_m - 1} b_m a_{i_m + 1} \ldots a_n \;\in\; L(N).$$

Validating the new string from scratch by running it through the automaton N takes $O(n|Q|^2 \log |Q|)$. We can easily do better by maintaining some auxiliary information. For simplicity in the presentation, we assume that we can find the rank of a specified node among its siblings in $O(1)$. This assumption is removed later.

Consider the case of a single renaming $u(i, b)$ for $1 \leq i \leq n$. Suppose that we have pre-computed, for each i, $1 < i < n$, the sets $\mathsf{Pre}(i) = \delta(q_0, a_1 \dots a_{i-1})$ and $\mathsf{Post}(i) = \{s \mid \delta(s, a_{i+1} \dots a_n) \in F\}$. If we precompute Pre and Post in arrays then we can retrieve $\mathsf{Pre}(i)$ or $\mathsf{Post}(i)$ in $O(|Q|)$. An $O(|Q|^2)$ algorithm for checking whether the string is in $L(N)$ following the update $u(i, b)$ is now obvious: If there is a state $s_1 \in \mathsf{Pre}(i)$, a state $s_2 \in \mathsf{Post}(i_1 + 1)$ such that $s_2 \in \delta(b, s_1)$ then the updated string is in $L(N)$.

However, the Pre and Post technique does not scale to m updates. Furthermore, maintaining Pre and Post is problematic because, following each update $u(i, b)$, we need to recompute all $\mathsf{Pre}(j)$ for $j > i$ and $\mathsf{Post}(j)$ for $j < i$. This requires $O(n|Q|^2 \log |Q|)$ time.

As the next step in the warmup, we can try to keep some additional auxiliary information in order to better handle multiple updates. For each i, j, $1 \leq i < j \leq n$, let T_{ij} be the *transition relation* $\{\langle p, q\rangle \mid p, q \in Q, q \in \delta(p, a_i \dots a_j)\}$. Note that $T_{ij} = T_{ik} \circ T_{kj}$, $i < k < j$, where $\circ$ denotes composition of binary relations. We also denote by δ_a the relation $\{\langle p, q\rangle \mid q \in \delta(p, a)\}$ for $a \in \Sigma$. If all T_{ij} are available, then checking validity of the updated string $a_1 \dots a_{i_1-1} b_1 a_{i_1+1} \dots a_{i_m-1} b_m a_{i_m+1} \dots a_n$ reduces to verifying that

$$\langle q_0, f\rangle \in T_{0(i_1-1)} \circ \delta_{b_1} \circ T_{(i_1+1)(i_2-1)} \circ \dots \circ T_{(i_m+1)(n)}$$

for some $f \in F$. This takes time $O(m|Q|^2 \log |Q|)$, if we assume that we have pre-computed in a 2-dimensional array all relations T_{ij}. In particular, the composition of two relations is a join operation. It can be accomplished in $O(|Q|^2 \log |Q|^2) = O(|Q|^2 \log |Q|)$ by employing sort-merge join. Each relation is sorted in $O(|Q|^2 \log |Q|)$ and then they are merged in $O(|Q|^2)$. The same complexity can be derived if we assume binary tree indices on each attribute of the relations and we

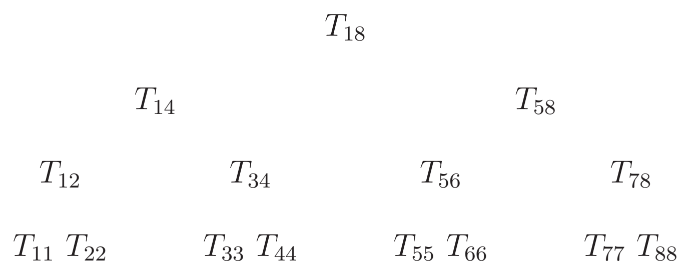

T_{18}
T_{14} T_{58}
T_{12} T_{34} T_{56} T_{78}
T_{11} T_{22} T_{33} T_{44} T_{55} T_{66} T_{77} T_{88}

Figure III.2: The tree $\mathcal{T}_{18}$

employ index-based join [36]. The size of the array required for the precomputation is $n^2|Q|^2$. However, maintaining the precomputed structure is prohibitively expensive, since we have to recompute every relation T_{ij} if there is an update between the ith and jth position of the string. We are therefore led to consider a more promising approach, which provides the basis for the solution we adopt.

Divide-and-conquer validation We describe a divide-and-conquer approach that allows validating a sequence of m renamings to a string of length n, as well as update the auxiliary structure, in $O(m|Q|^2 \log|Q| \log n)$ time. The size of the auxiliary structure is $O(|Q|^2 n)$. Note that the approach below is similar to that briefly sketched in [71].

For simplicity, assume first that n is a power of 2, say $n = 2^k$. The main idea is to keep as auxiliary information just the T_{ij} for intervals $[i, j]$ obtained by recursively splitting $[1, n]$ into halves, until $i = j$. More precisely, consider the *transition relation tree* $\mathcal{T}_n$ whose nodes are the sets T_{ij}, defined inductively as follows:

- the root is $T_{1,2^k}$
- each node T_{ij} for which $j - i > 0$ has children T_{ik} and $T_{(k+1)j}$ where $k = \frac{j-i+1}{2}$,
- T_{ii} are leaves, $1 \leq i \leq n$.

For example, $\mathcal{T}_8$ is shown in Figure III.2.

Note that $\mathcal{T}_n$ has $n + (n/2) + \ldots + 2 + 1 = 2n - 1$ nodes and has depth $\log n$. Thus, the size of the auxiliary structure is $O(n|Q|^2)$.

Consider now a string $a_1 a_2 \ldots a_n \in L(N)$, and a sequence of renamings $u(i_1, b_1), u(i_2, b_2), \ldots, u(i_m, b_m)$, where $i_1 < i_2 < \ldots < i_m$. The updated string is $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n$. Note that the relations T_{ij} that are affected by the updates are those laying on the path from a leaf $T_{i_v i_v}$ $(1 \leq v \leq m)$ to the root of $\mathcal{T}_n$. Let $\mathcal{I}$ be the set of such relations, and note that its size is at most $m \log n$.

The tree $\mathcal{T}_n$ can now be updated by recomputing the T_{ij}'s in $\mathcal{I}$ bottom-up as follows: First, the leaves $T_{i_v i_v} \in \mathcal{I}$ are set to δ_{b_v}, $1 \leq v \leq m$. Then each $T_{ij} \in \mathcal{I}$ with children T_{iv} and T_{vj} for which at least one has been recomputed is replaced by $T_{iv} \circ T_{vj}$. Thus, at most $m \log n$ T_{ij}'s have been recomputed, each in time $O(|Q|^2 \log |Q|)$, yielding a total time of $O(m|Q|^2 \log |Q| \log n)$.

The validation of the string $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n$ is now trivial: it is enough to check, in the updated auxiliary structure, that $\langle q_0, f \rangle \in T_{1n}$ for some $f \in F$. Thus, validation is also done in time $O(m|Q|^2 \log |Q| \log n)$.

The above approach can easily be adapted to strings whose length is not a power of 2 (for example, by appropriately truncating $\mathcal{T}_{2^k}$ where $k = \lceil \log n \rceil$).

Dealing with inserts and deletes We next extend the divide-and-conquer approach outlined for renamings to the case when node inserts and deletes are also allowed. The above approach no longer works, for two reasons: First, inserts and deletes cause the position of nodes in the string to change. Second, the length n of the string, and therefore the set of relevant intervals used to construct $\mathcal{T}_n$, are now dynamic. Due to these differences, inserts and deletes would require recomputing the entire tree $\mathcal{T}_n$, which is inefficient. Instead, we would like to use a tree structure $\mathcal{T}$ that can be incrementally maintained under inserts and deletes, as well as renamings, while preserving the properties that enabled our divide-and-conquer approach. Most importantly, the tree should continue to be balanced and have depth $O(\log n)$. This suggests adopting an approach based on B-trees, that we describe next. We assume basic familiarity with B-trees (e.g., see [36]).

The B-tree variant we use is the 2-3 tree, which was a precursor to B-trees [23]. Each node contains 3 cells. Each cell is either empty or contains a transition relation T_s corresponding to some subsequence s of $v_1 \dots v_n$, conforming to the rules described below. At most one of the 3 cells in a node can be empty, assuming $n \geq 2$. Each nonempty cell is either contained in a leaf node or has one node (with three cells) as a child. The following rules apply to the transition relations stored in the cells:

- if the root has two nonempty cells containing the relations T_{s_1} and T_{s_2} (resp. three cells containing the relations T_{s_1}, T_{s_2} and T_{s_3}) then $T_{s_1} \circ T_{s_2} = T_{[v_1 \dots v_n]}$ (resp. $T_{s_1} \circ T_{s_2} \circ T_{s_3} = T_{[v_1 \dots v_n)]}$);
- if an internal cell contains a relation T_s and its child node contains T_{s_1}, T_{s_2} (resp. T_{s_1}, T_{s_2}, and T_{s_3}) then $T_s = T_{s_1} \circ T_{s_2}$ (resp. $T_s = T_{s_1} \circ T_{s_2} \circ T_{s_3}$);
- the sequence of non-empty leaf cells is $T_{s_1} \dots T_{s_n}$ and $T_{s_i} = T_{[v_i]}$, $1 \leq i \leq n$.

We also maintain pointers providing in $O(1)$, for each element v in the input string, the leaf cell T_s for which the singleton s consists of v. Note that the position of the element is never recorded explicitly.

For example, the left part of Figure III.3 shows a sequence of seven nodes, several subsequences, and the corresponding tree. Note that the subscript of a node does not necessarily indicate its position in the string. Each sequence s_i is the singleton sequence n_i, for $i \in \{1, 2, 3, 5, 6, 7, 9\}$.

The requirement of having 3 cells per node of which at least 2 are non-empty ensures that the tree $\mathcal{T}$ remains balanced and of depth $O(\log n)$ as it is updated. This follows from the standard analysis of B-tree behavior under the maintenance algorithm [36], which we describe here. In a disk-based implementation one should increase the maximum number of cells per node. Furthermore, the leaf node cells need not correspond to singleton element lists. Indeed, we

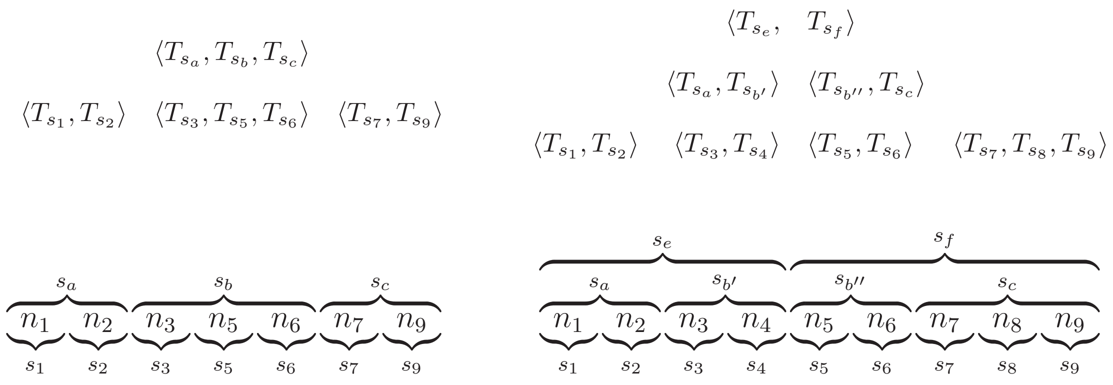

$\langle T_{s_a}, T_{s_b}, T_{s_c} \rangle$
$\langle T_{s_1}, T_{s_2} \rangle$ $\langle T_{s_3}, T_{s_5}, T_{s_6} \rangle$ $\langle T_{s_7}, T_{s_9} \rangle$
s_a s_b s_c
n_1 n_2 n_3 n_5 n_6 n_7 n_9
s_1 s_2 s_3 s_5 s_6 s_7 s_9
$\langle T_{s_e}, T_{s_f} \rangle$
$\langle T_{s_a}, T_{s_{b'}} \rangle$ $\langle T_{s_{b''}}, T_{s_c} \rangle$
$\langle T_{s_1}, T_{s_2} \rangle$ $\langle T_{s_3}, T_{s_4} \rangle$ $\langle T_{s_5}, T_{s_6} \rangle$ $\langle T_{s_7}, T_{s_8}, T_{s_9} \rangle$
s_e s_f
s_a $s_{b'}$ $s_{b''}$ s_c
n_1 n_2 n_3 n_4 n_5 n_6 n_7 n_8 n_9
s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9

Figure III.3: A $\mathcal{T}$ tree before and after the insertion of nodes n_4 and n_8

reduce storage requirements by associating each leaf node cell with a substring. Section III.F provides an evaluation of the performance effect of the number of cells in nodes and of the substring size corresponding to leaf node cells.

Recall that we wish to validate strings with respect to an NFA $N = \langle \Sigma, Q, q_0, F, \delta \rangle$. We describe below the maintenance algorithm for $\mathcal{T}$. Once $\mathcal{T}$ is computed for the current string, validation is easy: check that for some $f \in F$, $\langle q_0, f \rangle$ belongs to the composition of the sets T_s in the cells of the root node of $\mathcal{T}$, at cost $O(|Q|^2 \log |Q|)$.

The auxiliary structure $\mathcal{T}$ corresponding to a valid string w is initialized by starting from the empty string and constructing w by a sequence of inserts, using the maintenance algorithm. Then $\mathcal{T}$ is maintained incrementally as follows. If the update is a renaming of element v, $\mathcal{T}$ is updated much like $\mathcal{T}_n$: we use the index to find the leaf cell of T_v corresponding to v, then update all sets T_s along the path from T_v to the root. This involves $O(\log n)$ updates.

If the update is the insertion or deletion of a new labeled element, the maintenance algorithm mimicks the one for B-trees. In particular, recall that if nodes in a B-tree become too full as the result of an insertion they are split, and if they contain fewer than two non-empty cells as a result of a deletion they are either merged with a sibling node or non-empty cells are transferred from a sibling node. The node splits and merges may propagate all the way to the root. Due to the similarity to classical B-tree maintenance we omit the details but illustrate how to handle the first variant of insertion; deletion and the second variant of insertion are similar. Assume that an element y with label a is inserted after element x in the current string. If there is some empty cell in the leaf node n of $\mathcal{T}$ containing the set T_x corresponding to x we insert the relation $T_y = \delta_a$ in the cell following that for x and we revise the appropriate T_s relations in ancestor nodes. For example, if a new node n_8 is inserted in the left string of Figure III.3 after n_7, we insert T_{s_8} in the node $\langle T_{s_7}, T_{s_9}\rangle$, as shown in the right side of Figure III.3, and we revise T_{s_c}, which becomes $T_{s_7} \circ T_{s_8} \circ T_{s_9}$.

If the leaf node n for x has no non-empty cells, then we split n into two nodes n' and n'' containing two relations each. We delete from the parent the relation T_s, where s is the subsequence that corresponds to the node n, and we attempt to insert in the parent relations $T_{s'}$ and $T_{s''}$, which correspond to n' and n''. If the parent already has three relations, the deletion of T_s and the insertion of $T_{s'}$ and $T_{s''}$ will require splitting the parent into two nodes. As is the case for B-trees, this process may propagate all the way to the root and may end up creating a new root. For example, the insertion of a node n_4 following n_3 leads to splitting the node $\langle T_{s_3}, T_{s_5}, T_{s_6}\rangle$ into $\langle T_{s_3}, T_{s_4}\rangle$ and $\langle T_{s_5}, T_{s_6}\rangle$. The relation T_{s_b} is deleted and two new relations $T_{s_{b'}}$ and $T_{s_{b''}}$ are inserted into $\langle T_{s_a}, T_{s_b}, T_{s_c}\rangle$, which leads to a new split and a new root. The result tree is shown in the right side of Figure III.3. In the worst case, when an insertion in a leaf node results in splits propagating all the way to the root, we need to recompute $2\log n$ new relations (one at the leaf level, one at the new root, and $2(\log n - 1)$ at the internal nodes). Hence, the worst case complexity is $O(|Q|^2 \log |Q| \log n)$. Deletion proceeds similarly and may lead to node merging or root deletion, with the same complexity. As in the case of B-trees, the maintenance algorithm guarantees that $\mathcal{T}$ always has depth $O(\log n)$ for strings of length n. Altogether, maintenance of $\mathcal{T}$ after m updates takes time $O(m|Q|^2 \log |Q| \log n)$.

1-unambiguous regular expressions As discussed earlier, XML Schemas require regular expressions used in type definitions to be 1-unambiguous. If r is a 1-unambiguous regular expression, the corresponding DFA is of size linear in r. In this case, the relations T_s used in the above auxiliary structure have size $O(|Q|)$ rather than $O(|Q|^2)$. This brings down the size of the auxiliary structure to $O(|Q|n)$ and the complexity of maintenance and validation to $O(m|Q| \log |Q| \log n)$.

III.D Incremental DTD and XML Schema Validation

We begin this section by presenting an extension to DTDs and XML Schemas of our incremental validation algorithm for strings. Next, we study in depth a special class of regular languages, called local, that arises frequently in practice and that can be validated very efficiently. DTDs and XML schemas whose rules involve only local regular languages benefit from the efficient validation algorithm we present.

III.D.1 From strings to DTDs and XML Schemas

The incremental validation of DTDs and XML Schemas extends the divide-and-conquer algorithm for incremental validation of strings described in Section III.C. The following discussion excludes XML Schema validation for internal node renamings, which can be handled using the specialized DTD validation algorithm described in [11].

Let d be a DTD, $T = \langle t, \lambda \rangle$ a labeled tree satisfying d, and consider first updates consisting of a sequence of m label modifications yielding a new tree $T' = \langle t', \lambda' \rangle$. To check that T' satisfies d, we must verify that for each node v in t' with children $v_1 \dots v_n$ for which at least one label was modified, the sequence of labels $\lambda'(v_1) \dots \lambda'(v_n)$ belongs to $r_{\lambda'(v)}$. If the label of v has not been modified, i.e. $\lambda(v) = \lambda'(v)$, then validation can be done using the divide-and-conquer algorithm described in Section III.C for strings. However, if the label of v has been modified, so that $\lambda(v) \neq \lambda'(v)$, the sequence $\lambda'(v_1) \ldots \lambda'(v_n)$ has to be validated with respect to the new regular language $r_{\lambda'(v)}$ rather than $r_{\lambda(v)}$. Thus, it would seem that, in this case, validation has to start again from scratch. To avoid this, we preemptively maintain information about the validity of each string of siblings with respect to *all* regular languages r_a for $a \in \Sigma$. To this end we maintain some additional auxiliary information. Specifically, for each sequence s of siblings in the tree, we compute the transitions relations T_s of the divide-and-conquer algorithm described in Section III.C, for each NFA N_a corresponding to r_a, and $a \in \Sigma$. We denote the sets T_s for a particular $a \in \Sigma$ by T_s^a.[4] Since the auxiliary structure for each fixed NFA and string of length n has size $O(|Q|^2 n)$ (where Q is the set of states of the NFA), the size of the new auxiliary structure is at most $O(|\Sigma||d|^2|T|)$, where $|T|$ is the size of T and $|d| = max\{|r_a| \mid a \rightarrow r_a \in d\}$. The maintenance of the auxiliary structure is done in the same way as in the string case, at a cost of $O(m|\Sigma||d|^2 \log |d| \log |T|)$ for a sequence of m modifications. Finally, the updated tree T' is valid wrt d if for each node v with label a in T' such that either v or one of its children has been updated, $\langle q_0, f\rangle$ is in the relation T_s^a where s is the list of children of v, q_0 is the start state of N_a, and f is one of its final states. Each such test takes $O(|d|^2 \log |d|)$ and the number of tests is m in the worst case. This yields a total validation time of $O(m|\Sigma||d|^2 \log |d| \log |T|)$.

For the efficient incremental XML Schema validation of renamings of leaf nodes we maintain for each node of the tree its type; recall that the type of a node can be inferred from its label and the type of its parent. For each list of siblings, whose parent has type a^t, we maintain a transition relation tree for the NFA of the regular expression r_{a^t} that describes the content of nodes with type a^t. However, XML Schema validation for internal node renamings cannot be handled in the same way as DTD validation. The reason is that the renaming of a node v may change the types of all descendants of v. Indeed, it is easy to see that incremental validation of XML Schemas with respect to internal node renamings is at least

[4]Examples 14 and 15 of Section III.E illustrate the need and the remedy for a particular example.

as hard as validation of strings. This case requires a specialized DTDs validation algorithm, such as one described in [11].

Insertions and deletions are handled by a straightforward extension of the B-tree approach outlined in Section III.C, for both the DTD and XML Schema cases. In the case of XML Schemas we also compute and store the type of the inserted node. Insertion of m subtrees can be implemented by a sequence of insertions of the individual nodes of the subtrees. The data complexity of this implementation is $O(M \log |T|)$, where M is the total number of nodes of the inserted subtrees. A more efficient implementation first inserts the roots of the subtrees in $O(m \log |T|)$. Then it validates from scratch each subtree. In the case of DTDs the subtree rooted at node v must conform to a DTD that has the same rules but its root is $\lambda(v)$. In the case of XML Schemas the subtree rooted at node v must conform to an XML Schema that has the same types, rules and mapping from types to symbols but its root type is the type of v, which can be inferred from the parent of v and the label of v. The complexity of this implementation is $O(M + m \log |T|)$.

III.D.2 Local DTDs

In the remainder of this section we focus on a restricted class of DTDs that arises commonly in practice, and for which incremental validation can be done very efficiently. Specifically, these are DTDs using regular expressions for which validity of a string can be decided after an update by examining only substrings of bounded length around the position of the update. This is ensured by a property of the regular expressions called *locality*, which we define shortly. Locality turns out to be a very appealing property, since it immediately yields an incremental validation algorithm that, for all practical purposes, has constant data complexity. In addition, locality is a very common property in practice. We analyzed a set of 60 DTDs collected from OASIS and described in [20]. The DTDs contained 2141 complex regular expressions. Only 21 regular expressions in 10 DTDs were not

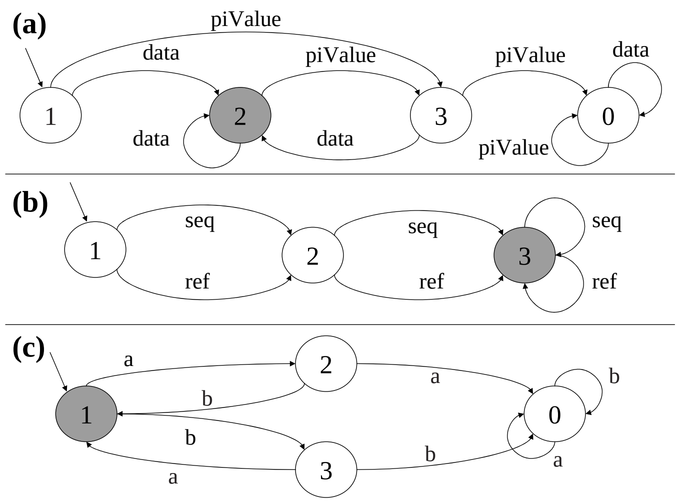


Figure III.4: Example DFAs: (a) and (b) define local languages; (c) does not.

local. We further analyzed these 21 expressions, by examining their content and contacting the authors. We determined that only 8 non-local regular expressions in 3 DTDs describe potentially large sequences of elements. The above experimental results are influenced by the fact that OASIS DTDs primarily describe message exchange information formats and relatively small files. As XML databases mature we expect that large sequences in XML documents will become more common and more non-local regular expressions will be in need of efficient incremental validation. Nevertheless, the results are indicative of how common locality is in practice.

