Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Data & Knowledge Engineering 44 (2003) 299–322

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

XML queries and algebra in the Enosys integration platform

Yannis Papakonstantinou *, Vinayak Borkar, Maxim Orgiyan, Kostas Stathatos, Lucian Suta, Vasilis Vassalos, Pavel Velikhov

Enosys Software, San Diego, CA, USA

Received 3 July 2002; received in revised form 3 July 2002; accepted 3 July 2002

Abstract

We describe the Enosys XML integration platform, focusing on the query language, algebra, and architecture of its query processor. The platform enables the development of eBusiness applications in customer relationship management, e-commerce, supply chain management, and decision support. These applications often require that data be integrated dynamically from multiple information sources. The Enosys platform allows one to build (virtual and/or materialized) integrated XML views of multiple sources, using XML queries as view definitions. During run-time, the application issues XML queries against the views. Queries and views are translated into the XCQL algebra and are combined into a single algebra expression/plan. Query plan composition and query plan decomposition challenges are faced in this process. Finally, the query processor lazily evaluates the result, using an appropriate adaptation of relational database iterator models to XML. The paper describes the platform architecture and components, the supported XML query language and the query processor architecture. It focuses on the underlying XML query algebra, which differs from the algebras that have been considered by W3C in that it is particularly tuned to semistructured data and to optimization and efficient evaluation in a system that follows the conventional architecture of database systems.

Keywords: Enosys platform; XML; eBusiness applications

* Corresponding author.
E-mail address: yannis@enosysmarkets.com (Y. Papakonstantinou).

0169-023X/03/$ - see front matter © 2002 Published by Elsevier Science B.V.
PII: S0169-023X(02)00141-6

1. Introduction

eBusiness applications that support more efficient, tightly integrated business processes demand access and integration of up-to-date information from multiple distributed and heterogeneous information systems. Information integration is a significant challenge: the relevant data are split across multiple information sources, often owned by different organizations. The sources represent, maintain, and export the information using a variety of formats, interfaces and semantics. Many challenges arise:

(1) Data of different sources change at different rates, making the data warehousing approach to integration hard to develop and maintain. The Enosys platform resolves this challenge by being based on the on-demand mediator approach [5,9,29,33,41,46], according to which data are collected dynamically from the sources, in response to application requests. In addition, the Enosys Software data integration platform enables the applications to access and integrate information using a high-level, declarative XML query language (XML-QL). Processing XML query statements in a dynamic mediator approach poses query processing challenges, as we discuss next.

(2) The mediator has to decompose application requests into an efficient sequence of requests targeted to the sources. These requests have to be compatible with the query capabilities of the underlying sources. For example, if the underlying source is an XML file the mediator may only retrieve the file sequentially. All selections, joins, transformations on the data of the file will have to be done in the mediator's space. On the other hand, if the underlying source is a powerful SQL database the mediator can send to it SQL queries that delegate the selections and joins to the SQL query processor, hence providing efficiency: The amount of data that are retrieved from the database is much smaller. Consequently, the SQL query takes correspondingly less time to be evaluated.

The mediator needs to rewrite the plan in order to push the most efficient supported query to the underlying sources. In the algebra-based Enosys mediator query processor this requirement is addressed by having the rewriter/optimizer transform the algebraic expressions in a way that subexpressions are delegated to the sources.

(3) The requirements on the power of the supported view definition and query language and the extensibility of the system are important. First, different types of information reside in different systems, have different structure, and are usually in heterogeneous formats. The mediator has to enable and facilitate the resolution of the heterogeneities. This requires a view definition/query language that can take care of complex transformations. Furthermore, extensibility is required in order to allow easy interface with function libraries built in other programming languages, such as Java. Second, different applications use different XML views and queries, which structure the XML data as close as possible to the application needs. For example, we have found that XML views and queries that structure the data in a way that is "isomorphic" to the HTML structure of the Web-application lead to huge time savings in building web applications. However, providing the flexibility to produce structures that fit the application requirements requires a powerful language for selection, join, and transformation.

Addressing the three points above poses the challenge of optimizing complex queries over views of distributed information sources with limited query capabilities. The task is accomplished in the Enosys platform using an algebra-based processor.

Roadmap Section 2 presents related work. Section 3 presents the architecture and components of the overall Enosys integration platform. Section 4 presents the architecture of the mediator query processor and introduces the reader to the role of the algebra in the mediator's query processing. Section 5 describes XCQL, which is the XML query language used by the mediator. [1] Section 6 presents the XML query algebra. It illustrates how XCQL queries are mapped into algebraic expressions.