III.D.3 Local regular languages

Before defining local regular languages, we illustrate the intuition with the following example.

Example 11 *Consider the language defined by the following regular expression, taken from the WellLogML DTD [93]* $CurveData = (data|(piValue, data))+$. *Its minimal DFA is shown in Figure III.4 (a). It has four states: state 1 is a starting state, state 2 is the only accept state (indicated by shading), and state 0 is a reject* *sink, i.e. a state from which no accepting state is reachable (note that a minimum DFA has at most one such state). Observe that the DFA has the following special property: when run on a valid string, the current state is uniquely determined by the most recent symbol. Indeed, all transitions from states other than the reject sink lead to state 2 for symbol "data", and to state 3 for symbol "piValue". Thus, when running on a valid string, the DFA is in state 2 if the last processed symbol was "data", and in state 3 if the last symbol was "piValue".*

In the minimum DFA for the regular expression in Example 11, the state is determined by the most recently read symbol (unless it is the sink state). As might be expected, some regular expressions require more than one symbol to determine the current state, as illustrated next.

Example 12 *The following regular expression is taken from DDML.DTD, available at http://www.w3.org/TR/NOTE-ddml: choice = $(seq|ref)(seq|ref)+$. This expression specifies that a "choice" element should have two or more children. This expression defines a local language. The minimal DFA is shown in Figure III.4 (b). From any given state, after a single symbol, the DFA may be in state 2 or 3. However, any two letter word always brings the DFA to state 3.*

We will call *k-local* a regular expression for which the current non-sink state is determined by the previous k symbols. Before providing the formal definition, we introduce the following notation. Given a DFA $M = \langle \Sigma, Q, q_0, F, \delta \rangle$, we denote by δ^* the natural extension of the transition function δ to $Q \times \Sigma^*$, defined by $\delta^*(q, \epsilon) = q$ and $\delta^*(q, sa) = \delta(\delta^*(q, s), a)$. The set of *potentially accepting* states of M, denoted Q^A, consists of the states in Q other than the reject sink state. We denote the reject sink state, if any, by q^{rej}.

Definition 16 (Local regular languages and DTDs) *A DFA is* k-local *if for every string s of length at least k, there are no distinct $p, q \in Q^A$, such that for some $p', q' \in Q$, $\delta^*(p', s) = p$, and $\delta^*(q', s) = q$. A regular language is* k-local *iff the* *minimum DFA accepting it is k-local. A regular language is* local *iff it is k-local for some k. The minimum k for which a regular language is k-local is called its* degree of locality*. A regular expression is (k-)local iff the language it defines is (k-)local. A DTD is (k-)local iff all regular expressions it uses are (k-)local.*

Let r be a k-local regular language. Note that, given a string w of length $\geq k$ and a potentially accepting state p, either $\delta^*(p, w) = q^{rej}$ or $\delta^*(p, w) = q$ where q is a potentially accepting state independent of p. We call a string w *rejecting* iff $\delta^*(p, w) = q^{rej}$ for every $p \in Q$. For every string w of length $\geq k$ that is not rejecting, we denote by $\delta^*(-, w)$ the unique potentially accepting state q such that $\delta^*(p, w) = q$ for some potentially accepting p.

Note that the locality of a language is defined as a property of the minimum DFA accepting the language. The following useful fact shows that locality of *any* DFA for the language is sufficient.[5]

Proposition III.D.1 *If a regular language is accepted by some k-local DFA, then it is k-local.*

Proof: Suppose a regular language r is accepted by some k-local DFA D, and let M be the minimum DFA accepting r. We show that M is also k-local. Recall that the DFA minimization algorithm produces M from D by merging equivalent states. Specifically, states p and q are equivalent if for all strings $s \in \Sigma^*$, $\delta_D^*(p, s)$ is accepting iff $\delta_D^*(q, s)$ is accepting, where δ_D is the transition function of D. The algorithm builds equivalence classes of states and replaces each class with a single state of the minimum automaton M. Thus, the minimization algorithm constructs a total mapping μ from the states of D onto the states of M that preserves transitions.

Consider a string s such that $|s| > k$. Let p, q, r, t be states in Q_M^A such that $\delta_M^*(p, s) = r$ and $\delta_M^*(q, s) = t$. There exist states p', q', r', t' in Q_D^A such

[5]In the cases where a language/DTD is non-local it may be worthwhile to consider modifying the DTD to a non-equivalent local one.

that $\mu(p') = p, \mu(q') = q, \mu(r') = r, \mu(t') = t$. Furthermore, $\delta_D(p', s) = r'$ and $\delta_D(q', s) = t'$. Since D is k-local, it follows that $r' = t'$, so $r = \mu(r') = \mu(t') = t$. Thus, M is k-local. □

Notice that Proposition III.D.1 does *not* imply that all DFAs accepting a k-local language must be k-local, or even local. For example, the language a^* is 0-local, and is accepted by a DFA with two accepting states and two a transitions from each state to the other. Clearly, this DFA is not local.

It is critical to our approach to determine, given a regular language, whether it is local, and if so to compute its degree of locality. We will show that both questions can be answered in time $O(|M|^4)$ where M is the minimum DFA for the language and the size of M, denoted by $|M|$, is $|Q| + |\Sigma| + |\delta|$. Furthermore, if M is local, then the degree of locality is bounded by $|Q|^2$ where Q is the set of states of M.

Theorem 13 *Given a regular language r and its minimum DFA $M = \langle \Sigma, Q, q_0, F, \delta \rangle$, it can be decided in $O(|M|^4)$ time whether r is local. Furthermore, if r is local, its degree of locality is at most $|Q|^2$ and can be computed in $O(|M|^4)$.*

Proof: Consider the directed labeled graph G whose vertices are pairs $\langle p, q \rangle$ where p, q are in Q^A, $p \neq q$, and there is an edge labeled $a \in \Sigma$ from $\langle p, q \rangle$ to $\langle p', q' \rangle$ iff $\delta(p, a) = p'$ and $\delta(q, a) = q'$. Note that the size of G is at most $|M|^2$. From the definition of locality, it follows that M is local iff G does not contain infinite paths. In other words, M is local iff G is acyclic.

Acyclicity of G can be tested in $O(|G|^2)$ by attempting to perform a topological sort of the vertices of the graph: first compute the in-degree of each node. Next, start with the nodes with in-degree zero, then remove their outgoing edges and decrease the in-degree count of the target nodes, and repeat. G is acyclic iff eventually all nodes have in-degree zero.

Now suppose G is acyclic, so M is local. From the definition it immediately follows that the degree of locality of M is the longest path in G, which is bounded by $|Q|^2$. This can be computed while the previous topological sort algorithm is performed, by associating a counter with each node n that stores the length of the maximum path from some initial node of in-degree zero to n at the point when n is reached. When the algorithm ends, the maximum value of the counters provides the length of the maximum path in G. Since G itself is quadratic in M, this yields an algorithm of $O(|M|^4)$.

Note that the analysis yielding $O(|G|^2)$ complexity of the above algorithm is extremely conservative, since it assumes that the complexity of finding a node v, that is the target of an edge, and its associated counter is $O(|G|)$. Assuming a data structure that allows locating a node in $O(1)$, which is a reasonable assumption in practice, the complexity is linear with respect to G. $\square$

III.D.4 Incremental validation of local DTDs

We next show how the locality property of regular languages can be used to obtain a very efficient incremental validation algorithm, of constant time data complexity. Given a DTD using only local regular expressions, if updates only affect leaf nodes without changing the labels of internal nodes, we can use a constant-time incremental validation algorithm without any additional data structure. However, if updates may also cause a renaming of a parent node the sequence of its children needs to be validated against the regular expression for the new label. As in the general incremental validation algorithm for DTDs, we therefore need to maintain some auxiliary information allowing to validate each string of siblings with respect to any of the regular expressions of the DTD, whenever the need arises. In the case of DTDs using only local regular expressions, we attach to each internal node a counter for each regular expression. Each counter's size should be at least $\lceil \log_2 n \rceil$, where n is the maximum size of a sibling list. Notice that 4-byte integers can accommodate sibling lists of up to 2^{32} elements and, hence, in our implementation we just use 4-byte integers for counters.

Our algorithm relies on the following observation.

Lemma 3 *Let r be a k-local regular expression. A string $s \in \Sigma^*$ of length $\geq k$ is valid with respect to r iff the following conditions hold:*

1. *there is no rejecting substring w of s of length $k+1$.*
2. *$\delta^*(q_0, w) \neq q^{rej}$, where w is the prefix of s of length k;*
3. *$\delta^*(-, w) \in F$, where w is the suffix of s of length k.*

Proof: Clearly, conditions (1-3) are necessary for validity. We show they are also sufficient. Suppose $s = a_1 \dots a_n$ is a string of length $\geq k$ that satisfies conditions (1)-(3). We show that

$$(\dagger) \quad \delta^*(q_0, a_1 \dots a_i) \neq q^{rej} \text{ for all } i, k \leq i \leq n.$$

Clearly, (†) proves the statement, since in conjunction with condition (3) and k-locality it implies that $\delta^*(q_0, a_1 \dots a_n) = \delta^*(-, a_{n-k+1} \dots a_n) \in F$.

We prove (†) by induction. The basis (for $i = k$) is true by condition (2). For the induction step, suppose that $i \geq k$ and $\delta^*(q_0, a_1 \dots a_i) \neq q^{rej}$. Consider $a_1 \dots a_{i+1}$. By the induction hypothesis and k-locality, $\delta^*(q_0, a_1 \dots a_{i+1}) = \delta(\delta^*(-, a_{i-k+1} \dots a_i), a_{i+1})$. By condition (1), $a_{i-k+1} \dots a_{i+1}$ is not rejecting (since it has length $k+1$). Therefore, there exist potentially accepting states p, p' such that $\delta^*(p, a_{i-k+1} \dots a_{i+1}) = p'$. By k-locality,

$$\delta^*(p, a_{i-k+1} \dots a_{i+1}) = \delta(\delta^*(-, a_{i-k+1} \dots a_i), a_{i+1}).$$

It follows that

$$\delta^*(q_0, a_1 \dots a_{i+1}) = \delta(\delta^*(-, a_{i-k+1} \dots a_i), a_{i+1}) = p' \neq q^{rej}.$$

This completes the induction and the proof of (†). □

Lemma 3 suggests the following validation algorithm for the sequence of siblings in an XML document. For each regular expression r of the DTD, determine its degree k of locality. As long as the string remains of length at most k, we validate the string with respect to r from scratch if needed. When the string exceeds length k (if ever), we check conditions (1)-(3) of the lemma. Conditions (2) and (3) are easy to check from scratch in constant time whenever required. Condition (1) is checked incrementally by maintaining some auxiliary information consisting in the count of the number of rejecting substrings of length $k+1$ in the current string. To do so, we first initialize the count in a pre-processing step that takes linear time in the size of the string (we do so as soon as the length of the string exceeds k). A single update occuring at position i affects the substrings of length $k+1$ containing i, whose number is constant. For each such affected substring we determine whether or not it becomes (or ceases to be) rejecting, and update the count accordingly. Whenever validation with respect to r is needed, we check that the count of rejecting substrings is zero. For multiple updates, we maintain the counters one update at a time. Thus, the required auxiliary structure consists, for each internal node in the XML tree, of one counter for each regular expression in the DTD. For a tree T with i internal nodes, the size of the auxiliary structure is $O(i \log(|T|/i))$, neglecting the fact that in practice lists will be smaller than 2^{32}, which can be accomodated by a typical 4-byte counter. Maintaining incrementally the auxiliary structure takes $O(m \log |T|)$ time with respect to the string for m updates. Although its theoretical worst-case complexity is no better than the general DTD validation algorithm, the incremental validation algorithm has a very efficient implementation, which practically provides constant time validation. First, the auxiliary structure consists of counters. Assuming that no list of siblings is longer than 2^{32} elements, then each counter can be a usual 32-bit integer. The counters are stored together with the data tree (internal nodes), hence simplifying their access, and their maintenance consists of simply incrementing or decrementing them, which in practice takes constant time.

Notice that, if updates do not involve the renaming of internal nodes, then we do not need to maintain counters. Instead, the number of rejecting substrings is zero before the transaction starts and we check that it remains zero after the transaction has ended. This requires that the updates generate no rejecting substring, the prefix of the sequence was not rejecting when starting at q_0, and the suffix of the sequence leads to an accepting state.

Furthermore, we do not need to maintain counters if every pair (r, r') of regular languages used by the DTD is *incompatible*, in the sense that it is impossible to turn a string that belongs to $L(r)$ into a string that belongs to $L(r')$, without a transaction performing a number of updates that is in the order of the size of the resulting string - hence, making revalidation of r' from scratch equally attractive to incremental validation. Indeed, whenever two regular expressions r and r' do not contain any common symbol a within the scope of a * they are incompatible. Notice that the reverse is not always true.

Even if we cannot fully eliminate counters, we may still be able to reduce the number of counters we have to maintain for each sibling list. Consider a sibling list whose parent is labeled a and the list belongs to $L(r_a)$. Consider also a label b and the corresponding regular expression r_b, which is incompatible with r_a. We do not need to maintain a counter for r_b since a renaming of a to b needs to be accompanied by a large number of updates on the sibling list and it is not beneficial to incrementally validate such a number of updates.

III.E Implementation

We implemented incremental and local validation algorithms on the XCacheDB XML database [9] which is built on top of a commercial relational database engine. The XCacheDB gives an administrator control over the decomposition of XML data into relations of the underlying RDBMS. For the purposes of this study we decomposed the XML data into normalized relational schemas. The following description applies exclusively to the normalized approach, which is similar to the “hybrid inlining” of [80]. [9] provides a description of alternative approaches.

The XCacheDB creates a table for every *repeatable* element in the DTD.

We say that an element is repeatable if it can occur an unlimited number of times in a sibling list. A table contains zero or one data attributes, one system-generated node ID attribute, and at least one of `prnt`, `rsib`, and `tid`. A data attribute is created for every element with string content. We encode the tags of elements in the elements' IDs to efficiently identify the table that needs to be accessed given an element's ID. To preserve the parent-child relationship, each table includes a parent reference `prnt`, with the exception of the table that stores the root element of the DTD. We extended XCacheDB to preserve the document order of XML elements, by adding the `rsib` attribute to every relation in the schema. This attribute stores the ID of the element's right sibling. Since the ID encodes the tag, the value of `rsib` also determines the table in which the sibling is stored.

Every element whose parent requires validation[6] also stores a `tid` attribute. The `tid` is a foreign key into the transtion relation storage table, which we will describe shortly, and stands for the pointers from elements to transition relation tree leafs.

We illustrate the XCacheDB data storage and auxiliary storage used for incremental validation with an example, which will be the running example of this section.

Example 14 *Consider the following DTD:*
$r = (r1|r2)*$
$r1 = (b|ab)+$
$r2 = (b|abb)+$
where a *and* b *are elements with string content. The DTD is based on the Well-LogML DTD [93], which contains the expression*
$CurveData = (data|(piValue, data))+$. *To illustrate the validation of the renaming of intermediate nodes, we added the* $r2$ *element to the DTD. The minimal DFAs of the expressions* $r1$ *and* $r2$ *are shown in Figure III.5 (c). Figure III.5 (a) shows the relational schema created by the XCacheDB. The tables* `R`*,* `R1`*,* `R2`*,* `A` *and*

[6]We do not validate trivial regular expressions, with minimal DFAs containing a single state.

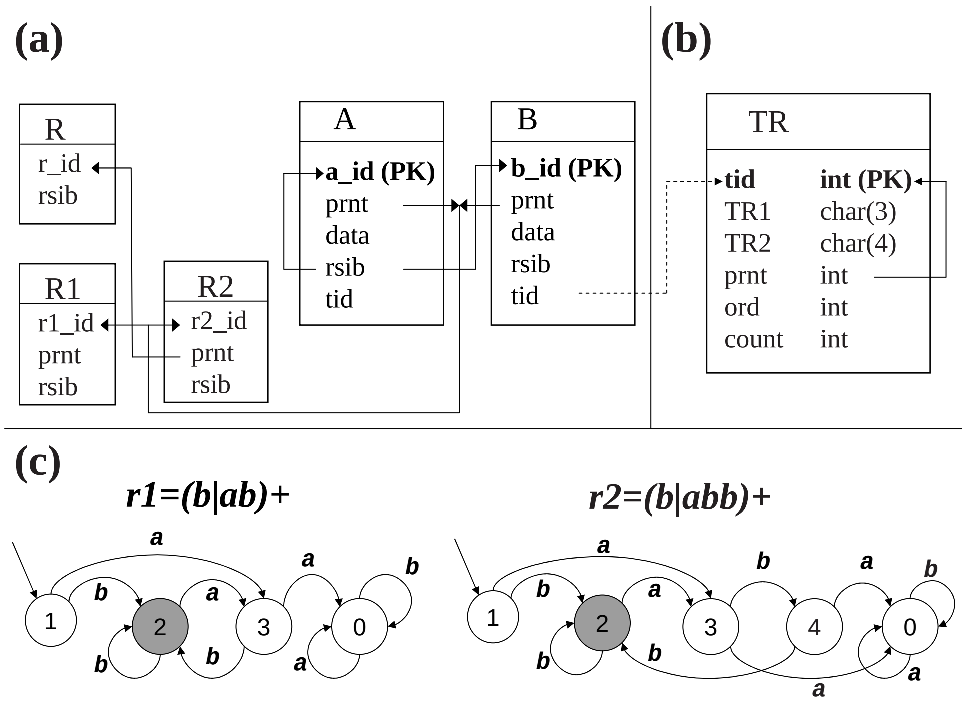
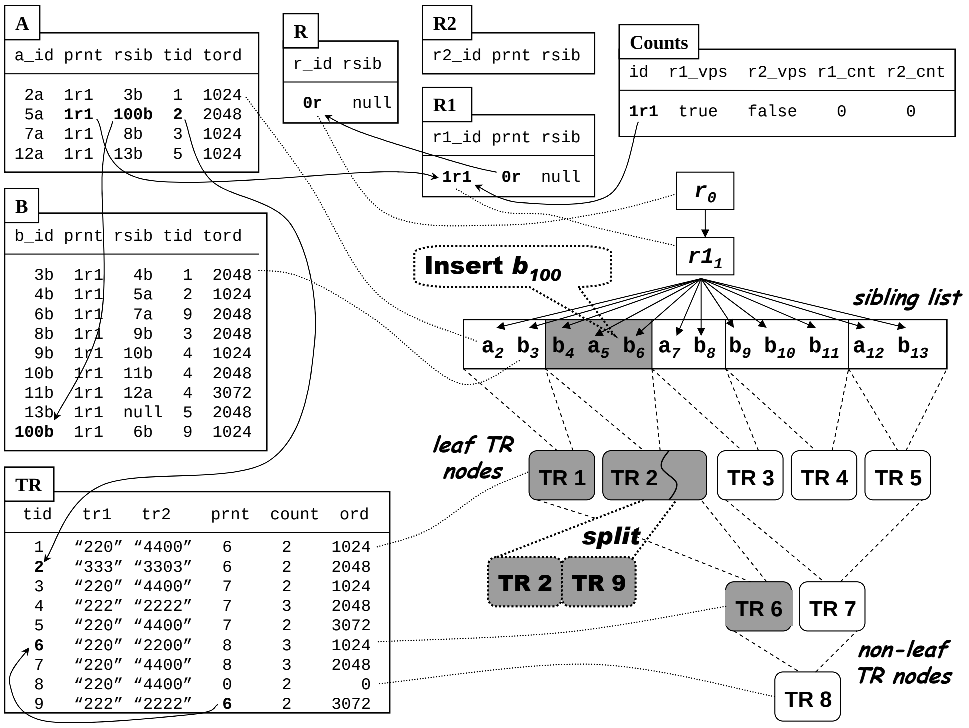


Figure III.5: Relational schema of XCacheDB and TR storage.

`B` *hold the data of the XML document. The string content of elements a and b is stored in the* `data` *attributes of tables* `A` *and* `B` *respectively. Tables* `A` *and* `B` *include parent references to* $r1$ *or* $r2$ *elements, since both* $r1$ *and* $r2$ *can have a and b as their children. For example,* `A.prnt` *references either* `R1.r1_id` *or* `R2.r2_id`*. Recall that the element IDs encode the tags of elements. In this example, the last two bits of the element ID determine whether the element is a, b,* $r1$*, or* $r2$*. Thus by looking at the value of the parent attribute of an element we immediately know whether it references* $r1$ *or* $r2$*. In Figure III.6 we explicitly show the tags as part of the elements' ID for clarity of exposition. For example,* $3b$ *is the ID of the* b_3 *element, and* $1r1$ *is the ID of* $r1_1$*.*

We implemented the three basic update operations, insert, delete and rename, as well as three validation algorithms, incremental (Section III.D), local (Section III.D.4), and a naive, which involves reading the whole sibling list and re-validating it from scratch.



A

a_id	prnt	rsib	tid	tord
2a	1r1	3b	1	1024
5a	**1r1**	**100b**	**2**	2048
7a	1r1	8b	3	1024
12a	1r1	13b	5	1024

B

b_id	prnt	rsib	tid	tord
3b	1r1	4b	1	2048
4b	1r1	5a	2	1024
6b	1r1	7a	9	2048
8b	1r1	9b	3	2048
9b	1r1	10b	4	1024
10b	1r1	11b	4	2048
11b	1r1	12a	4	3072
13b	1r1	null	5	2048
100b	1r1	6b	9	1024

R

r_id	rsib
0r	null

R2

r2_id	prnt	rsib

R1

r1_id	prnt	rsib
1r1	**0r**	null

Counts

id	r1_vps	r2_vps	r1_cnt	r2_cnt
1r1	true	false	0	0

TR

tid	tr1	tr2	prnt	count	ord
1	"220"	"4400"	6	2	1024
2	"333"	"3303"	6	2	2048
3	"220"	"4400"	7	2	1024
4	"222"	"2222"	7	3	2048
5	"220"	"4400"	7	2	3072
6	"220"	"2200"	8	3	1024
7	"220"	"4400"	8	3	2048
8	"220"	"4400"	0	2	0
9	"222"	"2222"	**6**	2	3072

Figure III.6: The state of the storage after a new b element is inserted.

Incremental Validation The transition relations (TR) trees required by the incremental update validation algorithm are stored in the TR relation, as illustrated in Figure III.5 (b). We store m TR nodes per tuple, where m is the number of regular expressions that the string has to be validated against. The transition relations that correspond to the same sequence of nodes with respect to different regular expressions are always accessed simultaneously, and it is advantageous to store them together. For instance, the DTD of Example 14 contains two regular expressions $r1$ and $r2$ that we need to be able to validate. Thus, the TR table for this DTD will have two columns `tr1` and `tr2` as shown in Figure III.6. We do not need to validate trivial regular expressions, such as r, with minimal DFAs containing a single state.

Each transition relation is stored as a string of length n, where n is the number of states in the minimal DFA of the regular expression that is being validated. We assume that the minimal DFA has at most 256 states and, hence, a byte is enough to represent a state. The reject sink is by convention the state 0, and its transitions are not stored in the database. For example, the minimal DFAs for $r1$ and $r2$ are shown in Figure III.5 (c). These DFAs have three and four potentially accepting states respectively. Hence, `tr1` is a string of size 3, and `tr2` has length 4.

Each tuple of the `TR` relation has a unique `tid` attribute, and a `prnt` attribute that references the parent TR node. The attribute `count` stores the number of children TR nodes, and `ord` is used for ordering sibling TR nodes, which map to the same parent. When assigning the `ord` numbers, we leave enough space to accommodate many updates without renumbering. Notice, that renumbering does not entail much overhead since the `ord` numbers have to be unique only among the children of the same TR node.

We create an index on the pair (`prnt`,`ord`), to facilitate efficient validation and TR splits. These operations require access to a list of sibling TR nodes.

Example 15 *Consider the XML fragment of Figure III.6 that is valid with respect to the DTD of Example 14. Note that the* `TR` *table maintains transition relations that validate the list of siblings both against* $r1$*'s content definition (those relations are stored in* `tr1`*) and transition relations that validate against* $r2$*'s content definition (those relations are stored in* `r2`*) despite the fact the parent of the list of siblings is not* $r2$*. The reason is that, as we explained in Section III.D, we need to be able to validate a renaming of* $r1$ *to* $r2$ *without validating the sequence of siblings from scratch. Assume that the TR tree is constructed for the list of a's and b's as shown in Figure III.6. Assume that the maximum TR tree node size is 3 (nodes cannot have more than 3 children).*

The first tuple of the TR relation in Figure III.6, corresponds to the TR1 node, which covers the substring a_2b_3*. If we run this string on the* $r1$ *DFA, the automaton will terminate in states* 2, 2, *and* 0 *if it was initialized at states* 1, 2, *and* 3 *respectively. Thus, the transition relation* `TR.tr1` *for this substring is encoded by "220".*

Now consider an insertion of a new b element with $id = 100$ between the fourth and fifth child of $r1$. Figure III.6 shows the state of the database after the insertion. Shading indicates elements and transition relations that were accessed to validate the insertion. Notice that a new TR node ("TR 9") has to be created, since "TR 2" cannot have 4 children elements, due to the maximum TR node size assumption.

Local Validation The table `Counts` stores counters used for validation of complex node renames by the local validation algorithm described in Section III.D.4. For each complex element, i.e., for each internal node of the data tree, we store a tuple that contains the element's ID and a pair of `vps` and `cnt` attributes for each regular expression that needs to be validated. The `vps` attribute is a boolean "valid prefix/suffix" flag which indicates whether conditions (2) and (3) of Lemma 3 are satisfied for the element's children list. The `cnt` attribute stores the number of rejecting sequences of length $k + 1$ in the list. The condition (1) of Lemma 3 is satisfied if this number is 0.

Example 16 *Figure III.6 shows a* `Counts` *tuple that corresponds to $r1_1$.*
The list of the element's children is valid with respect to expression $r1$. Indeed, `r1_vps`=true *and* `r1_cnt`=0*. However, this element cannot be renamed to $r2$, since the same list is not valid with respect to expression $r2$. Even though the sequence does not contain any rejecting substring (*`r2_cnt`=0*), the last two elements leave the minimal DFA of $r2$ in state 4, which is not accepting (see Figure III.5 (c)). This breaks condition (3) of the lemma, hence* `r2_vps`=false*.*

To facilitate efficient access to the element's left sibling, which is required by the local validation algorithm, we create indices on `A.rsib` and `B.rsib`.

III.F Applicability and Performance Experiments

Local validation was applicable on the majority of 60 real-world DTDs found at OASIS and described at [20]. In particular, there were only 21 non-local regular expressions in 10 DTDs; the total number of regular expressions was 2141. By investigating the documentation of the DTDs and contacting their authors we determined that 8 of those expressions describe potentially very large lists of data. In addition, the minimal k, which affects the performance of the local validation algorithm was always less than 4. In total, 2070 expressions were 1-local, 23 expressions were 2-local but not 1-local, and 27 expressions were 3-local but not 2-local. 39 DTDs contained only 1-local expressions, 8 DTDs contained only 2-local expressions (recall, the set of 2-local expressions includes the set of 1-local expressions) and 3 DTDs contained only 3-local expressions; recall, 10 DTDs contained non-local expressions.

Next we experimentally compare the performance of three update validation algorithms: local validation, described in Section III.D.2, general incremental validation described in Section III.D, and a naive algorithm that involves reading whole sibling lists and re-validating them from scratch. We simulated a number of update scenarios and analyzed the running time and space overhead of the algorithms under various parameters. We have considered a case where the database objects are perfectly clustered and a case where a fraction of the database objects is not placed at the optimal clusters, which is a typical assumption for a database that is being updated.

III.F.1 Experiment Setup

All the experiments were performed on a 1.2 GHz Pentium system with 512 MB of memory and 5400 rpm hard drive. The XCacheDB server and the RDBMS were installed on the same system. To offset the relatively small dataset size, the RDBMS was configured to use only 16 MB for the buffer cache.

We used a synthetic XML dataset containing about 150000 elements. The dataset conformed to the DTD of Example 14, which is similar to the WellLogML DTD [93]. The *a* and *b* elements have text content, which does not alter the validation algorithms, but its size affects the performance of the implementation since it increases the size of the dataset. Notice that this DTD is local, as the *r1* expression is 1-local, and *r2* is 2-local. Since the DTD is local, we use the same dataset to compare all three algorithms: local, incremental, and naive.

Scenarios and Parameters We controlled the following parameters of the dataset, and of the Transition Relation (TR) storage.

1. Sibling list size: Each dataset had 150000 leaf elements, evenly distributed between 10, 100, 1000, or 10000 top-level ($r1$ or $r2$) elements. Thus, the average length of a sibling list that had to be validated was 15000, 1500, 150, and 15 respectively. This is the number of elements that has to be accessed by a naive algorithm to validate each update.

2. Size of the `data` attributes of relations `A` and `B`. These attributes store text content for the leaf *a* and *b* elements. We tried sizes 25, 100, and 400 bytes, which translated to 8MB, 20MB and 65MB relational database sizes respectively.

3. Transition Relation (TR) Tree Leaf Node Size: This size refers to the maximum number of XML elements that can point to the same transition relation tree node. For every update the incremental algorithm has to read the data elements belonging to the same leaf TR node. For this parameter we tested values of: 4, 16, 64, and 256.

4. TR Tree Non-Leaf Node Size: This size refers to the maximum number of children an internal TR tree node is allowed to have. For this parameter we tested values of: 4, 16, 64, 256 and 1024.

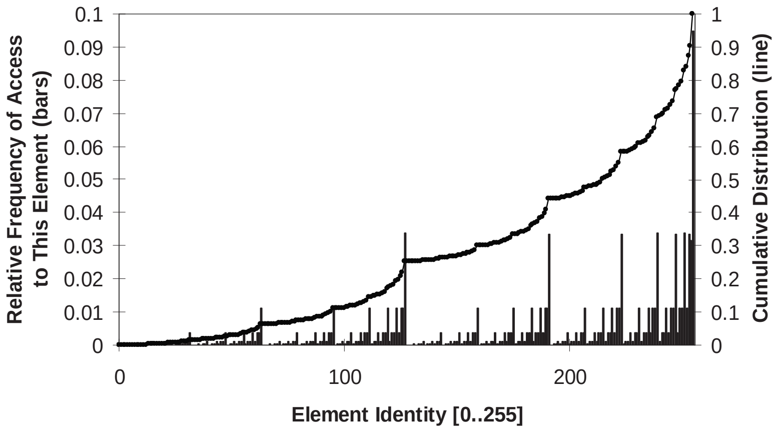


Figure III.7: Relative frequency and cumulative distribution of NURand() function, used in TPC-C.

We tested the performance of element renames and three insert scenarios: uniformly distributed insertions, insertions distributed according to the TPC-C benchmark, and append-only access.

Unless otherwise specified, the experiments were conducted with the transition relation tree nodes being 75% full. For example, when the TR leaf node size was 4 and the non-leaf node size was 64, a leaf TR node would be created for every 3 XML elements and a non-leaf TR node would be created for every 48 sibling TR nodes.

In the random insertion scenario, 10000 *b* elements were inserted at uniformly distributed random points in the document. All insertions were done as a part of a single transaction. The incremental algorithm had to maintain the data structure for each insertion. Notice that all our transactions are valid to avoid an (orthogonal) issue of transaction roll-backs. Since the DTD is local, incremental validation requires reading only a single TR tree node, unless the insertion happens at distance less than *k* from the end of the sub-list of elements that point to the same transition relation tree leaf node.

In the "average case" insertion scenario, which follows TPC-C, the same 10000 *b* elements were inserted at points picked by the NURand() non-uniform random function. The cumulative distribution of this function, for the case when one element is picked out of a list of 256 elements, is shown in Figure III.7 (taken from [53]). The same random function is used to construct update transactions in the TPC-C benchmark [84].

In the "append-only" scenario the dataset was constructed in the same way. However, the 10000 b elements were all appended to one of the r lists. Notice, that this scenario caused the maximum number of node splits, and, at the same time, took maximum advantage of database caching.

We performed the experiments on both clustered and unclustered datasets. In the first case, each list of siblings was stored consecutively in the tables A and B, hence leading to perfect clustering of the data. In the second case, 15% of randomly picked records were shuffled around by deleting and reinserting them back into the table. The second case models a databases that has been updated up to 15%, without having been reorganized for clustering purposes yet. Without the perfect clustering the naive approach was at even greater disadvantage, as it requires accessing full sibling lists for each update.

Optimizing the TR parameters First, we experimented with the TR node size parameters to maximize the performance of the incremental algorithm. Figure III.8 shows the average insertion time, which includes actual insertion and incremental validation time, for the uniform, average and append scenarios described above. The last graph corresponds to the "full nodes" scenario, which is a random insertion scenario modified so that all nodes were initially created 100% full. Thus any insertion will trigger one or more node splits. This is essentially a worst-case scenario for the incremental validation algorithm, as it requires the highest number of node splits and the database server cannot take full advantage of caching, due to the random locations of the updates.

All four graphs of Figure III.8 exhibit the tradeoff between large number of node splits needed for smaller node size and large sibling lists that need to be

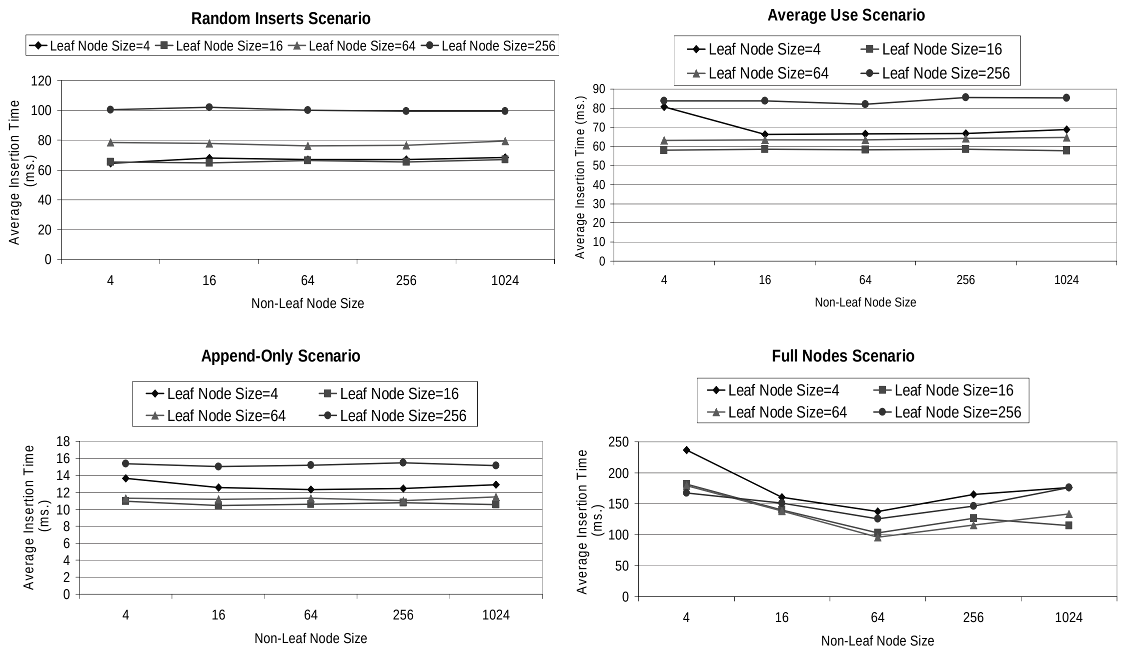


Figure III.8: Effects of the node size parameters under different update scenarios.

read to validate for larger node size. In all four scenarios, leaf node size of 16 performed better than very small and very large values.

The TR leaf node size affects the performance more than the size of the internal TR nodes, since leaf TR tree nodes need to be accessed for every insertion, while internal ones are accessed much less frequently[7]. The sibling list size and data attribute size for these experiments were fixed at 15000 and 100 respectively and the dataset was perfectly clustered. We don't include the graphs for different sibling list sizes and unclustered data, since they are very similar with the ones of Figure III.8.

The effects of the size of the `data` attribute on the "average-case" scenario are shown in Figure III.9. The non-leaf node size does not significantly affect this experiment and was fixed at 64. We report only validation time, as the insertion time, naturally, increases with the `data` size and creates the same offset for all methods.

[7]Since our DTD is locally updateable, non-leaf TRs are accessed only when the leaf node is split, or when the updated element happens to be within k of the end of sibling sub-list that maps to a particular TR leaf node.

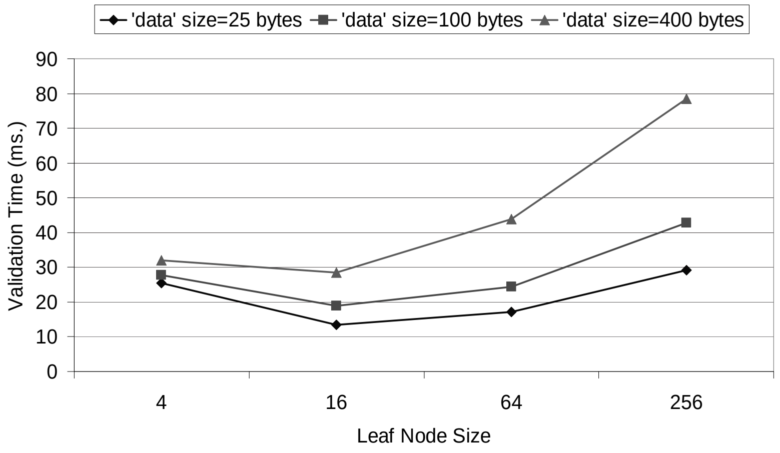


Figure III.9: Effects of the "data" attribute size.

Notice that in all three cases the leaf size 16 performed the best. However, with 25 byte `data` attributes, leaf size 64 performed better than leaf size 4, while with 400 byte `data` attributes the opposite can be observed. The reason is that with larger column size fewer tuples fit on a database page. Thus there is higher chance that an update validation will require access to multiple pages. This increases the negative effects of larger leaf node sizes.

In the rest of the experiments reported in this section we fix leaf and non-leaf node sizes to be 16 and 64 respectively, since these values consistently provide good performance.

Comparing the algorithms Figure III.10 shows that the local and incremental update validation algorithms are virtually insensitive to the sibling list size, while the naive algorithm scales almost linearly. Notice that the graphs are in logarithmic scale. The `data` attribute size was fixed at 100.

The local validation algorithm is a winner even when the updated element has as few as 15 siblings. This is remarkable considering how much more efficient the local validation implementation could have been if we had lower level access to the data. With small sibling list size, performance of the incremental validation is comparable to that of the naive algorithm. As the sibling list size grows, the

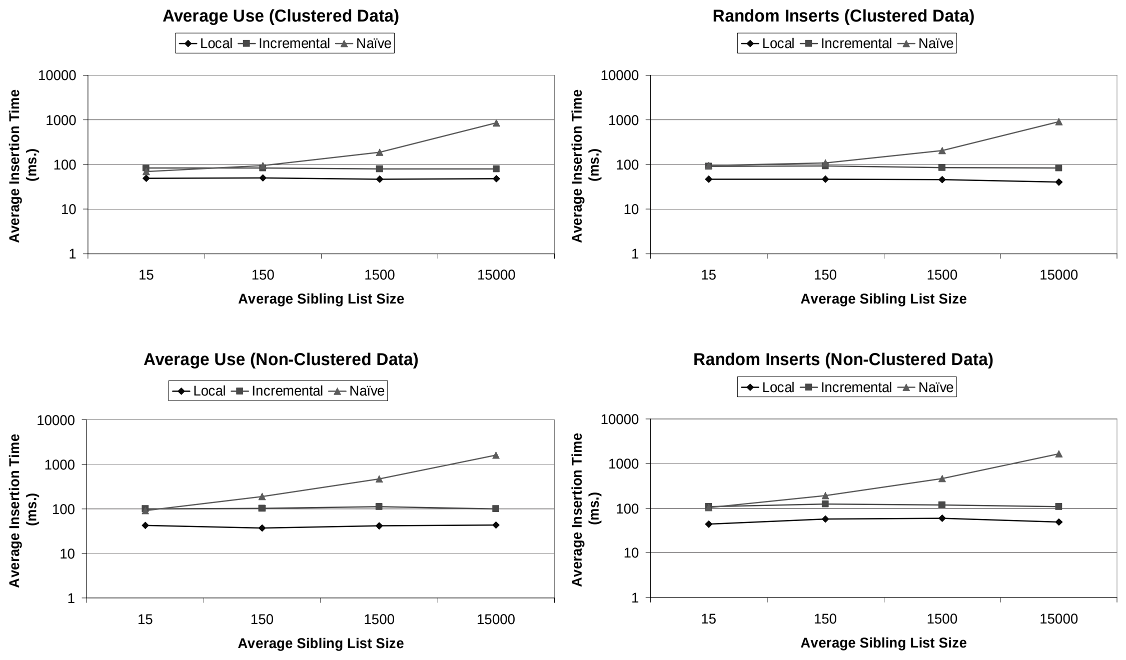


Figure III.10: Performance of the three validation algorithm under different update scenarios on clustered and non-clustered datasets.

incremental validation outperforms the naive by more than an order of magnitude (sibling list of length 15000).

Naturally, all three algorithms perform worse on non-clustered data, as they cannot take full advantage of caching and prefetching done by the database server. However, the naive algorithm's performance is impacted more since it has to access more data per update.

Figure III.11 shows effects of the `data` attribute size on all three validation algorithms in the "average-use" case. The sibling list size was fixed at 15000. Once again, the graph is in logarithmic scale. With all three algorithms the validation for larger `data` size takes longer as it requires reading more data. Notice that with small `data` size, clustering almost does not impact the performance of the naive algorithm. In this case the entire database fits in the buffer space of the RDBMS. However, when the `data` attribute size is large and the database is 4 times bigger than the buffer space, the naive algorithm slows down by almost an

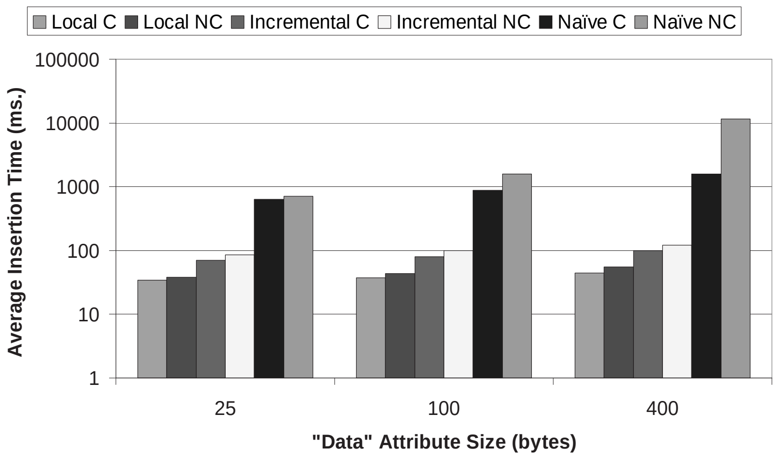


Figure III.11: Effects of the `data` attribute size on the three validation algorithms, on clustered and non-clustered datasets, in “average-use” case.

order of magnitude when running on non-clustered data. In this case the data has to be read from disk, and without the clustering, the algorithm cannot take full advantage of the prefetching.

Construction Overhead of Transition Relations Table III.1 shows time and space required to initially construct transition relations, with various node sizes, for datasets with average sibling list sizes 1500, 150, and 15. Notice that larger TR node sizes do not incur larger overhead. Local validation algorithm also has space overhead as it stores an integer counter for each complex element and each regular expression that the element is validated against. In our experiments this overhead ranged from 80 bytes (10 internal nodes) to 80 KB (10000 internal nodes).

Summary We compared naive (”from scratch”) update validation with the described in Section III.E implementation of general incremental validation and local validation, in the context of an XML database built on a relational database. The following key results and guidelines emerged. First, local validation should always be used, when applicable, since it outperforms naive and general incremental validation in all scenarios. Fortunately, our investigation in real-life DTDs posted

Table III.1: Overhead of Transition Relations

node size	*fan-out*	*time (sec)*	*size (KB)*
4	15000	194	1875
16	15000	40	341
64	15000	12.5	80
256	15000	6.6	20
1024	15000	5.2	5.3
4	1500	198	1886
16	1500	59	344
64	1500	26	82
256	1500	19.8	23
1024	1500	19	7.5
4	150	282	1912
16	150	128	395
64	150	103	126
256	150	92	25
1024	150	94	25
4	15	1068	2027
16	15	1051	685
64	15	881	244
256	15	881	244
1024	15	872	244

at OASIS showed that local DTDs are very common. Second, the tuning of the leaf node size parameters of the auxiliary structure needed for general incremental validation is relatively easy, since values in the 16-64 range consistently provided good results. General incremental validation marginally outperforms naive validation for sibling list sizes around 150 and significantly outperforms naive validation for sibling list sizes in the thousands and beyond. The relative comparison results were not significantly sensitive to how the data were clustered and the pattern of the sequence of updates.