2. Related work

Data integration has been an important database topic for many years. Most of the early works focused on the integration of structured sources—primarily relational databases. A survey and summary of such works can be found in [21,27,44]. In 1990s the scope of data integration systems was extended to include the integration of non-structured sources and the "mediator" concept was created [46]. The Enosys mediator follows the architecture of earlier virtual view mediators, such as TSIMMIS [33], YAT [9], HERMES [41], Garlic [5], and the Information Manifold [21].

The query language of the Enosys mediator, XCQL, borrows from similar languages such as XML-QL [12], MSL [33], FLORID [15], Lorel [38,31], and YAT [9]. Its most immediate relative is the XMAS language [28]. XMAS is similar to XML-QL; the queries of both are based on a `WHERE` clause that produces bindings for its variables and a `CONSTRUCT` clause that inputs the bindings and produces the result. Their main difference is that the `CONSTRUCT` clause of XMAS enables grouping using an explicit group-by construct, which resembles the group-by structure of OQL [2], while XML-QL enables grouping by skolem functions. Note that skolem functions also allow the specification of graphs. (Graphs are simulated in XML by use of references.) The group-by structure of XMAS does not allow for the specification of graphs—only trees can be specified. The expressive power disadvantage of group-by is countered by the fact that it has a clean reduction into a corresponding algebraic operator, which is a critical point in the course of building an algebra-based query processing engine.

XCQL extends XMAS in a number of directions. First, it supports nested queries in both the `CONSTRUCT` and `WHERE` parts of the query. Second, it supports a wide range of operators, such as union and outerjoin, in the `WHERE` clause. More generally, XCQL supports null bindings for the query variables and defines corresponding semantics that enable queries that efficiently handle irregular data sets. Note that null bindings do not correspond to XML nulls. Instead, their meaning is that a variable may not be bound.

XCQL is translated into the XCQL algebra. The XCQL algebra is the cornerstone of query processing in the Enosys XMediator, similar to the role algebras have played in relational [20] and object-oriented databases [7,11]. Algebras were also designed for the nested relational [26,39] and complex value models [1]. Recently XML query algebras emerge as the underlying infrastructure of XML databases and mediators [8,19,28]. The common characteristic of those algebras is that the operators input and output sets of tuples. This should be contrasted with functional

[1] XCQL is now being replaced by XQuery. Nevertheless, the key characteristics of the underlying algebra remain the same and many ideas of XCQL, such as group-by support, are incorporated into the extended XQuery that Enosys implements.

programming-based XML algebras, such as [3,14], which serves as the core of the semantics of the emerging XQuery W3C XML querying standard [4]. In those approaches the operators input and output lists of XML elements. The tuple orientation, which is also present on object-oriented algebras, allows one to carry proven aspects of relational algebra into XML. For example, it facilitates the specification of multiple physical join operators as has been the case with relational algebra systems, where a join may be physically executed using any of multiple physical join operators [20].

The algebra has also been the base of an iterator model that minimizes the memory footprint of executing a plan and the response time required for partial evaluation of the result. The iterator model is coupled with features that enable arbitrary forms of navigation in the query result. Furthermore it allows a computation to stop and resume its execution later, even by another process.

The XCQL algebra relates to the OQL algebra [7,11] and XML algebras [8,19,28]. However, it has many distinguishing features, oriented towards enabling higher efficiency in the query processing engine:

1. The "null bindings" feature facilitates the efficient handling of irregularities in the data. Furthermore, it simplifies the algebraic equivalent of queries, by reducing the number of required nested plans.
2. There are multiple operators for the same or similar logical operations. For example, there is a main memory-based nested loops join, an index-based join, and a two-pass sort-merge join (similar to the one used in relational database systems [20]). Furthermore, there are typically multiple algebraic expressions that accomplish the same XML query. The redundancy in the algebra is justified by cost-based optimization considerations.

The component of the rewriting optimizer that is responsible for decomposing queries into queries that are sent to the sources and are commensurate with their abilities is based on the conceptual background set up in [30,34,35,40,45], which, in turn, are related to the background created by works on answering queries using views either in the relational model (see [22] for a comprehensive survey) or semistructured/XML models [10,16,36]. The system architecture of the capabilities-based rewriting component of the Enosys mediator is related to the ones of [8,24,25] in the sense that it is built around a rewriting optimizer, such as the one of Starburst [23].