Chapter IV

XKeyword: Keyword Proximity Search In Semistructured Databases

IV.A Introduction

XML and its labeled graph abstraction emerge as the data model of choice for representing semistructured self-describing data. Novel query languages (see [2] for a survey and [101] for the emerging XQuery standard) provide features, such as flexible path expressions, that allow one to query semistructured data, i.e., graph data that are not characterized by rigid structure. However, one still needs sufficient knowledge of the structure, role of the requested objects and XQuery in order to formulate a meaningful query. Keyword search does not present such requirements; it enables information discovery by providing a simple interface. It has been the most popular information discovery method since the user does not need to know either a query language or the structure of the underlying data.

Typically the search engines enable keyword search on top of sets of documents. Given a set of keywords, the search engine returns all documents that are associated with these keywords. Usually, a set of keywords and a document are associated if the keywords are contained in the document. Their degree of associativity is often their distance from each other.

XKeyword follows a recent generation of information retrieval systems that provide keyword proximity search [39, 42, 16, 3] to structured and semistructured databases. In particular, XKeyword provides keyword proximity search on XML data that are modeled as labeled graphs, where the edges correspond to the element-subelement relationship and to `IDREF` pointers. XKeyword differs from prior systems for proximity search on labeled graphs in that it assumes the existence of a schema, similar to the XML Schema standard [91], to which the graph conforms. The schema facilitates the presentation of the results and is also used in optimizing the performance of the system. Note that the end-user does not need to be aware of the schema.

A *keyword proximity query* is a set of keywords and the results are trees of XML fragments (called *target objects*) that contain all the keywords and are ranked according to their size. Trees of smaller sizes denote higher association between the keywords, which is generally true for reasonable schema designs. For example, consider the keyword query "John, VCR" on the graph of Figure I.2. The first highlighted tree (thick edges) $name[John] \leftarrow person \leftarrow supplier \leftarrow lineitem \rightarrow linepart \rightarrow product \rightarrow descr[set\ of\ VCR\ and\ DVD]$ on the source XML graph of Figure I.2 is a result of size 6. The second highlighted tree (gray arrows) $name[John] \leftarrow person \leftarrow supplier \leftarrow lineitem \rightarrow linepart \rightarrow part \rightarrow subpart \rightarrow part \rightarrow name[VCR]$ is a result of size 8. The first result is considered to be a "better" one by XKeyword (as well as by all the other keyword proximity search systems) since the shorter distance corresponds to the closer connection between "John" and "VCR" in the first solution, where the "VCR" is the product that "John" supplied, as opposed to being a sub-part of another part supplied by "John". Notice that we allow edges to be followed in either direction.

As we mentioned in Section I.C, the presentation of the results faces two key challenges that have not been addressed by prior systems. First, the results need to be semantically meaningful to the user. Towards this direction, XKeyword associates a minimal piece of information, called *target object*, to each node and displays the target objects instead of the nodes in the results. In the DBLP demo (Figure IV.2) XKeyword displays target object fields such as the paper title and conference along with a paper.

The second challenge is to avoid overwhelming the user with a huge number of often trivial results, as is the case with DISCOVER [42] and DBXplorer [3][1]. Both of those systems present all trees that connect the keywords. In doing so they produce a large number of trees that contain the same pieces of information many times. For example, consider the keyword query "US, VCR" and the subgraph of the XML graph of Figure I.2 shown in Figure I.4. This XML fragment contains four results:

$$\begin{array}{ll} N_1 : p_1 \leftarrow l_1 \rightarrow pa_3 \rightarrow pa_1, & N_2 : p_1 \leftarrow l_2 \rightarrow pa_3 \rightarrow pa_2, \\ N_3 : p_1 \leftarrow l_2 \rightarrow pa_3 \rightarrow pa_1, & N_4 : p_1 \leftarrow l_1 \rightarrow pa_3 \rightarrow pa_2 \end{array}$$

The above results contain a form of redundancy similar to multivalued dependencies [36]: we can infer N_3 and N_4 from N_1 and N_2. In that sense, N_3 and N_4 are trivial, once N_1 and N_2 are given. Such trivial results penalize performance and overwhelm the user. XKeyword avoids producing "duplicate" results and uses a *presentation graph* that comprises the complete set of nodes participating in result trees. At any point only a subset of the graph is shown (see Figure IV.1), as it is formulated by various navigation actions of the user. Initially the user sees one result tree r_0. By clicking on a node of interest the graph is expanded to display more nodes of the same type that belong to result trees that contain as many as possible of the other nodes of r_0. Towards this purpose we define a minimal expansion concept. For example, clicking on the *lineitem* node of Figure IV.1 (a) displays all *lineitem* nodes which are connected to the *person* and *part* in the initial tree, as shown in Figure IV.1 (b).

Two key challenges arise on the way to providing fast response times.

[1]Both systems work on relational databases, but the presentation challenges are similar.

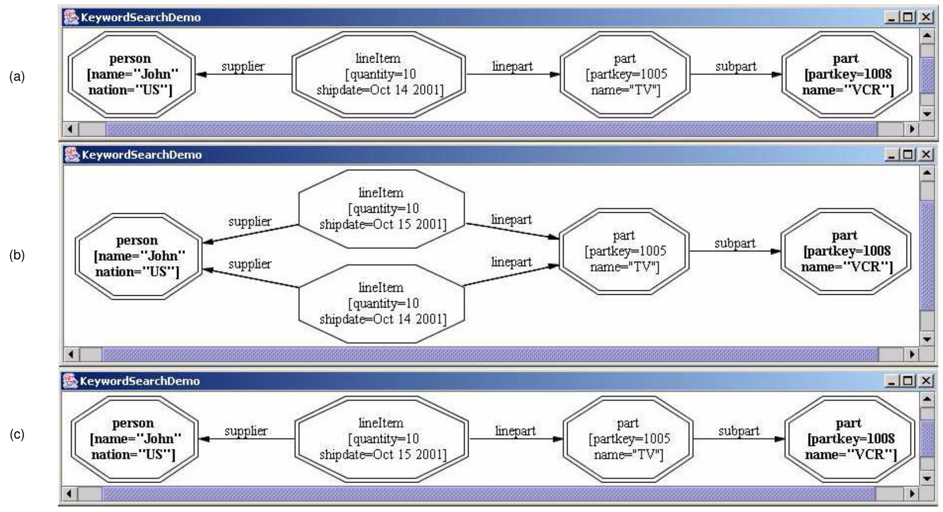

(a)
KeywordSearchDemo
person [name="John" nation="US"]
supplier
lineItem [quantity=10 shipdate=Oct 14 2001]
linepart
part [partkey=1005 name="TV"]
subpart
part [partkey=1008 name="VCR"]
(b)
KeywordSearchDemo
person [name="John" nation="US"]
supplier
lineItem [quantity=10 shipdate=Oct 15 2001]
supplier
lineItem [quantity=10 shipdate=Oct 14 2001]
linepart
linepart
part [partkey=1005 name="TV"]
subpart
part [partkey=1008 name="VCR"]
(c)
KeywordSearchDemo
person [name="John" nation="US"]
supplier
lineItem [quantity=10 shipdate=Oct 15 2001]
linepart
part [partkey=1005 name="TV"]
subpart
part [partkey=1008 name="VCR"]

Figure IV.1: Presentation graph (expanded nodes have single outliner)

First, the XML data has to be stored efficiently to allow the fast discovery of connections between the elements that contain the keywords. We follow the architecture of multiple recent XML database systems and store the XML data in a relational database [17, 80, 33, 58, 25, 75, 9], which we tune to provide the needed indexing and clustering. Then XKeyword builds a set of *connection relations*, which precompute particular path and tree connections on the TSS graph. Connection relations are similar to path indices [31] since they facilitate fast traversal of the database, but also different because they can connect more than two objects and they store the actual path between a set of target objects, which is needed in the answer of the keyword query. A core problem is the choice of the set of connection relations that are precomputed.

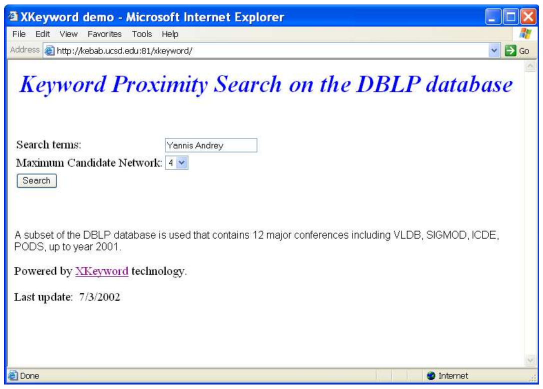
Second, the cost of computing the full presentation graph is very high. Hence XKeyword uses an on-demand execution method, where the execution is guided according to the user's navigation. We present an algorithm that generates a minimal set of queries to the underlying database in response to the user's navigation.

XKeyword consists of two stages (Figure IV.5). In the preprocessing stage, the *master index* is created along with a set of connection relations. The master index is an inverted index that stores for each keyword k a list of elements that contain k. The most suitable *decomposition*, i.e., representation of the target object graph with a set of connection relations, is selected, given the performance and space requirements. We compared different decomposition strategies and found that in order to compute the top-1 result for each result schema, which is needed to construct the presentation graph, the most space effective decomposition is to create inlined fragments [9], i.e., fragments that do not contain multivalued dependencies. On the other hand, a combination of the inlined and the minimal decomposition, where a connection relation is generated for each edge of the schema graph, is more efficient for the on-demand expansion of the presentation graph.

In the query processing stage, XKeyword retrieves from the master index the schema nodes, whose elements contain the keywords, and exploits the schema graph's information (in contrast to [39, 16]) to generate a complete and non-redundant set of connection trees (*candidate networks (CN)*) between them. Each CN may produce a number of answers to the keyword query, when evaluated on the XML graph. A presentation graph is generated for each CN, since they correspond to the different schemata of results. The *CN Generator* of XKeyword is an extension to XML databases of the CN Generator of DISCOVER [42]. We also present ways to improve the performance of the algorithm described in [42].

The CN's are passed to the query optimizer, which generates an *execution plan*. The key challenges of the optimizer are (a) to decide which connection relations to use to efficiently evaluate each CN and (b) to exploit the reusability opportunities of common subexpressions among the CN's. Both decisions, which are shown to be NP-complete, dramatically affect the performance as we show experimentally.

Finally, the results are presented to the user. XKeyword offers two presentation methods: displaying a presentation graph for each CN (Figure IV.2 (c)),


XKeyword demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Go
Search
Done
Internet

http://kebab.ucsd.edu:81/xkeyword/

Keyword Proximity Search on the DBLP database

Search terms: Yannis Andrey

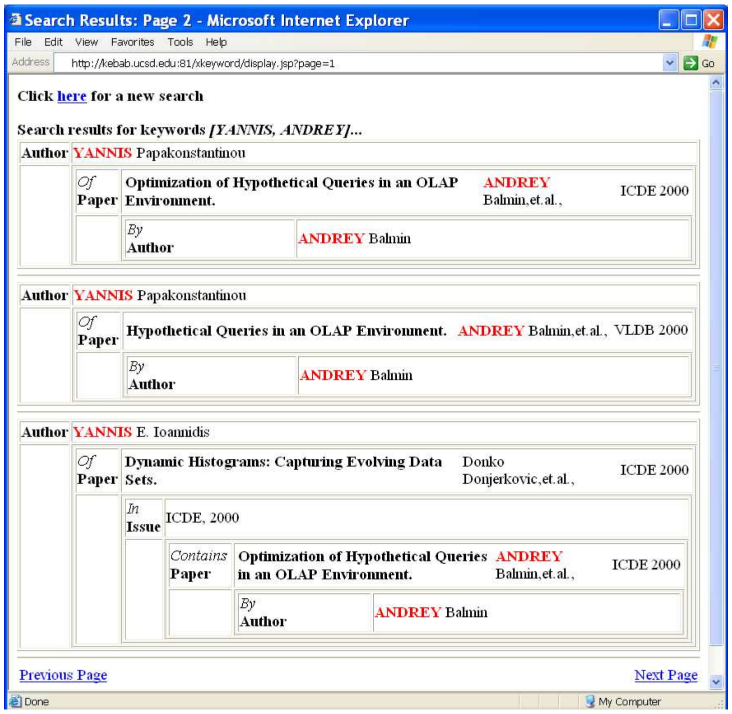
Maximum Candidate Network: 4

A subset of the DBLP database is used that contains 12 major conferences including VLDB, SIGMOD, ICDE, PODS, up to year 2001.

Powered by XKeyword technology.

Last update: 7/3/2002

(a) Query page


Search Results: Page 2 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Go
Done
My Computer

http://kebab.ucsd.edu:81/xkeyword/display.jsp?page=1

Click here for a new search

Search results for keywords *[YANNIS, ANDREY]*...

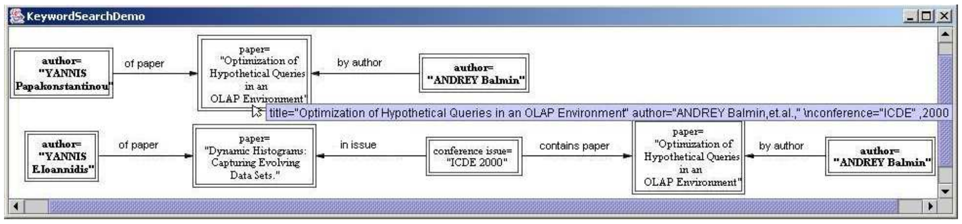
Author YANNIS Papakonstantinou
Of **Paper** **Optimization of Hypothetical Queries in an OLAP Environment.** ANDREY Balmin,et.al., ICDE 2000
By **Author** ANDREY Balmin

Author YANNIS Papakonstantinou
Of **Paper** **Hypothetical Queries in an OLAP Environment.** ANDREY Balmin,et.al., VLDB 2000
By **Author** ANDREY Balmin

Author YANNIS E. Ioannidis
Of **Paper** **Dynamic Histograms: Capturing Evolving Data Sets.** Donko Donjerkovic,et.al., ICDE 2000
In **Issue** ICDE, 2000
Contains **Paper** **Optimization of Hypothetical Queries in an OLAP Environment.** ANDREY Balmin,et.al., ICDE 2000
By **Author** ANDREY Balmin

Previous Page Next Page

(b) Presentation as a list of results


KeywordSearchDemo
author= "YANNIS Papakonstantinou"
of paper
paper= "Optimization of Hypothetical Queries in an OLAP Environment"
by author
author= "ANDREY Balmin"
title="Optimization of Hypothetical Queries in an OLAP Environment" author="ANDREY Balmin,et.al.," \nconference="ICDE" ,2000
author= "YANNIS E.Ioannidis"
of paper
paper= "Dynamic Histograms: Capturing Evolving Data Sets."
in issue
conference issue= "ICDE 2000"
contains paper
paper= "Optimization of Hypothetical Queries in an OLAP Environment"
by author
author= "ANDREY Balmin"

(c) Displaying results using Presentation Graphs

Figure IV.2: XKeyword demo

or displaying a full list of results (Figure IV.2 (b)), where each result is a tree that contains every keyword exactly once. The former method offers a more compact and non-redundant representation, while the latter favors faster response times.

In summary, we make a number of contributions in the area of keyword proximity search:

- We present keyword proximity search semantics, extended to capture our novel result presentation method, which prevents information overflow and allows the user to navigate in the result.
- We present an architecture and framework that allows for choosing which connections between objects will be precomputed. We present rules to avoid generating any *useless* connection relation, i.e., connection relations that are not efficient to evaluate any CN. We show how to bound the number of joins needed to output a solution.
- We address the on-demand performance requirement of the presentation approach and we compare and analyze different decomposition schemes with respect to it. We also present an algorithm that efficiently generates the full list of results by caching partial results and avoiding to recompute the common result portions and show experimentally that it is up to 80% faster than the naive approach used in [42] and [3].

XKeyword has been implemented (Figure IV.2) and a demo is available at http://www.db.ucsd.edu/XKeyword, which operates on the XML data of the DBLP database.

IV.B Related Work

There is a number of proposals for less structured ways to query XML database by incorporating keyword search [34, 104] or by relaxing the semantics of the query language [49, 5]. However none of these works incorporates proximity search. Florescu et al. [34] propose an extension to XML query languages that enables keyword search at the granularity of XML elements, which helps novice users formulate queries. Another difference of this work from XKeyword is that it requires the user to specify the elements where the keywords are.

In [39] and [16], a database is viewed as a graph with objects/tuples as nodes and relationships as edges. Relationships are defined based on the properties of each application. For example an edge may denote a primary to foreign key relationship. In [39], the user query specifies two sets of objects, the $Find$ and the $Near$ objects. These objects may be generated from two corresponding sets of keywords. The system ranks the objects in $Find$ according to their distance from the objects in $Near$. An algorithm is presented that efficiently calculates these distances by building hub indices. In [16], answers to keyword queries are provided by searching for Steiner trees [73] that contain all keywords. Heuristics are used to approximate the Steiner tree problem. Two drawbacks of these approaches are that (a) they work on the graph of the data, which is huge and (b) the information provided by the database schema is ignored. In contrast, XKeyword (a) works on the relatively compact set of target objects connections (see Section IV.C) and (b) exploits the properties of the schema of the database. XKeyword also provides relatively scalable performance as the available space to store fragments (see Section IV.E) increases.

DISCOVER [42] and DBXplorer [3] work on top of a DBMS to facilitate keyword search in relational databases. They are middleware in the sense that they can operate as an additional layer on top of existing DBMS's. In contrast, XKeyword is a system dedicated to providing efficient keyword querying of XML databases, by using elaborate duplication and indexing techniques. XKeyword provides guarantees on the performance of the keyword queries, which is not possible for a middleware system. DISCOVER and DBXplorer do not consider building materialized views, which is the equivalent of redundant fragments in XKeyword. Furthermore, XKeyword adopts an elaborate presentation method using interactive graphs of results. In contrast, DISCOVER and DBXplorer output a list of results, including trivial ones. The inherent differences of XML from relational data are handled in XKeyword by introducing the notion of target object.

Both DISCOVER and XKeyword exploit reusability opportunities among the candidate networks, in contrast to DBXplorer. The candidate network generator of XKeyword is an extension of the candidate network generator of DISCOVER to exploit the information provided by the XML schema like the disjunction nodes and the maxoccurence of an edge.

XKeyword stores the XML data in a relational database [17, 80, 33, 58, 25, 75, 9], to allow the addition of structured querying capabilities in the future and leverage the indexing capabilities of the DBMS's. Some of these works [33, 58, 25] did not assume knowledge of an XML schema. In particular, the Agora project employed a fixed relational schema, which stores a tuple per XML element. This approach is flexible but it is much less competitive than the other approaches, because of the performance problems caused by the large number of joins in SQL queries. XKeyword is different because it exploits the schema information to store the relationships between the target object id's of the XML data. The actual data are stored in XML BLOB's which are introduced in [9].

IV.C Framework and Proximity Keyword Query Semantics

We use the conventional labeled graph notation to represent XML data. The nodes of the graph correspond to XML elements and are labeled with the tags of the corresponding elements and an optional string value. Figure I.2 shows an example of an XML graph. An edge of the graph denotes either containment (e.g., any "*person* → *name*" edge) or an IDREF-to-ID relationship (e.g., any "*supplier* → *person*" edge) or a cross-document XML Link [90]. We will collectively refer to IDREF-to-ID and XML Link edges as *reference edges* and to the rest as *containment edges*.

We allow the graph to have multiple roots, i.e., multiple nodes with no incoming containment edge, for two reasons: First, the administrator may choose to omit the root of an XML document from the graph, since the root often provides an artificial connection between semantically unrelated first level elements. For example, had we included a root in Figure I.2 it would appear as persons and parts are closely connected (two edges way) via the root; such a connection would be artificial. A second reason for multiple roots is that we may want the graph to capture multiple XML documents, potentially linked via cross-document XML Links. We also assume that every node has a unique id, invented by the system if the corresponding element has no ID attribute. Note that the graph does not consider any notion of order among the nodes $v_, \ldots, v_n$ pointed by a parent node v. In summary:

Definition 17 (XML Graph) *An XML graph G is a labeled directed graph where every node v has a unique id* $\mathrm{id}(v)$*, a label $\lambda(v)$ coming from the set of element tags T and optionally a value* $\mathrm{val}(v)$ *coming from the set of values V. Edges are classified into containment and reference edges.*

Figure I.2 illustrates an XML graph. By convention, we indicate containment edges with solid lines and reference edges with dotted lines. We omit id's from the figures and we include the values in brackets.

Schema Graphs We use *schema graphs*[2] to describe the structure of the XML graphs. Schema graphs are similar to XML Schema definitions [91] but have typed references. We have simplified the content types captured by an XML Schema and kept only the constructs that are useful for performance optimization.

Definition 18 (Schema Graph) *A schema graph is a directed graph G_s where every node v^s is annotated with a label $\lambda(v^s)$ coming from the set of element tags T and a* content type $\mathrm{type}(v^s)$ *taking the value* `all` *or the value* `choice`*. Every edge*

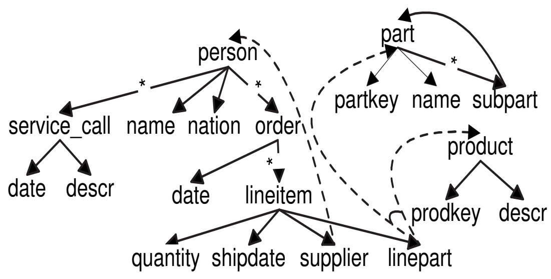


Figure IV.3: TPC-H based schema graph

e^s is classified as being a containment edge or a reference edge and is annotated with a maximum occurrence occ(e^s)*, which can be a positive number or* `*`

and (ii) a semantic explanation *string, which is a human-readable string that describes the intuitive meaning of edges of this type.*

A XML graph is said to conform to a schema graph if there is a mapping μ that maps every node v of the XML graph into a node $\mu(v)$ of the schema graph and the following conditions apply:

- v is a root node of the XML graph if and only if $\mu(v)$ is a root node of the schema graph.
- $\lambda(v) = \lambda(\mu(v))$
- if there is a containment (resp. reference) edge e from v to u in the XML graph then there is a containment (resp. reference) edge e^s from $\mu(v)$ to $\mu(u)$ in the schema graph.
- given a XML graph node v, if $\mu(v)$ has an outgoing containment (resp. reference) edge e^s that points to a schema graph node u^s and $occ(e^s) = n, n \neq *$ then there are at most n containment (resp. reference) edges e from v to a node u where $\mu(u) = u^s$.
- given a XML graph node v, if $type(\mu(v))$ = `choice` then for all edges $e_1, \ldots, e_n$ from v to corresponding target nodes $u_1, \ldots, u_n$ it is $\mu(u_1) = \ldots \mu(u_n)$.

The data of Figure I.2 conform to the TPC-H-like schema of Figure IV.3, where dotted lines denote reference edges and solid lines stand for containment edges. We denote `choice` nodes with an arc over their outgoing edges; all other nodes are of type `all`. In Figure IV.3, only "linepart" is a `choice` node.

Finally, we define an *uncycled directed graph* $G(V, E)$ to be a directed graph, whose equivalent undirected graph $G_u(V, E')$ has no cycles. An edge (v_1, v_2) is created in G_u if G has edges (v_1, v_2) or (v_2, v_1).

IV.C.1 Semantics of Keyword Queries and Presentation of Results

A *keyword query* is a set of keywords $k_1, \ldots, k_m$. The result of a keyword query is the set of all possible *Minimal Total Target Object Networks (MTTON's)*. We define MTTON's after we have first defined minimal total node networks (MTNN's). A *node network* j of an XML graph G is an uncycled subgraph of G, such that for each edge $(n_1, n_2) \in j$ it is is $(n_1, n_2) \in G$. A *total node network* j of the keywords $\{k_1, \ldots, k_m\}$ is a node network, where every keyword k is contained in at least one node n of j, i.e., $\forall k \in \{k_1, \ldots, k_m\}$, $\exists n \in j : k \in \mathit{keywords}(n)$, where $\mathit{keywords}(n)$ is the set of keywords contained in the tag or the value of n. A *Minimal Total Node Network (MTNN)* j of the keywords $\{k_1, \ldots, k_m\}$ is a total node network where no node can be removed and j still be a total node network. The *score* of a MTNN j is its size in number of edges. For example the following MTNN N_0 of the keyword query "John, VCR" has size 8.

$$\begin{gathered} N_0 : \quad \mathit{name}[\mathit{John}] \leftarrow \mathit{person} \leftarrow \mathit{supplier} \leftarrow \mathit{lineitem} \rightarrow \\ \mathit{linepart} \rightarrow \mathit{part} \rightarrow \mathit{subpart} \rightarrow \mathit{part} \rightarrow \mathit{name}[\mathit{VCR}] \end{gathered}$$

Notice that in the general case, the size of the MTNN's of a keyword query is only data bound. Hence the user specifies the maximum size Z of an MTNN that is of interest to him/her.

To ensure that the result of a keyword query is semantically meaningful for the user we introduce the notion of *target objects*. For every node n in the XML graph we define (using the schema, as we will see later) a segment of the XML graph, called *target object* of the node n (or simply called target object when the node n is obvious from the context.) Intuitively, a target object of a node n is a piece of XML data that is large enough to be meaningful and able to semantically identify the node n while, at the same time, is as small as possible. For example, consider the MTNN N_0 above. The user would like to know which is the part number of the VCR, which is the part p of which the VCR is a subpart, which line item includes p, and what is the last name of John.[2]

The target objects provide us such information. It makes sense to output the "partkey" of the VCR part as well as the name and "partkey" of the TV. On the other hand it would not make sense to output all the subparts of the TV or the orders of the person. They could be too many and of no interest in semantically identifying the node. Hence, we define the person element with the name and nation subelements to be a target object, and the part with the "partkey" and name to be another target object.

Given a MTNN j with nodes $v_1, \ldots, v_n$ there is a corresponding MTTON t,[3] which is a tree whose nodes is a minimal set of target objects $\{t_1, \ldots, t_m\}$ such that for every node $n_k \in j$ there is a $t_l \in t$ such that $\textit{target}(n_k) = t_l$. There is an edge from a target object t_i to a target object t_j if there is an edge (or as path of *dummy nodes* as defined below) from a node that belongs to t_i to a node that belongs to t_j. The score of a MTTON t is the score (size) of its corresponding MTNN. The answer to a keyword query is unique.

Specification of Target Objects The target objects are defined from an administrator using the *Target Schema Segment (TSS)* graph described next. A *TSS* graph is an uncycled graph whose nodes are called target schema segments. The

[2]Due to space limitations we do not include a last name field in the figures.

[3]The definition does not guarantee the uniqueness of the MTTON t. The nodes of j may be split in minimal sets of target objects in multiple ways. However, this is of limited practical importance since in practice it is unlikely that target objects overlap with each other in ways that enable a network to be split in multiple ways in target objects.

TSS graph is derived from a partial mapping of the nodes of the schema graph G. A node t_S is created in G_{TSS} for each set $S = \{s_1, \ldots, s_w\}$ of nodes of G that are mapped to t_S. Some nodes in G, which are called *dummy schema nodes*, are not mapped to any node in G_{TSS}, because they do not carry any information. For example *supplier*, *subpart* and *linepart* are dummy schema nodes. An edge $(t_S, t_{S'})$ is created in G_{TSS} if the schema graph has nodes $s \in S$ and $s' \in S'$, that are connected directly through an edge (s, s') or indirectly through a path of dummy schema nodes. Typically we assign to a node t_S of the TSS graph a name that is the label of the "most representative" schema graph node $s \in S$. For example, the TSS node corresponding to $\{person, name, nation\}$ is named *person* (see Figure IV.4).

Figure IV.12 illustrates the TSS graph behind our DBLP demo. Notice the semantic explanations, with the obvious meanings, that annotate the edges. Each edge is annotated with two semantic explanations: the first explains the connection in the direction of the edge and the second in the reverse direction. Similarly, the semantic explanations of the TPC-H TSS graph are shown in Figure IV.4.

Given the TSS graph, it is straightforward to define a *target decomposition* of the XML graph into target objects, connected to each other. For example a target object decomposition of the schema of Figure IV.3 and the corresponding TSS graph are shown in Figure IV.4. The MTTON of the MTNN N_0 presented above is highlighted in Figure I.2.

IV.C.2 Presentation Graph

In its simplest result presentation method (Figure IV.2 (b)) XKeyword spawns multiple threads, evaluating various plans for producing MTTON's, and outputs MTTONs as they come. The smaller MTTON's, which are intuitively more important to the user, are usually output first, since they require smaller execution times. The threads fill a queue with MTTONs, which are output to the

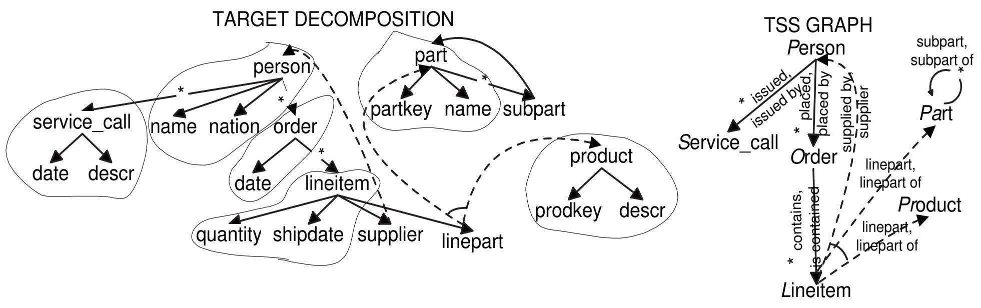


Figure IV.4: Target decomposition of a schema graph

user page by page as in web search engine interfaces.

The naive presentation method described above (and currently used by the DBLP demo) provides fast response times, but may flood the user with results, many of which are trivial. In particular, as we explained in the introduction, a redundancy similar to the one observed in multivalued dependencies emerges often. Displaying to the user results involving multivalued dependencies is overwhelming and counter-intuitive. XKeyword faces the problem by providing an interactive interface (and corresponding API) that allows navigation and hides the trivial results, since it does not display any duplicate information as we show below.

XKeyword's interactive interface presents the results grouped by the candidate networks (see Section IV.D) they conform to. Intuitively, MTTON's that belong to the same candidate network have the same types of target objects and the same type of connections between them. XKeyword groups the results for each candidate network to summarize the different connection types (schemata) between the keywords and to simplify the visualization of the result.

XKeyword compacts the results' representation and offers a drill-down navigational interface to the user. In particular, a *presentation graph* $PG(C)$ (Figure IV.1) is created for each candidate network C. The presentation graph contains all nodes that participate in some MTTON of C. A sequence of subgraphs $PG_0(C), \ldots, PG_n(C)$ are *active* and are displayed at each point, as a result of the user's actions. The initial subgraph, $PG_0(C)$, is a single, arbitrarily chosen

MTTON m of C, as shown in Figure IV.1 (a).

An expansion $PG_{i+1}(C)$ of $PG_i(C)$ on a node n of type N is defined as follows. All distinct nodes n', of type N, of every MTTON m' of C are displayed and marked as *expanded* (Figure IV.1 (b)). Note that we have to consider the statement "of type N" in a restricted sense: A candidate network may involve the same schema type in more than one roles (as is the case with tuple variable aliases in SQL.) For example, in Figure IV.1 "part" objects on the right side are VCRs while the "part" objects to their left are the "part" that contain the VCR parts. We consider those two classes of "part" objects to be two different types as far as presentation graphs are concerned. In addition a minimal number of nodes of other types are displayed, so that the expanded nodes appear as part of MTTON's. More formally, given a presentation graph instance $PG_i(C)$, its expansion $PG_{i+1}(C)$ on a node n of type N has the following properties: (a) $PG_i(C)$ is a subgraph of $PG_{i+1}(C)$, (b) for each MTTON $m' \in C$, where the node $n' \in m'$ is of type N, n' is included in $PG_{i+1}(C)$, (c) for each node $v \in PG_{i+1}(C)$ there is a MTTON z contained in $PG_{i+1}(C)$, such that $v \in z$, and (d) there is no instance $PG'_{i+1}(C)$ satisfying the above properties and the set of nodes of $PG'_{i+1}(C)$ is subset of the nodes of $PG_{i+1}(C)$.

In the implementation of XKeyword, an expansion on a node n occurs when the user clicks on n. Notice also that if the expanded nodes are too many to fit in the screen then only the first 10 are displayed.

On the other hand, a contraction $PG_{i+1}(C)$ of $PG_i(C)$ on an expanded node n of type N is defined as follows. All nodes of type N, except for n, are hidden (Figure IV.1 (c)). In addition a minimum number of nodes of types other than N are hidden, while satisfying the restriction that for each node in $PG_{i+1}(C)$ there is a containing MTTON in $PG_{i+1}(C)$ (see condition (c) below). More formally, given a presentation graph instance $PG_i(C)$, its contraction $PG_{i+1}(C)$ on an expanded node n of type N has the following properties: (a) $PG_{i+1}(C)$ is a subgraph of $PG_i(C)$, (b) n is the only node in $PG_{i+1}(C)$ of type N, (c) for each node

$v \in PG_{i+1}(C)$ there is a MTTON z contained in $PG_{i+1}(C)$, such that $v \in z$, and (d) there is no instance $PG'_{i+1}(C)$ satisfying the above properties while $PG'_{i+1}(C)$ has more nodes than $PG_{i+1}(C)$. In the implementation of XKeyword, similar to the expansion, a contraction on an expanded node n occurs when the user clicks on n.

The presentation graphs model allows the user to navigate into the results without being overwhelmed by a huge number of similar MTTON's. Furthermore, if he/she is looking for a particular result it is easy to discover it by focusing on one node at a time.

The presentation of the results of a keyword query by the interactive presentation graphs evokes the following requirements for the execution unit: First the top MTTON of each candidate network, which is the initial presentation graph, must be computed very quickly to provide a quick initial response time to the user. Second the expansion of the presentation graph must be performed on demand. This cannot be done simply by moving the cursor of some query we submit to the underlying database. Instead, when a user clicks on a node, a new minimal set of focused queries is sent to the database. These requirements and corresponding solutions are addressed in Section IV.F.

IV.D Architecture

The architecture of XKeyword (Figure IV.5) consists of a load stage, where the data are loaded to the system and all precomputations are performed, and a query processing stage that answers keyword queries.

In the *load stage*, the *decomposer* inputs the schema graph, the TSS graph and the XML graph and creates the following structures:

1. A *master index*, which stores for each keyword k a list of triplets of the form $\langle TO\_id, node\_id, schema\_node \rangle$ where TO_id is the id of the target object that contains the node of type *schema_node* with id *node_id*, which contains

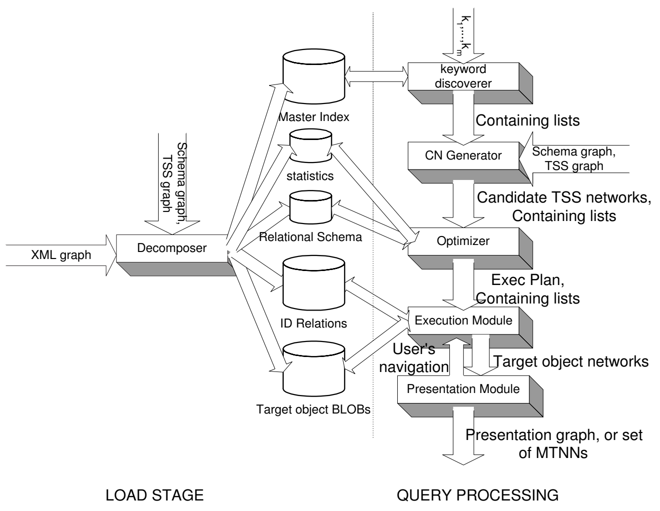


Figure IV.5: Architecture

k. The *node_id* [4] and *schema_node* are needed when calculating the score of a candidate network (Definition 19), since as we describe below, the generated relations only store target object id's.

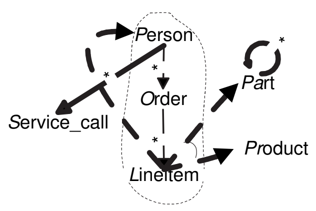
2. A set of statistics specifying: (a) the number $s(S)$ of nodes of type S in the XML graph and (b) the average number $c(S \rightarrow S')$ of children of type S' for a random node of type S.

3. *BLOBs of target objects*, which given an object id instantly return the whole target object.

4. A decomposition of the TSS graph into *fragments*, which correspond to *connection relations* that allow efficient retrieval of MTTON's.

Figure IV.6 shows a valid decomposition of the TSS graph of Figure IV.4, where the thick arrows and the closed dotted curves denote single edge and multiple edge fragments respectively. We map each fragment into a connection relation. For example, $P \rightarrow O \rightarrow L$ (in short POL, since the arrows are unambiguously implied by the TSS graph) is the connection relation that corresponds to the

[4] *node_id* is needed to distinguish two nodes of the same type and of the same target object.



POL	P_id O_id L_id
PaPa	Pa1_id Pa2_id
PS	P_id S_id
LP_ref	L_id P_id
LPa_ref	L_id PA_id
LPr_ref	L_id Pr_id

Figure IV.6: TSS graph Decomposition

fragment in the dotted line. It stores the connections among the *Person*, *Order* and *Lineitem* TSS's. *LPref* is the connection relation that corresponds to the fragment (indicated by the thick dotted line) containing the reference edge between *Lineitem* and *Person*.

Query processing consists of five stages. The *keyword discoverer* inputs the set of keywords and outputs for each keyword k the *containing list* $L(k)$ of $\langle TO\_id, node\_id, schema\_node\rangle$ where the node identified by *node_id* contains k.

The *CN Generator* takes from the containing lists the information about which schema nodes contain the keyword and works on the schema graph to calculate all possible *candidate networks (CN's)*) (Definition 19). The CN Generator works on the schema graph and not on the TSS graph because (a) important schema information like the choice nodes may be lost when we create the TSS graph and (b) the score of the MTTON's is measured in terms of schema graph edges.

A *schema node network* is an uncycled directed graph of schema nodes, where for each edge (S_1, S_2) of adjacent schema nodes S_1, S_2 there is an edge (S_1, S_2) in the schema graph. The same edge of the schema graph may appear more than once in a schema node network. Intuitively, this corresponds to the fact that target objects of the same type may be playing different roles in the MTTON's. A schema node S is free (S) if its corresponding extension has nodes that contain keywords. Otherwise it is non-free. The non-free schema node S^K is the set of nodes *of type* S that contain all keywords in K.

A node network j *belongs* to a schema node network N ($j \in N$) if there is a tree isomorphism mapping h from the nodes of j to the schema nodes of N, such that for each node $n \in j$, $n \in h(n)$.

Definition 19 (Candidate Network) *Given a keyword query $k_1, \ldots, k_m$ and a schema graph, a schema node network C is a* candidate network (CN)*, if there is an instance of the XML graph that conforms to the schema graph and has a MTNN $m \in C$.*

The CN's of size up to $Z = 8$ generated for the keyword query "TV, VCR" are the following:

`CN1`:$name^{TV} \leftarrow part \rightarrow subpart \rightarrow part \rightarrow name^{VCR}$

`CN2`:$name^{TV} \leftarrow part \rightarrow subpart \rightarrow part \rightarrow subpart \rightarrow part \rightarrow name^{VCR}$

`CN3`:$name^{TV} \leftarrow part \rightarrow subpart \rightarrow part \rightarrow subpart$
$\rightarrow part \rightarrow subpart \rightarrow part \rightarrow name^{VCR}$

`CN4`:$name^{TV} \leftarrow part \leftarrow linepart \leftarrow lineitem \leftarrow order$
$\rightarrow lineitem \rightarrow linepart \rightarrow part \rightarrow name^{VCR}$

`CN5`:$name^{TV} \leftarrow part \leftarrow linepart \leftarrow lineitem \leftarrow order$
$\rightarrow lineitem \rightarrow linepart \rightarrow product \rightarrow descr^{VCR}$

There are four more CN's which are the dual of CN1, CN2, CN3 and CN5 (CN4 is symmetric, so it is equivalent to its dual) when "TV" and "VCR" are swapped. We ignore these CN's in the analysis that follows for compactness.

The CN generator algorithm is based on the algorithm described in DISCOVER [42]. It has been extended to address the unique features of XML, such as choice nodes and distinction of containment and reference edges The CN Generator algorithm is complete (the proof is an extension of the proof of [42]), that is, all MTNN's of size up to Z belong to an output CN. Furthermore, the algorithm is non-redundant, that is, for each output candidate network C there is an instance of the database that contains a MTNN $j \in C$ and there is no other candidate network C' such that $j \in C'$.

Recall that the connection relations store only target object id's. Hence we reduce the candidate networks to *TSS networks*, which are uncycled directed graphs of TSS's, where for each edge (T_1, T_2) between TSS's T_1, T_2 there is an edge (T_1, T_2) in the TSS graph. The unique TSS network that corresponds to a candidate network is called *candidate TSS network (CTSSN).*

The candidate TSS networks corresponding to the candidate networks of size up to $Z = 8$ are the following, where $T^{k,S}$ denotes a TSS T that contains keyword k in its schema node S:

`CTSSN1:` $Part^{TV,name} \rightarrow Part^{VCR,name}$

`CTSSN2:` $Part^{TV,name} \rightarrow Part \rightarrow Part^{VCR,name}$

`CTSSN3:` $Part^{TV,name} \rightarrow Part \rightarrow Part \rightarrow Part^{VCR,name}$

`CTSSN4:` $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Part^{VCR,name}$

`CTSSN5:` $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Product^{VCR,descr}$

The candidate TSS networks are output by the CN Generator. The *Optimizer* is an adaptation of the optimizer of [42]. It uses the schema information on the connection relations and the available statistics to generate the best *Execution Plan* that evaluates the set of candidate TSS networks.

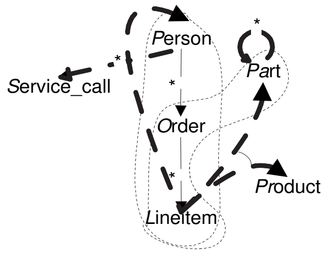
An important feature of the optimizer is that it exploits the common subexpression reusability opportunities among the candidate TSS networks as in $CTSSN4$ and $CTSSN5$ below. For example, an execution plan for the above set of candidate TSS networks is:

$CTSSN1 \leftarrow PaPa^{(TV,part1.name),(VCR,part2.name)}$

$CTSSN2 \leftarrow PaPa^{(TV,part1.name)} \bowtie_{Pa2\_id=Pa1\_id} PaPa^{(VCR,part2.name)}$

$CTSSN3 \leftarrow PaPa^{(TV,part1.name)} \bowtie_{Pa2\_id=Pa1\_id}$

$PaPa \bowtie_{Pa2\_id=Pa1\_id} PaPa^{(VCR,part2.name)}$



POL	P_id O_id L_id
PaPa	Pa1_id Pa2_id
PS	P_id S_id
LP_ref	L_id P_id
LPa_ref	L_id PA_id
LPr_ref	L_id Pr_id
OLPa	O_id L_id Pa_id

Figure IV.7: Another TSS Graph Decomposition

$$temp1 \leftarrow LPa\_ref^{TV,part.name} \bowtie_{L\_id=L\_id} POL \bowtie_{O\_id=O\_id} POL$$
$$CTSSN4 \leftarrow temp1 \bowtie_{L\_id=L\_id} LPa\_ref^{VCR,part.name}$$
$$CTSSN5 \leftarrow temp1 \bowtie_{L\_id=L\_id} LPr\_ref^{VCR,product.descr}$$

The *Execution Module* inputs the execution plan from the Optimizer. If the presentation graph method is used to present the results, the Execution Module interacts with the *Presentation Module* to direct the execution according to the user's navigation on the presentation graphs. In the case of the full list of results presentation method, a stream of results is output. The details of the Execution Module are described in Section IV.F.

Finally the Presentation module displays the results as described in Section IV.C.2.

IV.E XML Decompositions

The decomposition of the TSS graph into fragments determines how the connections of the XML graph are stored in the database, and consequently the generated execution plan for the candidate TSS networks. We have found that the selected decomposition can dramatically change the performance of XKeyword, especially for top-K queries.

Example 17 *Consider the keyword query "TV, VCR" and $CTSSN4$:*

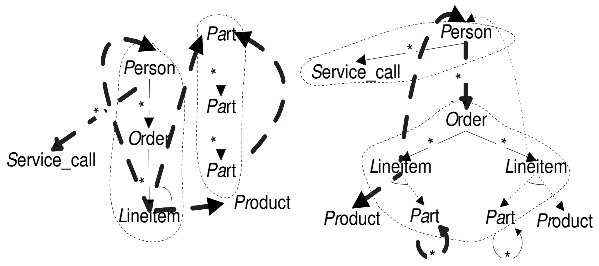


Figure IV.8: Unfolded TSS Graph Decompositions

$Part^{TV,name}$ ← Lineitem ← Order → Lineitem → $Part^{VCR,name}$ from Section IV.D. $CTSSN4$ requires three joins given the decomposition of Figure IV.6. Consider the TSS graph decomposition of Figure IV.7, which includes an OLPa fragment. With this decomposition, $CTSSN4$ can be evaluated with a single join $OLP^{TV,part.name} \bowtie OLP^{VCR,part.name}$.