Note that many of the query processing challenges of the mediator's query processor are also faced by systems that export an XML view of a single relational source [17,18,42,43].

On the commercial front, many data integration companies and corresponding systems have emerged during the last few years [6,32]. More recently, the adoption of XML and its perfect fit to integration applications has led to the emergence of XML-based information integration companies, such as Xyleme [47] and Nimble [13].

3. High-level platform architecture and components

The Enosys platform is based on the wrapper-mediator architecture, as shown in Fig. 1. The wrappers, also called XMLizers, access multiple, distributed, heterogeneous information sources

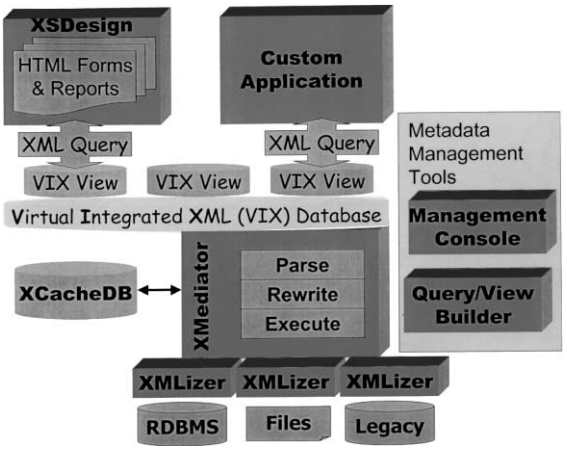


Fig. 1. The high-level architecture of the enosys platform.

and export Virtual XML views of them. The XMediator exports the Virtual Integrated XML (VIX) database, which consists of all the individual views exported by the wrappers. Then virtual integrated XML views can be built on top of the VIX database. The views organize information from the distributed sources into XML objects that conform to the application's needs. For example, to a marketplace application the integrated XML view can provide front-end access to an integrated catalog, where the heterogeneities between the suppliers' products are resolved, and the products are integrated and classified according to the needs of the marketplace. Each product object contains catalog data along with attributes from the pricing, delivery, and other databases. The views provide distribution transparency, i.e., the originating sources and methods of access are transparent to the application. For example, it is transparent to the application that the product specifications in the catalog come from a product database, while the pricing and delivery information may be coming from a customer relationship management system, which often provide customized pricing and delivery for each customer.

The virtual database and views enable the front-end applications, which may be the Enosys XQForms web form generator or custom applications, to seamlessly access distributed heterogeneous information sources as if they were a single XML database. In particular, the application can issue an XML query against either the XML database or the views. The query typically selects information from the views and structures it in ways that are convenient for the application. For example, a HTML application will create queries that structure the XML results in a way that easily translate into the target HTML pages. In the catalog example, if the resulting HTML page presents the data grouped by product family then the XML result will greatly facilitate the construction of the HTML page if the results are grouped by product family.

At a sufficient level of abstraction, when the application issues an XML query to a VIX database or view, the platform decomposes the XML query into requests that are directed to the sources. The source responses are assembled into the XML query result that is sent to the source. Keep in mind though that in practice the information exchange between the client, the mediator and the sources is more complicated, in order to support efficient demand-driven (lazy) query evaluation.

When the sources correspond to slow and static sources one may prefer to cach the XML view of those sources into the XML Store, which is the Enosys XML database. Typically, the data of slow and static sources are collected, integrated, and cached in advance, while the components originating at fast dynamic sources are collected dynamically. It is transparent to the application which pieces of the view originate from dynamic sources and which ones originate from XML Store.

The data integration server is accessible to applications through a query language API and a document object model (DOM)-based API. XCacheDB is an XML database, primarily used for caching purposes. XSDesign offers a web-base front-end generator for the easy construction of web/HTML-based query form and report templates. Finally, the platform includes management tools that enable the user to easily create and manage front-ends, view definitions, queries, and source connections. Each of the key components is described below.

3.1. XMLizers

There are two classes of XMLizers. The first class turn structured and semi-structured data into virtual XML views. Enosys Markets currently offers such XMLizers for

- Relational databases. Every JDBC-compliant database is supported. The virtual XML view exported by the XMLizing wrapper is a straightforward conversion of the tables and tuples of the database into an XML structure. Note that the XML view also includes an XML Schema of itself. The XMLizing wrappers produce the XML Schemas by reading the metadata tables of the database (also called system tables, or catalog tables in some RDBMS's.)
- XML files or sets of files. The XML Schema of its file is computed in one of three ways: If the file already refers to an XML Schema then this is the returned XML Schema. If the file already refers to a DTD then this DTD is converted into an XML Schema. If the file has no XML Schema or DTD associated with it then a schema is inferred using straightforward conventions.
- HTML files and sets of files. The HTML files are viewed as XHTML.
- Delimited text files. The XMLizing wrapper turns the text file into an XML file by following directions found in a configuration file about how to translate into XML the data that are found between commas, tabs, lines, or other symbols that we may wish to include in the set of delimiters.