Often we need to build *unfolded* fragments that contain the same TSS more than once, to store the same edge of the TSS graph more than once, as shown in the example below.

Example 18 *Consider the network $CTSSN2$:*
$Part^{TV,name}$ → Part → $Part^{VCR,name}$ of Section IV.D. This network connects three Part nodes by following the Part → Part edge twice. Under any non-unfolded decomposition this network cannot be executed without a join. However, the first unfolded TSS graph of Figure IV.8, which "unrolls" the PartPart cycle, allows the creation of the Part → Part → Part fragment, which can evaluate $CTSSN2$ without a join.

Similarly, $CTSSN4$ can be evaluated without a join, if we create the Part ← Lineitem ← Order → Lineitem → Part fragment on the second unfolded TSS graph of Figure IV.8, where the Order → Lineitem edge has been "split", i.e., the Order TSS has two children Lineitem TSS's. Notice that not all edges of the unfolded TSS graphs have to be in the decomposition. For example in the second unfolded TSS graph of Figure IV.8, the second Lineitem → Person edge is not in *a fragment, since there is a fragment for the first Lineitem $\rightarrow$ Person edge.*

Definition 20 (Walk Set, Unfolded TSS Graph) *A walk set of a TSS graph G, denoted $WS(G)$, is the set of all possible walks in G. A graph G_u is an unfolded TSS graph of the TSS graph G if $WS(G_u) = WS(G)$.*

Definition 21 (TSS Graph Decomposition) *A decomposition of a TSS graph $G = \langle N, E\rangle$ is a set of fragments $F_1, \ldots, F_n$, where for each fragment $F\langle N, E\rangle$ there is an unfolded TSS graph $G_u = \langle N_u, E_u\rangle$ of G, such that F is a subgraph of G_u. Every edge of G has to be present in at least one fragment.*

Lemma 4 *Any candidate TSS network can be evaluated given a TSS graph decomposition with the properties of Definition 21.*

The size of a fragment is the number of edges of the TSS graph that it includes. Note that a TSS graph decomposition is not necessarily a partition of the TSS graph – a TSS may be included in multiple fragments (Figure IV.7).

Each fragment $F = \langle N, E\rangle$ corresponds to a *connection relation* R, where each attribute corresponds to a TSS and is of type ID[5]. A tuple is added to R for each subgraph of type F in the *target object graph*, which is the representation of the XML graph in terms of target objects, that is, each node of the target object graph is a target object. Connection relations are a generalization of *path indexes* [31].

IV.E.1 Decomposition Tradeoffs

There is a tradeoff between the number of fragments that we build and the performance of the keyword queries, as we shown in Section IV.G. Assume that we consider solutions to the keyword queries which contain up to $M+1$ target objects. That is, the maximum size of a candidate TSS network is M. The one extreme is to create the *minimal decomposition*, where a fragment is built for each

[5]In RDBMS's we use the "integer" type to represent the "ID" datatype.

edge of the TSS graph. Then, each candidate TSS network C requires $S-1$ joins to be evaluated, where S is the size of C. We have found that the minimal is the most efficient decomposition for the on-demand expansion of a presentation graph, because the execution algorithm first tries to connect the new target objects to the adjacent nodes in the presentation graph, and gradually tries further nodes (Figure IV.11).

The other extreme is the *maximal decomposition*, where a fragment F is built for every possible candidate TSS network C. F is created by replacing the non-free TSS's of C with free TSS's. Then C is evaluated with zero joins. Clearly, the maximal decomposition is not feasible in practice due to the huge amount of space required.

Notice that M can be calculated by the maximum size Z of the MTNN's of the keyword query. In particular, the size S of a candidate TSS network C is bound by the size S' of the corresponding candidate network C' with the *size association function* f, which depends on the schema graph, the number of keywords and the TSS graph. It is $S \leq f(S')$. Hence

$$M = f(Z) \tag{IV.1}$$

For the schema graph of Figure IV.3, two keywords and the TSS graph of Figure IV.4, it is $f(S') = 2 \cdot S' + 2$.

The clustering and indexing of the connection relations are critical because they determine the performance of the joins. In the maximal decomposition, a multi-attribute index is created for every valid (i.e., the keywords can be on these attributes) combination of attributes of every connection relation. In all non-maximal decompositions, we found (Section IV.G) that the performance is dramatically improved when a connection relation R is clustered on the direction that R is used. For example, consider the execution plan of Section IV.D. If the evaluation of $CTSSN3 \leftarrow PaPa^{(TV,part1.name)} \bowtie_{Pa2\_id=Pa1\_id} PaPa \bowtie_{Pa2\_id=Pa1\_id} PaPa^{(VCR,part2.name)}$ starts from the left end, then all three $PaPa$ connection relations should be clustered from left to right. If creating all clusterings for each fragment is too expensive with respect to space, then single attribute indices are created on every attribute of the connection relations, since we found that multi-attribute indices are not used by the DBMS optimizer to evaluate join sequences.

The number of joins to evaluate the query q corresponding to a candidate TSS network is critical, because of the nature of q, which always starts from "small" connection relations. Also, the connection relations only store ID's and have every single attribute index, which makes the joins index lookups. The significance of the number of joins was verified experimentally (Section IV.G). Hence, we specify for each decomposition an upper bound B to the number of joins to evaluate any candidate TSS network of size up to M. For example $B = 0$ and $B = M - 1$ for the maximal and minimal decompositions respectively.

Given B, we generally prefer to build fragments of small sizes to limit the space of storing them. Theorem 1 proves that we can bound the size of the fragments of the decomposition.

Theorem 1 *There is always a decomposition D, whose fragments' maximum size is $L = \lceil \frac{M}{B+1} \rceil$ and any candidate TSS network of size up to M is evaluated with at most B joins.*

Proof: Assume that D is the decomposition that contains exactly all possible fragments of size L. We show how to evaluate a candidate TSS network C of size M (if the size is smaller than M it is an easier case) using D. First we partition the edges of C into connected sets of size L. Notice that the last set s may have size smaller than L. The number of sets is $\lceil M/L \rceil = B + 1$. Each such set corresponds to a fragment in D. For the last set s we pick a fragment that contains s. Hence we have to join $B + 1$ fragments, which needs B joins.

Depending on the TSS graph, we may need to build all possible fragments of size L to satisfy the constraint B on the number of joins. Theorem 2 shows such a class of TSS graphs. □

Theorem 2 *If all edges of the TSS graph are star ("*") edges and $\exists L \in \mathbf{N}$, such* *that $M = L \cdot (B+1)$, then the decomposition D must contain all fragments of size L to satisfy the constraint B on the number of joins.*

Proof sketch: Assume that a fragment F of size L is not in D. We show that there is a candidate TSS network C that cannot be evaluated with B joins. C is constructed as follows: If r is the root of F then, we replicate F $B+1$ times and make their root common. Then C needs more than B joins if F is not available. $\diamondsuit$

Often it is not efficient to build all fragments of size L, because a fragment may take up too much space despite its small size (in number of edges). This happens when the corresponding connection relation of a fragment has a non-trivial multivalued dependency (MVD), as the $PaLOLPa$ fragment in Figure IV.8, which has the MVD $O\_id \twoheadrightarrow L1\_id, Pa1\_id$. We say that a fragment has an MVD when its corresponding connection relation has an MVD.

Theorem 3 *A fragment F has a non-trivial MVD iff F contains a path $p = (e_1, \ldots, e_n)$ and $\exists e_i \in \{e_1, \ldots, e_n\}, \exists e_j \in \{e_1, \ldots, e_n\}, i < j$, and*

- $e_i \in \{\overset{*}{\leftarrow}, \xrightarrow{ref}, \underset{ref}{\overset{*}{\rightarrow}}, \underset{ref}{\overset{*}{\leftarrow}}\}$ *and*
- $e_j \in \{\overset{*}{\rightarrow}, \xleftarrow{ref}, \underset{ref}{\overset{*}{\rightarrow}}, \underset{ref}{\overset{*}{\leftarrow}}\}$ *and*
- $\nexists l, i < l < j-1, e_l \in \{\rightarrow\} \wedge e_{l+1} \in \{\leftarrow\}$

Proof sketch: Assume that R is the corresponding connection relation of F. First we prove that if F contains p, then F has an MVD. We assume that V is the set of nodes of F. We can assume that there is no star edge $e \in \{e_{i+1}, \ldots, e_{j-1}\}$. If there were, we would consider the path $p' = \{e_i, \ldots, e\}$ or $p' = \{e, \ldots, e_j\}$ if e is $\overset{*}{\rightarrow}$ or $\overset{*}{\leftarrow}$ respectively. For the same reason we assume that there are no ref edges in $\{e_{i+1}, \ldots, e_{j-1}\}$. Assume that $e_i = (v_i, v_i')$ and $e_j = (v_j, v_j')$. By the hypothesis no l exists, so there is a one-to-one relationship between v_i' and v_j. Also, by the hypothesis it is obvious that one-to-many relatioships exist between v_i' and v_i, and

v_j and v_j'. Hence, R has the MVD $v_i' \twoheadrightarrow v_i \cup V_L$, where V_L is the set of nodes of p on the left of v_i.

The inverse specifies that if R has an MVD then the conditions of the theorem hold. Assume that the MVD is $v_m \twoheadrightarrow V_m$, where $v_m \in V$ and $V_m \subseteq V$. If the MVD is non-trivial there must be a one-to-many relationship from v_m to an attribute $v_i \in V_m$ and from v_m to an attribute $v_l' \in (V - V_m - v_m)$. If the hypothesis about l did not hold, then F would be empty since $R = \pi_{V_m \cup v_m} R \bowtie_{v_m = v_m} \pi_{V - V_m} R$, by the definition of an MVD. $\diamondsuit$

We classify TSS graph fragments and decompositions based on the storage redundancy in the corresponding connection relations. Connection relations that correspond to a single edge in the TSS graph, by definition are always in 4NF. Some wider connection relations, for example the *OLPa* relation of Figure IV.7 can be in 4NF, however most of them will not be in 4NF. Non-MVD, no-4NF connection relations, are called *inlined* connection relations. A fragment is 4NF, inlined, or MVD, if the resulting connection relation is 4NF, inlined, or MVD respectively.

There are two classes of fragments that should never be built because no candidate TSS network can efficiently use them. We call such fragments *useless*:

1. If a fragment F contains a choice TSS T and more than one children of T, then F is useless, since the children of T can never be connected through T. For example, the fragment *PaLPr* is useless since *Lineitem* is a choice TSS.

2. A fragment that contains the construct $T_1 \xrightarrow{l_1} T \xleftarrow{l_2} T_2$ is useless, if $l_1 \neq ref$ and $l_2 \neq ref$, because T_1 and T_2 are never connected through T. For example, the fragment L_1PrL_2 is useless since two *Lineitem* target objects cannot connect through a *Part* target object.

We ignore useless fragments in the decomposition algorithm presented below.

Decomposition Algorithm. XKeyword uses two different decompositions. First, an *inlined, non-MVD* decomposition generated by the algorithm of Figure IV.10 is built, where B is the maximum number of joins and M is the maximum candidate

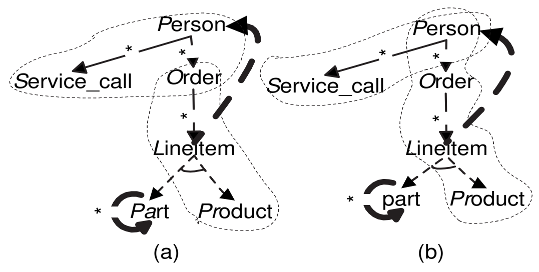


Figure IV.9: Replacing an MVD with a non-MVD fragment

TSS network size. This decomposition is used to efficiently generate the top-K results (MTTON's) in the web search engine-like presentation, and the top-1 MTTON of each CN C which corresponds to the initial instance of the presentation graph of C. Second, the minimal decomposition is built, which is used along with the *inlined, non-MVD* decomposition in the on-demand expansion of the presentation graphs.

The algorithm in Figure IV.10:

- satisfies the B constraint on the number of joins
- avoids building MVD fragments if possible
- builds non-MVD fragments of size larger than $L = \lceil \frac{M}{B+1} \rceil$ if they can eliminate MVD fragments of size L

We say that a candidate TSS network C is *covered* by a decomposition D when C can be evaluated with at most B joins.

Given $M = 4$ and $B = 1$, Figure IV.9 shows how the candidate TSS network $S \leftarrow P \rightarrow O \rightarrow L \rightarrow Pr$ is covered if we build the non-MVD fragment $POLPr$ of size $L + 1$ instead of the MVD fragment SPO of size L.

IV.F Execution

The execution module of XKeyword aims at providing fast response time to keyword queries. Depending on the presentation method selected (see Sec-

```
Decomposition Algorithm(B,M){
 Add to the decomposition $D$ the non-MVD fragments of size $\leq L$;
 Create a list $Q$ of all candidate TSS networks
   of size up to $M$ not covered by $D$;
 Add all possible non-MVD fragments of size
   greater than $L$, that help in covering at
   least one candidate TSS network $C \in Q$ and remove $C$ from $Q$;
 Add the minimum number of MVD fragments of size up to $L$
   to cover all candidate TSS networks in $Q$;
}
```

Figure IV.10: Decomposition Algorithm

tion IV.C.1), we follow a different execution approach.

Web search engine-like presentation In the case of the web search engine-like presentation of the MTTON's (Figure IV.2 (b)), we use the inlined, non-mvd decomposition (Figure IV.10) to speedup the execution of the top-K keyword query. If the CN's[6] were evaluated sequentially, and the first one did not produce any results, then the time to get the first result would be too long. We solve this problem be using a thread pool. A thread is assigned to each CN starting from the smaller ones, which need less execution time and also produce higher ranked results. A thread is returned to the pool when either the corresponding CN's evaluation completed, or a total of K results have been generated by all threads, in which case the execution ends.

The evaluation of a single CN C of the keyword query $k_1, \ldots, k_m$ is challenging for two reasons. First, since we look for K results, sending a SQL statement for C is inefficient, because the DBMS's do not currently efficiently support top-K queries. XKeyword uses nested loops join, where the nesting of the loops is determinded by a depth first traversal of C that first finds a connection between k_1

[6]For simplicity, in this section we use the term CN for both a CN and its corresponding candidate TSS network.

and k_2, then to k_3, etc. The execution is terminated after K results are produced. For example, consider $CTSSN2 : part^{TV,name} \rightarrow part \rightarrow part^{VCR,name}$ of Section IV.D. The outermost loop will iterate over the TSS $part^{VCR,name}$[7], the second loop over *part* and the innermost over $part^{TV,name}$.

The second challenge is that the *naive* nested loops join algorithm has a serious inefficiency, because it may send the same queries multiple times. In the above example, consider the case where two target objects t_1, t_2 in $part^{VCR,name}$ connect to the same target object in the *part* TSS. Then, when evaluating $CTSSN2$ for t_2, the innermost loop (over $part^{TV,name}$) should not be executed since it will produce the same results as before. Notice that this optimization would not be possible if we had put $part^{TV,name}$ as the outermost loop, because $part \rightarrow part$ is a containment and not a reference edge, so no two target objects in $part^{TV,name}$ could connect to the same target objects in the *part* TSS. The speedup of the *optimized* execution algorithm over the naive one is experimentally evaluated in Section IV.G.

In the optimized execution algorithm, there is a tradeoff between storing the past results to avoid repeating a query and keeping no past results but sending more queries. XKeyword uses a fixed size cache for each keyword query to store past results and if the cache gets full, the queries are re-send to the DBMS.

Presentation Graphs In the case of the on-demand execution based on the presentation graphs' navigation we need to modify the optimized algorithm, because we do not need the complete MTTONs, but only to find the set of expanded nodes that the user requested and their minimal connections to the presentation graph. In the above example, if the user clicks on the $part^{TV,name}$ TSS, then for each expanded node n in $part^{TV,name}$, we need to find a single connection to the *part* TSS and we ignore additional connections. In particular, we first check if n is connected to a node of *part* already in the presentation graph $PG(CTSSN2)$,

[7]In the next paragraph we explain why VCR was selected as k_1.

```
Expansion Algorithm(PG(C),n){
 $PG(C)$: current instance of presentation graph
 $n$: node to be expanded. $n$ is of type $N$
 Let $S$ be the set of target objects of type $N$;
 for each node $u$ in $S$ do
   $l := 1$;
   while $u$ not connected to all keywords and $l \leq size(C)$ do
     Check if $u$ is connected to all keywords through $PG(C)$
       with $l$ extra edges;
     l++;
   If no connection was found ignore $u$;
   else add $u$ with its connection edges to $PG(C)$;
 }
```

Figure IV.11: On-demand expansion algorithm

because we need to expand the $PG(CTSSN2)$ in a minimal way. If such a connection is not possible, we search for a connection to a fresh node of the *part* TSS. The on-demand expansion algorithm is shown in Figure IV.11.

Initially, the XKeyword decomposition (Figure IV.10) is used to efficiently retrieve the top result of each CN. Then we use a combination of the minimal and the *inlined, non-MVD* decomposition to find the minimal connection of the expanded nodes to the presentation graph, as we explain in Section IV.G.

IV.G Experiments

To evaluate the performance of XKeyword we performed a set of experiments. First, we measure the performance of the keyword queries for various decompositions of the XML schema, for top-K and full results. Then we evaluate the performance of the optimized execution algorithm for the search engine-like presentation method described in Section IV.F. Finally the performance of the on-demand expansion algorithm is evaluated.

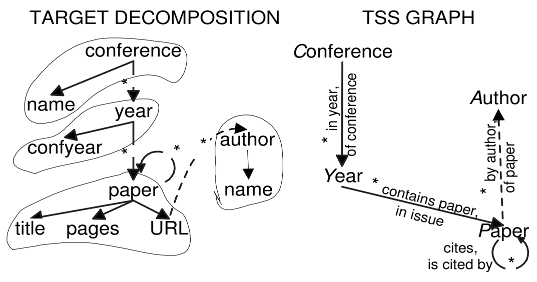


Figure IV.12: Target decomposition of DBLP

We use the DBLP XML database with the schema shown in Figure IV.12. The citations of many papers are not contained in the DBLP database, so we randomly added a set of citations to each such paper, such that the average number of citations of each paper is 20. We use Oracle 9i, running on a Xeon 2.2GHz PC with 1GB of RAM. XKeyword has been implemented in Java and connects to the underlying DBMS through JDBC. The master index is implemented using the full-text Oracle 9i interMedia Text extension. Clustering is performed using index-organized tables.

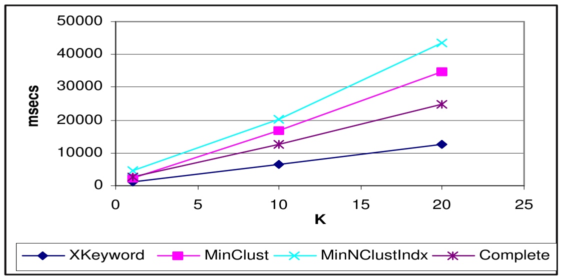
Decompositions We assume that the maximum candidate networks' size is $Z = 8$ and focus on the case of two keywords. Notice that we select a big Z value to show the importance of the selected decomposition. The absolute times are an order of magnitude smaller when we reduce Z by one. For the TSS graph of Figure IV.12, the maximum size of the CTSSN's is $M = f(8) = 8 - 2 = 6$. We require that the maximum number of joins is $B = 2$, hence from Theorem 1 it is $L = 2$. We compare five different decompositions:

1. The $XKeyword$ decomposition created by the algorithm of Figure IV.10.

2. The $Complete$ decomposition, which consists of all fragments of size L.

3. The $MinClust$ decomposition, which is the minimal decomposition with all possible clusterings for each fragment.

4. The $MinNClustIndx$ decomposition, which is the minimal decomposition



(a) Top-K results

Decomposition	secs
$XKeyword$	170.8
$Complete$	302.9
$MinClust$	225.2
$MinNClustIndx$	347.4
$MinNClustNIndx$	137.3

(b) All results

Figure IV.13: Execution times

with single attribute indices on every attribute of the ID relations.

5. The $MinNClustNIndx$ decomposition, which is the minimal decomposition with no indices or clustering.

We compare the average performance of these decompositions to output the top-K results for each candidate network. The results are shown in Figure IV.13 (a). Notice that the $Complete$ decomposition is slower than $MinClust$ although it requires a smaller number of joins, because of the huge size of the fragments that correspond to relations with multi-valued dependencies and the more efficient caching performed in the $MinClust$ decomposition. Also notice that the non-clustered decompositions (the results for $MinNClustNIndx$ are not shown, because they are worse by an order of magnitude) perform poorly for the top-K results.

Figure IV.13 (b) shows the average execution times to output all the results for each candidate network. Notice that the $MinNClustNIndx$ is the fastest, since the full table scan and the hash join is the fastest way to perform a join when the size of the relations is small relatively to the main memory and the disk transfer rate of the system, which is the case here, since all relations of the minimal decomposition have just two id (integer) attributes.

IV.H Conclusions and Future Work

XKeyword is a system that offers keyword proximity search on XML databases that conform to an XML schema. The XML elements are grouped into target objects, whose connections are stored in connection relations. Redundant connection relations are used to improve the performance of top-K keyword queries. XKeyword presents the results as interactive presentation graphs, which summarize the results per candidate network. The execution of the queries is optimized to offer fast response times.

In the future, we plan to look into different semantics for keyword queries on structured and semi-structured databases, going beyond the distance between keywords. We also work on integrating the master index tighter into the execution engine of XKeyword and on improving the response time of the system.

Chapter V

Design of a Semi-Structured Search System (S^4)

V.A Introduction

In this section we outline a Semi-Structured Search System (S^4), designed to bridge the gap between traditional database and information retrieval systems. The goal is to enable a non-expert user to ask ad-hoc, information discovery queries with a minimal knowledge of the dataset and its schema.

Traditionally, users are protected from databases by the application layer, which translates user inputs into a query language. However, with the emergence of XML and the Internet, large datasets are becoming available either “As Is” or with very simple web-based applications which do not satisfy users’ information needs. A good example of the latter is the DBLP [24] dataset, which can be queried on-line to obtain lists of papers for a specific author, conference, or a journal issue. However, this interface does not support many other search features common for the bibliography information, such as subject search, date range predicates, or authority ranking.

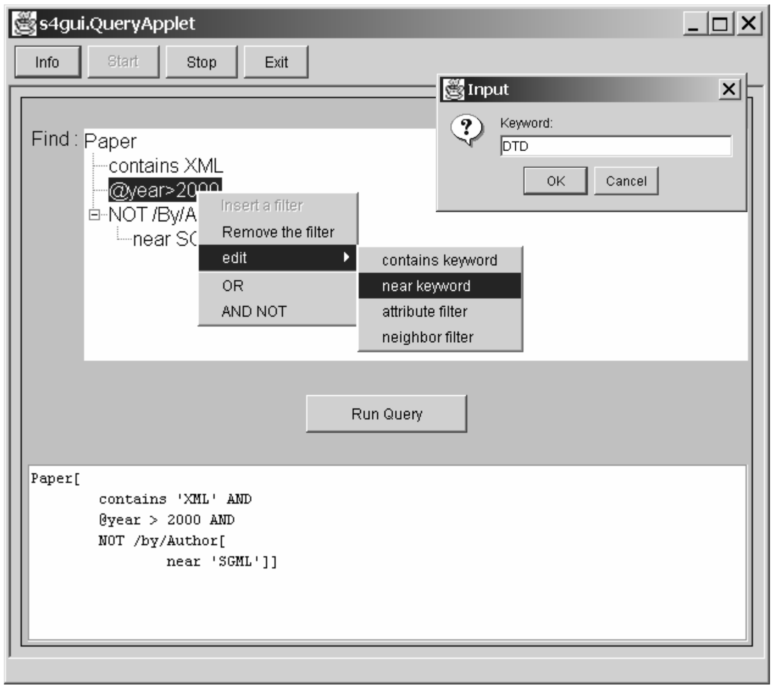
Example 19 *Consider a junior Ph.D. student trying to put together a reading list of recent papers, that are not cited in his textbooks yet. He is interested in the most* *authoritative papers on XML databases published after the year 2000. This query is very difficult to answer using the DBLP site, even though the underlying dataset contains all the necessary information.*

If the DBLP application turns out to be inadequate, a user can download the full dataset as a single XML file, store it in an XML database and use a query language such as XPath or XQuery. However, in this case the user has to understand the dataset's structure (i.e. element names and roles) and be familiar with a query language. Finally, XQuery-like languages are not suited for information retrieval type requests that are likely to arise in this scenario. A number of works propose to extend XQuery syntax [102, 35, 33] or modify its semantics [49, 5] to allow less restrictive queries. However, none of these proposals completely solve the problem. We discuss them in detail in Section V.E.

Since the structured query systems are inadequate, the user has to turn to unstructured information retrieval tools, which are based on text search. However, these systems are unable to query numeric data and the structure of XML datasets. Without numeric data, the student in Example 19 would not be able to express the [$year > 2000$] predicate of the query. Thus, he would have to sift through pages of results filled with older papers which tend to accumulate citations (and rank) over years. Text search also ignores the wealth of structural information in XML data that can be used to evaluate the query and rank the results.

V.A.1 S^4 Approach

S^4 is designed to use a "semi-structured" approach, which combines both structured and unstructured elements, to query XML datasets at the abstract level of entities and relationships. This conceptual model is much more intuitive to the user than a particular schema that implements the concept. For example, it is likely that a casual search user will know that a bibliography database contains "paper" and "person" entities with "citation" and "authorship" relationships be-



```
Paper[
        contains 'XML' AND
        @year > 2000 AND
        NOT /by/Author[
                near 'SGML']]
```

Figure V.1: A sample S^4 graphical query interface

tween them[1]. It is less likely that this user will know the organization of the bibliography. For example, papers maybe grouped by author, or authors may be grouped by paper. Conference elements may contain papers, or papers may contain references to conference elements. It is even less likely that this user will know that citations are implemented using IDREFs, XPointers, or some data attributes that need to be joined by the query, as is the case in DBLP. The use of graph data model with nodes and edges labeled by entity and relationship names, respectively, provides the flexibility and abstraction needed by search applications.

The S^4 features efficient processing of tree pattern queries. A node of the query tree can be labeled with an entity name, a wildcard, a keyword, or a local predicate, i.e., a predicate on the attributes of a single entity. Edges of the pattern are labeled with relationship names or wildcards. To simplify the presentation, we introduce a textual query language, S^4QL, that can easily be translated to

[1]The S^4 users do not need to know the exact entity and relationship names, even though they can pose more precise queries if they do.

our tree pattern formalism. The syntax of S^4QL can be extended and modified in the future, as long as basic underlying semantics remain the same. In fact, it is possible to completely forgo a textual query language, by providing a model-aware graphical query interface which assists the user in formulating the queries (Figure V.1).

The results of the search are fragments of the XML dataset that satisfy the query conditions. The fragments are ordered by their query-specific rank, computed by a PageRank-like algorithm [7], which runs on a graph of entity and relationship instances.

For example, consider the query of Figure V.1. The first two predicates of the query instruct the system to find papers related to "XML" that were published after the year 2000. Suppose the user is aware that there are two XML "communities". One with roots in semi-structured data that views XML as data, and another related to SGML that views XML as documents. If the user is only interested in papers that came out of the first community, she may restrict the query by using the last two lines, which would restrict paper's authors to those that are not related to SGML.

V.B Data Model

For semistructured search we use the labeled graph data and schema models defined in Section IV.C. We model XML Schemas as directed labeled graphs, where nodes are labeled with entity names and edges are labeled with relationships. Each relationship is represented with two edges going in opposite directions, with possibly different labels. E.g. Person and Paper entities are connected with edges "Author Of" and "By"; as in, "a person is *the author of* the paper" and "a paper *by* the person". For example, a mapping of a DBLP-like XML schema into entities and relationships is shown in Figure V.2.

Notice that the mapping does not have to be one-to-one. For example,

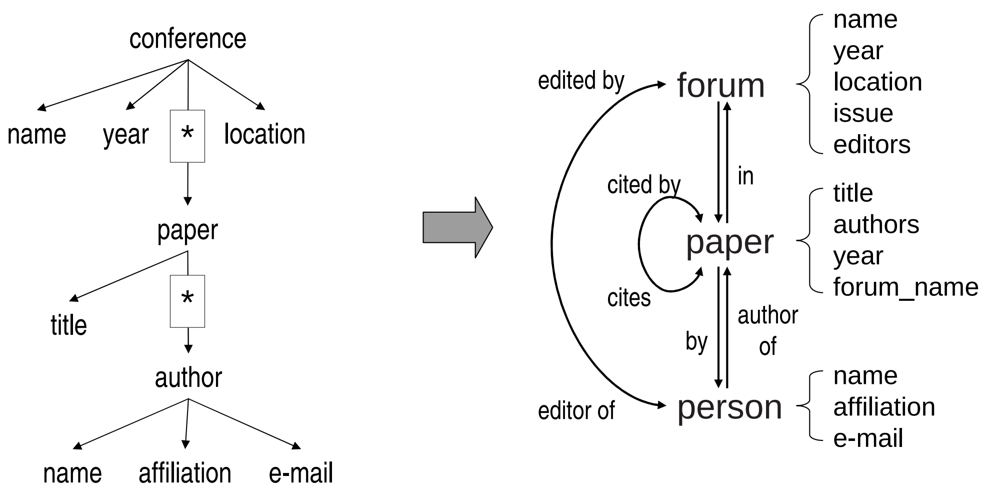


Figure V.2: An example of XML schema to S^4 schema graph mapping.

conference year information appears in both `forum` and `paper` entities. Furthermore, the paper author information appears both as an `authors` attribute, which contains a string with all authors names, and a paper-person relationship, which provides additional information about authors, such as their affiliation and contact information. The relationship also serves as a link to other papers written by the same author, which may be important for some queries.

We model XML data containing intra- and inter-document links, such as IDREF's and XPointer's, as a labeled directed graph D. The data graph is reduced to a target objects graph (TOG) by mapping D into *target objects* that conform to schema entities. All data graph nodes that participate in a target object's mapping become attributes of that target object.

Target Objects (defined in Section IV.C.1) are atomic units of information in the system and have to be small enough to be presented to the user and large enough to be meaningful. These units are similar to *Business Object* [82] in database applications or *Universal Business Objects* [44] in the data integration industry. Target objects, may or may not correspond to physical XML files in semistructured data representation.

Definition 22 (Target Object Graph) *A TOG is a labeled directed graph where* *every node v has a unique id* $\mathrm{id}(v)$*, a label* $\lambda(v)$ *coming from the set of entity names E and zero or more attributes. Each attribute a has a name* $\mathrm{name}(a)$ *coming from the set of attribute names A, and a value* $\mathrm{val}(v)$ *coming from the set of values V. A node cannot have two attributes with the same name. Each edge e of TOG has a label* $\mathbf{r}(e)$ *coming from the set of relationship names R.*

We require every relationship to have an inverse. Thus any two nodes $TO_1, TO_2 \in TOG$ are connected by either zero or two relationship edges going in the opposite directions. TO_1 and TO_2 are connected if there is a data edge $e = (n_1, n_2) \in D$, such that ($n_1 \in TO_1$ and $n_2 \in TO_2$), or ($n_1 \in TO_1$ and $n_2 \in TO_2$). The labels on relationship edges are obtained from the schema graph based on the entity names of TO_1 and TO_2, and the direction of e.

V.C Querying S^4

We will consider querying semistructured data using tree patterns with keywords and local predicates. This formalism is powerful enough to express both the usual Select-Project-Join queries and keyword queries. In the future it could be used as a basis for a more powerful query language, just as the tree pattern based XPath became the navigation language of the XQuery [101].

Definition 23 (Query Syntax) *A query is a tuple $\langle P, C\rangle$, where P is called* pattern tree *and C is called* condition expression*.*

P is a labeled tree that consists of two types of nodes:

- *Entity nodes, whose labels come from the entity name set E or a wildcard label "*". Each entity node n may also be labeled with a variable $Var(n)$.*
- *Union nodes. The same set of variables must occur in all children subtrees of a Union node. Two nodes cannot be labeled with the same variable, unless their lowest common ancestor is a Union node.*

Edges that lead to entity nodes are labeled with relationship names from R or wildcards "child" or "descendant".

C *is a logical expression involving logical predicates, logical connectives, constants, variables that occur in P, and function calls.*

The set of supported functions is implementation-dependant, except for the following functions that have to be supported:

- String containment function *contains*$(Var, Keyword)$ returns *true* if the value of some attribute of an object bound to variable Var contains the $Keyword$.

- Function *near*$(Var, Keyword)$ returns two values: a relevance score of an object bound to the variable Var with respect to the $Keyword$, and a boolean flag that shows whether or not the object is relevant. For example, the boolean result may signify that the object's relevance score is higher than a certain "noise" level.

- Attribute access function *attribute*$(Var, Name)$ returns the value of the $Name$ attribute of an object bound to variable Var. If such attribute does not exist, the function returns *null*[2].

V.C.1 S^4QL Semantics

The S^4QL query semantics are based on first matching the pattern tree with the TOG to obtain bindings and then filtering the bindings by applying the condition expression to each one. The result of the query is a list of target objects that are bound to the root node of the pattern tree. The result list is ordered by a system-specific ranking function, which computes relevance score of each target object with respect to the query.

We do not propose any particular ranking function, but provide a framework where different functions can be plugged in. The ranking function takes as

[2]We treat *null* values in the usual way - any logical predicate with a *null* argument returns $false$.

its input a set of resulting target objects and their "supporting" data. For each object, the ranking function takes a set of binding trees that produced this objects, i.e. that have this object as a root, and the scores produced by the $near()$ functions of the query for each binding. The ranking function will calculate the rank of each result object based on its structure and values of the bindings that "support" this object. One way to calculate the ranks can be based on the ObjectRank system [7]. In Section VI.B.1 we discuss challenges that arise when multiple ObjectRank need to be combined into the final object ranking.

The semantics of the pattern tree are defined in three steps. First, we rewrite "descendant" wildcards into union expressions using schema information and a system parameter $maxDepth$, which limits the depth of recursion. We define "descendant" relation as a limited-depth transitive closure of "child". Thus $descendant :: x$ is replaced with a union of $maxDepth$ different paths:
$child :: x \bigcup child :: */child :: x \bigcup \ldots \bigcup child :: */\ldots/child :: */child :: x$

Once again, the actual length of the result path that matched a "descendant" edge should be used by the ranking algorithm of the system. The $maxDepth$ cut-off is only used to disqualify paths that are too long to have any semantic meaning. Otherwise, in a connected graph any node will be a descendant of any other.

In the second step, we remove Union nodes and wildcards and produce a forest of *conjunctive pattern trees*, by traversing the pattern tree bottom-up and replacing each Union node non-deterministically by one of its children. This process is similar to producing a disjunctive normal form of a logical expression. In the same traversal we also non-deterministically replace each entity wildcard "*" with any entity label from E, and each relationship wildcard "child" with any label from R. The set of bindings produced by the pattern tree is defined as a union of the sets of bindings produced by each of the conjunctive pattern trees.

Formally, let P be a pattern tree of a query and g be the TOG. Let $Var(P)$ be the set of variables in P. Let $P_1 ... P_l$ be a set of all conjunctive pattern

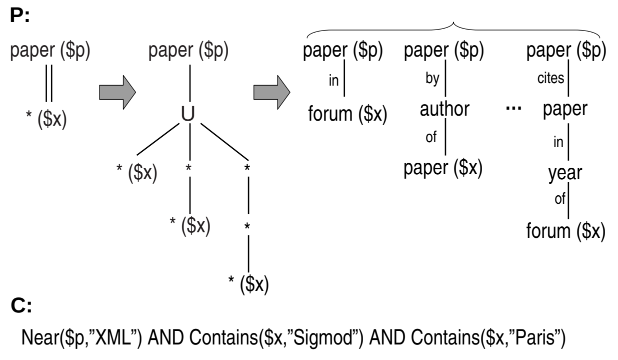


Figure V.3: An example of S^4 query semantics.

trees of P. Note that $Var(P) = Var(P_i), \forall i \in [1, l]$. A *variable binding* $\hat{\beta}$ maps each variable of $Var(P)$ to a node of g. The set of variable bindings is computed based on the set of pattern tree bindings. A pattern tree binding β maps each node n of some conjunctive pattern tree P_i to a node of g. The pattern tree binding is valid if $\beta(root(P_i)) = r$, where r is some node in g, and recursively, traversing P depth-first left-to-right, for each child p_j of a node $p \in P_i$, assuming p is mapped to $x \in g$, there exists a child x_j of x such that $\beta(p_j)) = x_j$ and $label(p_j) = label(x_j)$.

The set of variable bindings consists of all bindings $\hat{\beta} = [V_1 \mapsto x_1, \ldots, V_n \mapsto x_n]$ such that there is a pattern tree binding $\beta = [c_1 \mapsto x_1, \ldots, p_n \mapsto x_n, \ldots]$, such that $V_1 = Var(p_1), \ldots, V_n = Var(p_n)$.

The condition expression C is evaluated using the binding values and if it evaluates to true, the variable binding is qualified.

Once a set of qualified bindings is identified, the result set is constructed by removing duplicates from the set of all pattern root node bindings $\beta(root(P_i))$.

Example 20 *Consider a user trying to find papers on XML that where published in the Sigmod conference that took place in Paris. The user may have some information about the schema of the dataset, but not know all the details. Suppose the user knows that the objects that he is looking for are called "paper", but does not know what objects contain the conference information. This user's query is shown in Figure V.3. This query asks for papers that are relevant to the term "XML"* *and somehow closely connected to any object that contains the words "Sigmod" and "Paris" in it.*

The figure also shows the query transformations we used to define the semantics of the S^4QL. First, the "descendant" edge is replaced with a union of three paths, assuming that maxDepth is three. Second, a set of conjunctive patterns is produced. The figure shows three patterns, out of many possible. The first pattern will return results that the user expects, and the second one will find no results, assuming that no paper target object has the words "Sigmod" and "Paris" in it. However, the last pattern may find some results that the user did not expect to see. Namely, it will find XML papers that cite some papers published in Paris Sigmod.

The problem of finding extra results, which were not wanted by the user, is inherent in unstructured and semi-structured search process. That's why any semi-structured query requires a ranking mechanism to identify "better" results and present them to the user first. Also in a search system the user is likely to examine the results individually and identify the unwanted false positives. In the process, the user will learn new information about the structure of the dataset and the results, which will allow him to narrow down the search to eliminate false positives. In the above example, the user just needs to realize that all needed conference information is stored in the `forum` objects which are directly connected to `paper` objects. Thus, in the query of Figure V.3, the "descendant" edge label can be replaced with the "child" to produce the desired result.

Notice that we only use conversion to conjunctive pattern trees to define the semantics of S^4QL queries. We do not propose that the S^4 query processing algorithm actually follows the same process.

V.C.2 S^4QL Semantics Compared to XCacheDB QL

S^4QL semantics are very similar to semantics of the XCacheDB query language, described in Section II.E. However, S^4QL has the following differences:

- Edges that lead to element nodes are labeled with relationship names. Edges that lead to union nodes do not have labels.
- Element nodes may be labeled with "*" wildcard instead of a tag. Edges that lead to element nodes may be labeled with one of two kinds of wildcard - "child" or "descendant".
- The result tree is trivial. The query returns target objects that bind to the root of the pattern tree.
- Condition expressions can contain keyword search functions.

The first change is due to a data model extension to include edge labels. In XCacheDB data model only data nodes had labels, but in S^4 data graphs both nodes (entities) and edges (relationships) are labeled.

Wildcards do not change the expressive power of the language, since we assume existence of schema, allow unions, and limit the recursion depth. However, a translation that eliminates wildcards can exponentially increase the query size. The key to efficient (polynomial time) query execution, is processing wildcards as a whole, without rewriting them into unions. As we mention in Section VI.B, in some cases polynomial time query execution can be guaranteed, and in other cases a best-effort algorithm is needed.

The third change to the XCacheDB query language is due to the nature of S^4QL queries, which are designed for information discovery. S^4 does not construct results and only extracts individual target objects that match the root of the pattern. One possible direction for future work is to introduce result construction and aggregation into S^4.

The last item on the list is a clarification rather than an extension. XCacheDB does not have any restrictions on the type of condition expressions it can support. To simplify presentation we'll limit the list of functions and operators to $=, \neq, <, \leq, >, \geq$, and required S^4QL functions *contains*(), *near*(), and *attribute*().

V.D S^4 System Architecture

The proposed architecture of the S^4 is based on the XCacheDB system, which, in turn, is built on top of a commercial RDBMS. XML files that contain target objects and links between them, are shredded into tables for efficient query processing. Copies of target objects are also stored in text format which facilitates quick output.

Using the decomposition of an XML Schema into entities, an S^4 query can be rewritten to eliminate wildcards and translated into a XCacheDB plan exactly as described in Section II.E. Keyword conditions are processed using an external text index which outputs object IDs of the qualifying target objects into temporary tables (one table is created per keyword), which in turn participate in XCacheDB plan.

While feasible, this approach may perform poorly due to the large size of the rewritten query. If each wildcard rewrites into a union of n paths, a query with k wildcards rewrites into a union of up to n^k queries.

We have observed that various rewritten queries, by construction, tend to reuse and re-execute the group of same paths. For example, a query fragment *paper//paper*[3], may appear in many bibliography queries. However, each time it will be translated into the same union of simple paths:
$paper/author/paper \bigcup paper/cites :: paper/$
$(cites :: paper \bigcup cited\_by :: paper) \bigcup (paper/$
$cited\_by :: paper/(cites :: paper \bigcup cited\_by :: paper))$

To avoid re-evaluation of popular query fragments and speed-up query processing, we suggest using tried and true technique of rewriting queries using materialized views. The XCacheDB lends itself naturally to the use of materialized views. In Section II.E we showed how the XCacheDB constructs a query plan given a query and a schema decomposition. In presence of the materialized views, the

[3]From here on, we'll use an XPath-like textual notation for the S^4QL queries and tree pattern fragments.

same algorithm works without any changes. However, the decomposition is picked dynamically from a set of materialized views that we have identified as eligible for a given query.

We decide whether a materialized view is eligible to answer (a subset of) a query, by establishing containment between tree patterns.

Example 21 *Consider the following query, which looks for papers related to "XML", with authors related to "IBM"* $Q = paper[//*[contains\ 'XML']and\ author//*[contains\ 'IBM']]$ *and a materialized view* $V = *//*$*, which stores pairs of nodes that bind to the first and second "*" wildcard, respectively. We discuss view definitions in Section V.D.2.*

To execute the query, keywords "XML" and "IBM" are looked up in the index, producing two temporary tables T_{XML} *and* T_{IBM}*. Each of the tables has one column with a list of object IDs of target objects containing the keywords.*

Given the TOG schema, Q can be translated into a rather large union of wildcard-free queries, which can be executed directly. However, the presence of the view V can significantly improve query performance.

To take advantage of V, we first identify subsets of the query that are contained in V. There are two such subsets: $Q_1 = paper//*$ *and* $Q_2 = author//*$*. Thus, the query can be answered by executing the following query, which joins a* PA *"paper-author" table with two copies of V that cover* Q_1 *and* Q_2 *respectively.*

```
SELECT V1.s
FROM V as V1, V as V2, PA, T_XML, T_IBM
WHERE V1.s = PA.p AND V2.s = PA.a
AND V1.t = T_XML.id AND V2.t = T_IBM.id
```

V.D.1 Execution Module

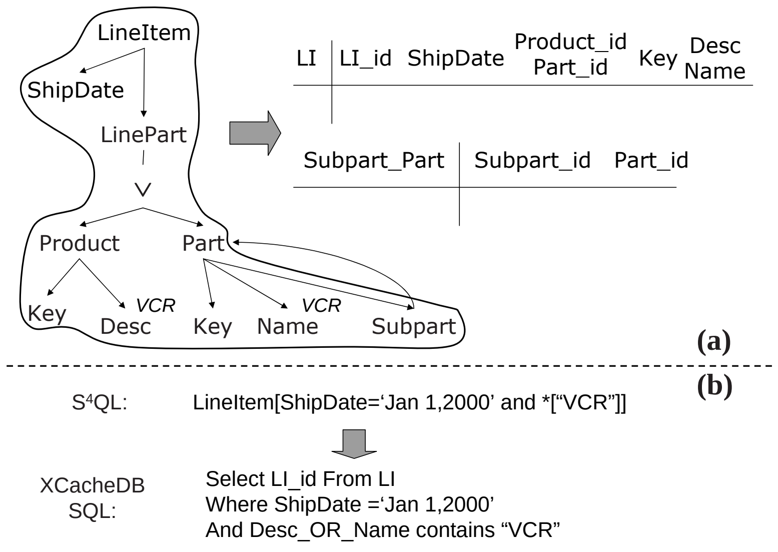
Query execution proceeds in two stages. First, text index is used for each keyword predicate in the query to find all objects that satisfy the predicate. The IDs of the resulting objects are stored in temporary tables – a table for each keyword. Second, a single SQL query is executed over the temporary keyword tables and the XCacheDB relational storage to find IDs of the objects that satisfy the S^4 query. Result objects' bodies are retrieved from the XCacheDB object storage and presented to the user.

Just as in XKeyword system, we use off-the-shelf Oracle Inter-Media text index. The drawback of the index is that it works only on individual database columns. Thus, every text index request would have to be translated into a separate Inter-Media query for each text column of each table in the system. To avoid the escalating number of text index requests, we modify the underlying XCacheDB system to store "output" copies of objects in text form instead of the XCacheDB binary encoding. These textual copies of all S^4 objects stored in a single column can all be covered by a single Inter-Media index. However, this change increases storage requirements for XCacheDB (typically by 5% to 20%, and theoretically up to 50%) and correspondingly increases time to read and output results. A better solution would be to modify the text index to work on XCacheDB binary format. However, it requires access to a source code of Inter-Media or any other comparable tool.

Once object IDs returned by the keyword searches are inserted into temporary tables, and it is known what types of objects match keyword predicates, the SQL query is produced as follows.

First, the query is restricted using schema information and types of objects that match keyword predicates.

The SQL query to the XCacheDB relational storage is constructed by joining together all temporary keyword tables, entity tables for entities that have local predicates and relationship tables for connecting keyword and entity tables. More specifically, given a query DAG, we create the "FROM" clause of the query by adding a relationship table (a.k.a *edge* table) for each edge of the query, an entity table for each node of the query that has a local predicate, and a keyword table for each node of the query that has a keyword predicate. The last two kinds



```
LineItem[ShipDate='Jan 1,2000' and *["VCR"]]
```

```
Select LI_id From LI
Where ShipDate ='Jan 1,2000'
And Desc_OR_Name contains "VCR"
```

Figure V.4: S^4 view definition containing union and it's use.

of tables are also called *node* tables. The "WHERE" clause is constructed from join predicates on ID attributes for each pair of connecting edges, as well as for each pair of node and edge tables, where the node is one of the ends of the edge.

V.D.2 S^4 Materialized Views

S^4 will rely heavily on materialized views to improve performance of the query execution module. These views are defined using S^4QL with a single restriction that condition expressions are not allowed in view definitions. Technically, our view matching framework can support views defined with predicates, even the ones containing keywords. However, it is unlikely that a user would be able to exactly predict the predicates of the future queries and materialize the appropriate views. Using keyword predicates to define the views is even less practical, since a separate view has to be stored for each keyword predicate. Text indexing can facilitate keyword searches just as efficiently and at a fraction of the storage cost.

Semantics of the S^4 views differ significantly from S^4QL queries. Instead of single objects returned by the query, an S^4 view stores values and IDs of all target objects that match any node of the expanded view definition. Thus, a view stores all bindings of the query to the target object graph, as defined in Section V.C.1. In other words, S^4 views are a natural extension of XCacheDB inlined fragments with unions. Indeed, we have observed in XCacheDB that inlined fragments perform well with predicated queries. XKeyword demonstrated usefulness of union tables for keyword searches. It is only natural to combine these features for S^4 queries which mix keywords and structured predicates. We illustrate this extension of XCacheDB fragments with the following example.

Example 22 *Figure V.4(a) shows an example of a* union-inlined *fragment rooted at "LineItem" node, which the XCacheDB translates into a single LI table. This fragment covers a disjunction, since a "LineItem" can contain one of two kinds of "LinePart" – either a simple "Product" object or a tree of "Part" objects. Note that a single fragment cannot cover the cycle in the graph. Thus, a separate table Subpart_Part is created for the edge uncovered by the LI fragment.*

Figure V.4(b) shows an S^4QL query that looks for a line item that was shipped on a certain date and connects to a product or part object with the word "VCR" in it. Given the above union-inlined fragment, this query can be answered by a single SQL query that does not include any table joins.

V.E Related Work

There are a number of proposals for less structured ways to query XML database by incorporating keyword search [34, 104] or by relaxing the semantics of the query language [49, 5]. However, none of these works incorporate proximity search. Florescu et al. [34] propose an extension to XML query languages that enables keyword search at the granularity of XML elements, which helps novice users formulate queries. Another difference of this work from our work is that it requires the user to specify the elements where the keywords are.

In [39] and [16], a database is viewed as a graph with objects/tuples as nodes and relationships as edges. Relationships are defined based on the properties of each application. For example an edge may denote a primary to foreign key relationship. In [39], the user query specifies two sets of objects, the $Find$ and the $Near$ objects. These objects may be generated from two corresponding sets of keywords. The system ranks the objects in $Find$ according to their distance from the objects in $Near$. An algorithm is presented that efficiently calculates these distances by building hub indices. In [16], answers to keyword queries are provided by searching for Steiner trees [73] that contain all keywords. Heuristics are used to approximate the Steiner tree problem. Two drawbacks of these approaches are that (a) they work on the graph of the data, which is huge and (b) the information provided by the database schema if available is ignored. In contrast, we provide smart indexing techniques that allow the quick navigation of the XML graph/tree.

DISCOVER [42] and DBXplorer [3] work on top of a DBMS to facilitate keyword search in relational databases. They are middleware in the sense that they can operate as an additional layer on top of existing DBMS's. In contrast, the system we present is dedicated to providing efficient keyword querying of XML databases, by using elaborate duplication and indexing techniques. Furthermore, we adopt an elaborate presentation method using interactive graphs of results. In contrast, DISCOVER and DBXplorer output a list of results, including trivial ones. Finally we handle the inherent differences of XML from relational by introducing the notion of target object.

Chapter VI

Conclusions and Future Work

In this text we have addressed management of XML data. We have identified addressed a novel schema validation problem, which is characteristic of the tree-structured data. We presented two incremental validation algorithms. First, a complete algorithm, which requires an index-like structure and operates in logarithmic time. Second, an even more efficient algorithm that does not require additional structures and works for a significant subset of real-life schemas.

We have examined a full specter of options for querying XML data. From structured querying of XCacheDB (Chapter II) appropriate when the database schema is fully known to the user, to the keyword proximity search of XKeyword(Chapter IV) – the only option when the schema is completely unknown. We proposed a Semi-Structured Search System (S^4), a middle of the road approach, useful in information discovery when users have incomplete knowledge of the schemas. We have observed that the same technique of using relational tables to store pre-computed pieces of query answers, works well to improve performance of all these systems. Inlined for XCacheDB, id's for XKeyword, and their combination for S^4

VI.A Future Work in Update Validation

The incremental validation algorithms we exhibited are significant improvements over brute-force validation from scratch. However, several issues on update validation need further investigation:

Lower bounds To understand how close our algorithms are from optimal, it would be of interest to exhibit lower bounds on incremental maintenance of strings, DTDs, and specialized DTDs. There are known results that yield lower bounds for validation from scratch: acceptance of a tree by a tree automaton is complete for uniform NC^1 under DLOGTIME reductions [57]. However, this does not seem to yield any non-trivial lower bound on the incremental validation problem. We are not aware of any work providing such lower bounds applicable to our framework.

Optimizing over multiple updates For a sequence of m updates, our incremental validation algorithm modifies the auxiliary structure one update at a time, then checks validity of the final updated tree. Clearly, it is sometimes more efficient to consider groups of updates at a time. For example, this may avoid performing unnecessary intermediate line rearrangements in the incremental algorithm for specialized DTDs. Also, if the number of updates is large compared to the size of the resulting tree, it may be more efficient to re-validate from scratch.

More complex updates on trees We only considered here elementary updates affecting one node at a time. Some scenarios, such as XML editors, require more complex updates arising from manipulation of entire subtrees (deletion, insertion, cut-and-paste, etc). Our approach can still be applied by reducing each of these updates to a sequence of elementary updates. However, in this case it may be more efficient to consider updates of coarser granularity.

Update Languages It is expected that XQuery will soon be augmented with an update language [81, 83]. Systems supporting complex update languages can naturally use our work: first compute the set of updates of particular nodes and then apply the incremental validation techniques described in this paper. However, this approach may miss the extra optimization opportunities presented by the fact that the set of updates has been developed by a single update statement. Realizing those opportunities requires analysis of the update statement.

VI.B Future Work in S^4

The problem of rewriting using materialized views for the S^4QL queries contains a number of challenging open problems:

1. Deciding view containment.
2. Integrating view containment and query rewriting.
3. Query plan selection.
4. Materialized view selection.
5. Ranking of the results.

The first open problem refers to proving that all query results are contained in a materialized view, and thus, that the view can be used to answer the query. The view containment problem for S^4QL is equivalent to that of boolean tree patterns with "*", "//", branching and disjunction. This problem has been shown to be coNP-hard in [60]. The same containment problem was shown to be undecidable in presence of negation or DTD schema constraints in [65].

We recently addressed the containment problem for a subset of XPath which is very similar to S^4QL ([8]). In that work we show that an incomplete, but sound and efficient algorithm based on tree mappings, can be used to detect tree containment in a vast majority of cases. The tree containment algorithm of [8] handles tree patterns with disjunctions and predicates, and thus, can be used to solve the first open problem, albeit incompletely.

The second problem can be addressed in the similar fashion. The basic view matching algorithm described in [8], determines which views can potentially be used to answer the query. However, it has one major shortcoming. It does not take advantage of schema information to expand the query, thus, it may miss many useful view matches. For example, consider the schema graph of Figure V.4(a). Assume that following two views exist, instead of a single union-inlined materialized view: $V_1 = LineItem/Part$ and $V_2 = LineItem/Product$. None of the views can be used to answer the query of Figure V.4) by itself:

$$LineItem[ShipDate = Jan1, 2000 \; and \; *[VCR]]$$

Even the disjunction of V_1 and V_2 cannot be useful, unless we can prove (using schema information) that the $*$ wildcard in the query can refer only to a "Part" or a "Product" object.

Recall, that the execution module uses schema information to expand the query, by rewriting it to eliminate the wildcards. One approach is to run matching on the expanded query. The same matching algorithm, when applied to expanded query, will produce maximal set of matches. However, the cost of matching will be prohibitively high, since the size of expanded query may be exponential in the size of the original. Notice that it is not necessary tho expand the view definition. The expansion only makes the expression more restrictive, thus, by definition, if an expanded view definition has matched the query, so will the original one.

To solve the performance problem and to find as many usable views as possible, we need to interleave view matching with schema-based expansion. One possible algorithm proceeds as follows.

First, all view definitions are matched against each node of a query, producing a set of tree mappings. If the query does not contain any wildcards, or if each query wildcard is covered by some mapping, we are done. However, if some wildcards remain un-covered, we expand each of these wildcards into union query fragment. For each wildcard, we run the matching algorithm with this wildcard expanded, and the rest remaining intact. If some of the wildcards still remain uncovered, we expand and try matching for all pairs of wildcards; then all triplets and so on. We proceed in this fashion until all wildcards are eliminated, or we exceed pre-set optimization budget, which specifies how many times we can run the matching algorithm. The budget is needed to prevent a very unlikely, but theoretically possible case where an exponential number (in the size of the query) of iterations will be needed to match all wildcards.

The third open problem has to do with constructing an SQL query given a set of matching materialized views. Once all possible tree mappings are available, S^4 constructs an SQL query that utilizes a subset of the eligible materialized views. The SQL query construction algorithm is identical to that of XCacheDB, since the materialized view are just schema fragments. The addition of union fragments does not change query processing in any way.

One new challenge that does arise in this setting is picking a subset of eligible views to use during SQL query construction. XCacheDB assumes a single schema decomposition during query translation, whereas S^4 may have overlapping view mappings, which is equivalent to a set of available decompositions. Incidentally, the very same problem was encountered in the XKeyword project (Section IV.E), where heuristics were used to identify the best set of tables to use. The same heuristics could be extended to be used in S^4. However, we feel that a carefully designed cost-based optimizer would be the best solution is this case.

Materialized view selection is an important problem that goes hand in hand with query processing using materialized views. Once we know how to take advantage of materialized views, the question arises as to which views should be constructed in the system to optimize performance of a given workload. The usual cost-based approach of commercial index advisors [105, 4] seems to be applicable to S^4. This approach requires a cost model that can estimate the query processing time. Given a cost model, the work load cost can be estimated with a number of potential materialized view sets. Usually, heuristics are used to prune the search space and to arrive at a near-optimal result in a reasonable amount of time.

VI.B.1 Ranking the Results

The S^4 is an information retrieval system, in a sense that the queries may be imprecise, and some results may be more relevant than others. Thus, the system has to rank the results to make sure that the more interesting ones are output first. For example, "Paper[near XML]" will return a very large number of results, but the user is interested in the most influential papers, which are frequently cited and whose authors are highly ranked as well. This problem can be addressed by a PageRank-like ranking algorithm based on transfer of authority along the relationship edges of the TOG graph.

The PageRank algorithm [18] has proven to be very successful in searching the web. Recently the XRANK [40] suggested using similar ranking algorithm for XML fragments. Our ObjectRank system [7] extends this work to provide *query specific* ranking of objects in graph databases. To obtain these query specific rankings the ObjectRank performs a random walk computation for every keyword, using the target objects that contain the keyword as the staring points of the walk.

Our user studies (see [7]) demonstrated the effectiveness of the ObjectRank in supporting individual "near" operators. However, more work is required to investigate combining these scores and possibly other information about the S^4 query bindings into the overall ranking of S^4 result objects.

Bibliography

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

[2] S. Abiteboul, D. Suciu, and P. Buneman. Data on the Web : From Relations to Semistructured Data and Xml. *Morgan Kaufmann Series in Data Management Systems*, 2000.

[3] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. *ICDE*, 2002.

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB Conference*, pages 496–505, 2000.

[5] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. *International Conference on Extending Database Technology (EDBT)*, 2002.

[6] Apache Software Foundation. Xindice. http://xml.apache.org/xindice/.

[7] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[8] A. Balmin, F. Özcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. of VLDB*, Toronto, Canada, 2004.

[9] A. Balmin and Y. Papakonstantinou. Storing and querying xml data using denormalized relational databases. *VLDB J.*, 14(1):30–49, 2005.

[10] A. Balmin, Y. Papakonstantinou, K. Stathatos, and V. Vassalos. System for querying markup language data stored in a relational database according to markup language schema, 2000. Submitted by Enosys Software Inc. to USPTO.

[11] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[12] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented database system : the story of O2*. Morgan-Kaufmann, 1992.

[13] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - the XML enabled data management system. In *ICDE 2000, Proceedings of the 16th International Conference on Data Engineering*, pages 561–568. IEEE Computer Society, 2000.

[14] D. Barbosa, A. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of xml documents. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, Los Alamitos, CA, 2004. IEEE Computer Society.

[15] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of the 7th Int'l. Conf. on Database Theory*, volume 1540 of *Lecture Notes in Computer Science*, New York, 1999. Springer.

[16] G. Bhalotia, C. Nakhey, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. *ICDE*, 2002.

[17] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of ICDE*, 2002.

[18] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 1998.

[19] A. Bruggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[20] B. Choi. What are real DTDs like? In *Proceedings of the Fifth International Workshop on the Web and Databases, WebDB*, pages 43–48. Informal proceedings, 2002.

[21] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 177–188, New York, 1998. ACM.

[22] Coherity. Coherity XML database (CXD). http://www.coherity.com.

[23] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[24] DBLP. http://www.sigmod.org/sigmod/dblp/db/index.html.

[25] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*. ACM Press, 1999.

[26] G. Dong and J. Su. Space-bounded foies. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 139–150, New York, 1995. ACM.

[27] Ellipsis. DOM-Safe. http://www.ellipsis.nl.

[28] The universal, real-time data integration platform, 2000. White paper at http://www.enosyssoftware.com.

[29] eXcelon Corp. eXtensible information server (XIS). http://www.exln.com.

[30] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *SIGMOD Conference*, 2001.

[31] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, pages 14–23, 1998.

[32] M. F. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: trading between relations and XML. In *WWW9 / Computer Networks*, pages 723–745, 2000.

[33] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[34] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. *WWW9 Conference*, 1999.

[35] N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *SIGIR*, pages 172–180, 2001.

[36] H. Garcia-Molina, J. Ullman, and J. Widom. *Principles of Database Systems*. Prentice Hall, 1999.

[37] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, 1980.

[38] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[39] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. *VLDB*, 1998.

[40] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. *ACM SIGMOD*, 2003.

[41] B. Hesse and N. Immerman. Complete problems for dynamic complexity classes. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, Los Alamitos, CA, 2002. IEEE Computer Society.

[42] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. *VLDB*, 2002.

[43] Infonyte. Infonyte DB. http://www.infonyte.com.

[44] Infosphere. http://www.almaden.ibm.com/software/projects/infosphere.

[45] Ipedo XML Database: Technical overview. Available at `http://www.ipedo.com/downloads/products_ixd_technical_overview.pdf`.

[46] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *VLDB J.*, 11(4):274–291, 2002.

[47] F. Jalili and J. Gallier. Building friendly parsers. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, New York, 1982. ACM.

[48] J. X. Josephine M. Cheng. XML and DB2. In *ICDE 2000, Proceedings of the 16th International Conference on Data Engineering*, pages 569–573. IEEE Computer Society, 2000.

[49] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. *PODS*, 2001.

[50] A. Kemper and G. Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *IS 17(2)*, pages 117–145, 1992.

[51] A. Krupnikov. DBDOM. http://dbdom.sourceforge.net/.

[52] J. Larcheveque. Optimal incremental parsing. *ACM Transactions on Programming Languages and Systems*, 17(1):1–15, 1995.

[53] C. Levine. Standard benchmarks for database systems, 1997. Presented at Sigmod 97. Available at `http://www.tpc.org/information/sessions/sigmod/indexc.htm`.

[54] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB 2001, Proceedings of the 27th International Conference on Very Large Databases*, pages 361–370, 2001.

[55] W. Li. A simple and efficient incremental LL(1) parsing. In *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics(SOFSEM '95)*, volume 1012 of *Lecture Notes in Computer Science*, New York, 1995. Springer.

[56] G. Linden. Incremental updates in structured documents, 1993. Licentiate Thesis, Report C-1993-19, Department of Computer Science, University of Helsinki.

[57] M. Lohrey. On the parallel complexity of tree automata. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications, (RTA 2001)*, volume 2051 of *Lecture Notes in Computer Science*, New York, 2001. Springer.

[58] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with XML and relational. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 623–626. Morgan Kaufmann, 2000.

[59] M/Gateway Developments Ltd. eXtc. http://www.mgateway.tzo.com/eXtc.htm.

[60] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *Proceedings of PODS*, pages 65–76, 2002.

[61] P. Miltersen, S. Subramanian, J. Vitter, and R. Tamassia. Complexity models for incremental computation. *TCS*, 130(1):203–236, 1994.

[62] A. Murching, Y. Prasant, and Y. Srikant. Incremental recursive descent parsing. *Computer Languages*, 15(4):193–204, 1990.

[63] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara internet query system. In *IEEE Data Engineering Bulletin 24(2)*, pages 27–33, 2001.

[64] NeoCore. Neocore XML management system. http://www.neocore.com.

[65] F. Neven and T. Schwentick. Xpath containment in the presence of disjunction, dtds and variables. In *Proceedings of ICDT*, 2003.

[66] OpenLink Software. Virtuoso. http://www.openlinksw.com/virtuoso/.

[67] M. T. Özsu and P. Valduriez. *Principles of distributed database systems.* Prentice Hall, 1999.

[68] Y. Papakonstantinou and V. Vassalos. The Enosys Markets data integration platform: Lessons from the trenches. *CIKM*, pages 538–540, 2001.

[69] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 35–46. ACM, 2000.

[70] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2000), Dallas, Texas, USA*, pages 35–46, New York, 2000. ACM.

[71] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *JCSS*, 55(2):199–209, 1997.

[72] L. Petrone. Reusing batch parsers as incremental parsers. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, New York, 1995. Springer.

[73] J. Plesn'ik. A bound for the Steiner tree problem in graphs. *Math. Slovaca 31*, pages 155–163, 1981.

[74] M. Rys. State-of-the-art XML support in RDBMS: Microsoft SQL server's XML features. In *IEEE Data Engineering Bulletin 24(2)*, pages 3–11, 2001.

[75] A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In D. Suciu and G. Vossen, editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 47–52. Springer, 2001.

[76] H. Schöning and J. Wäsch. Tamino - an internet database system. In *EDBT 2000, Proceedings of the 7th International Conference on Extending Database Technology*, pages 383–387, 2000.

[77] E. Sedlar. Managing structure in bits & pieces: the killer use case for xml. In *Proc. of SIGMOD*, pages 818–821, New York, NY, USA, 2005. ACM Press.

[78] L. Segoufin. Personal communication, 2002.

[79] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 65–76. Morgan Kaufmann, 2000.

[80] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[81] G. Sur, J. Hammer, and J. Simeon. UpdateX - an XQuery-based language for processing updates in XML. In *International Workshop on Programming Language Technologies for XML (PLAN-X 2004)*. Informal Proceedings, 2004.

[82] J. Sutherland. Business objects in corporate information systems. *ACM Comput. Surv.*, 27(2):274–276, 1995.

[83] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld". Updating xml. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, 2001. ACM.

[84] TPC benchmark C. Available at `http://www.tpc.org/tpcc/`.

[85] TPC benchmark H, 1999. Standard specification available at http://www.tpc.org.

[86] Unstructured information management architecture (UIMA). http://www.research.ibm.com/UIMA/.

[87] H. Vollmer. *Introduction to Circuit Complexity.* Springer Verlag, New York, 1999.

[88] W3C. Document object model (DOM), 1998. W3C Recomendation at http://www.w3c.org/DOM/.

[89] W3C. The extensible markup language (XML), 1998. W3C Recomendation at http://www.w3c.org/XML.

[90] W3C. XML Linking Language (XLink), 2001. W3C Recomendation available at http://www.w3.org/TR/xlink/.

[91] W3C. *XML Schema*, February 2002. See `http://www.w3.org/XML/Schema`.

[92] T. Wagner and S. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(2):980–1013, 1998.

[93] WellLogML DTD. Available at `http://www.posc.org/ebiz/WellLogML/`.

[94] Wired Minds. MindSuite XDB. http://xdb.wiredminds.com/.

[95] X-Hive Corporation. X-Hive/DB. http://www.x-hive.com.

[96] *Extensible Markup Language (XML) 1.0*, second edition, October 2000. W3C Recommendation, See `http://www.w3.org/TR/REC-xml`.

[97] XML editor products. Available at `http://www.perfectxml.com/soft.asp?cat=6`.

[98] XML Global. GoXML. http://www.xmlglobal.com.

[99] XMLmind XML Editor. Available at `http://www.xmlmind.com/xmleditor/`.

[100] xmlspy document editor. Available at `http://www.xmlspy.com/products_doc.html`.

[101] *XQuery 1.0: An XML Query Language*, November 2005. W3C Candidate Recommendation, See `http://www.w3.org/TR/xquery`.

[102] *XQuery 1.0 and XPath 2.0 Full-Text*, February 2005. W3C Working Draft, See `http://www.w3.org/TR/query-full-text`.

[103] Y. Xu and Y. Papakonstantinou. XSearch demo. http://www.db.ucsd.edu/People/yu/xsearch/.

[104] XYZFind Corporation. XYZFind server. http://www.xyzfind.com.

[105] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *VLDB Conference*, pages 087–1097, 2004.