The second class of XMLizers corresponds to *functions*. Functions may be invoked by the XCQL query statements. They have specific inputs and outputs. Each input and output may be a basic type, such as string, integer, etc. or an XML type. The standard library of Enosys includes functions that connect to WSDL-described Web services and SOAP interfaces. It is easy to see the similarity between WSDL specifications and the I/O specifications of our functions. Indeed, function specifications will be conformant to the WSDL format soon.

Additional XMLizer functions can easily be written for other data sources. Existing investments in XML information exchange (e.g., use of adapters by WebMethods or SeeBeyond) provide excellent leverage for the development of these XMLizer functions.

3.2. XMediator

The XMediator accesses the virtual views exported by the XMLizers, accesses the functions, and provides a virtual integrated XML view to the applications. The integrated view appropriately transforms and integrates the XML views of the information sources into XML that conforms to the target application needs. The transformation and integration is rapidly and concisely specified in the XML-QL XCQL, which is described in Section 5.

The XMediator allows queries directly on the VIX views. The query processor transforms the query to replace references and conditions on views to references and conditions on the actual data sources. The query is then parsed into a query plan and is optimized by the query rewriter. The decomposer chooses an efficient way of decomposing the optimized query plan into requests that are sent to the information sources. The plan is finally run by the execution engine, which sends the requests to the wrappers, collects the information, and composes it into the XML query result.

3.3. The XML store database

The Enosys Integration Platform uses the XML Store to cache VIX View's that correspond to slow or static sources. The caching can happen on demand or at regularly scheduled intervals. The XCacheDB is a native XML database in the sense that it stores XML and responds to XML queries with XML results. XCacheDB utilizes a JDBC compliant relational database for storage and query processing and is optimized for Oracle8i. The developer does not need to be aware of the underlying relational database. Nevertheless, XCacheDB offers management functionality that allows a developer/administrator to provide hints on how the data should be stored. The architecture of XCacheDB uses proprietary storage and query processing algorithms to deliver improvements in run-time efficiency.

3.4. The XSDesign web front–end generator

The XSDesign family of tools enables the rapid development of customized Web front–ends that access the integrated view. XSDesign [37] is designed to be used by the business analyst and provides forms for parametric querying of the data sources in the integrated view, summarization and navigation of large query results and query assistance in formulating and refining queries.

3.5. Metadata management tools

As it is the case with database servers as well, the Enosys Markets data integration platform provides a set of management tools for

- System administration.
- Management of views and management of the access rights on the views.
- Dynamic reconfiguration of the views without having to shut down the mediator server. This is a particularly useful feature when one has to cope with sources that may be out of function for a while.

The Enosys platform also includes a set of development tools, the most notable of which is the Query Builder. The Builder allows the graphical creation of XCQL view definitions and queries. The builder first imports the XML schemas of the information sources or the existing views. The user then uses a drag-and-drop interface and wizards to define joins, function invocations, filtering conditions and more on the input data. The builder also allows the user to graphically arrange the output data into a different XML schema by specifying mappings to the input schemas, groupings, and creation of new elements.

4. Query processor architecture

In the rest of the paper we focus on the XMediator, its query language and the underlying algebra. As we described earlier, the XMediator inputs an XCQL query, an optional XCQL view, a description of the sources, and optional XML Schemas of the sources. It returns the XML query result.

Query processing is based on the set-oriented XCQL algebra. In particular, incoming XCQL queries are translated into XCQL algebra expressions by the *translator* module of the query processor (see Fig. 2). Consequently, the *rewriting optimizer* applies a series of rewritings on the algebra expression in order to optimize it. For example, selection and projection conditions are pushed down, join operators and join orders are selected according to characteristics of the sources, such as presence of indices, and parts of the algebraic query expression are delegated to the underlying source.

At a sufficient level of abstraction, the *plan execution engine* receives the query plan, sends requests/queries to the wrappers, and combines the results received from the XMLizers into the

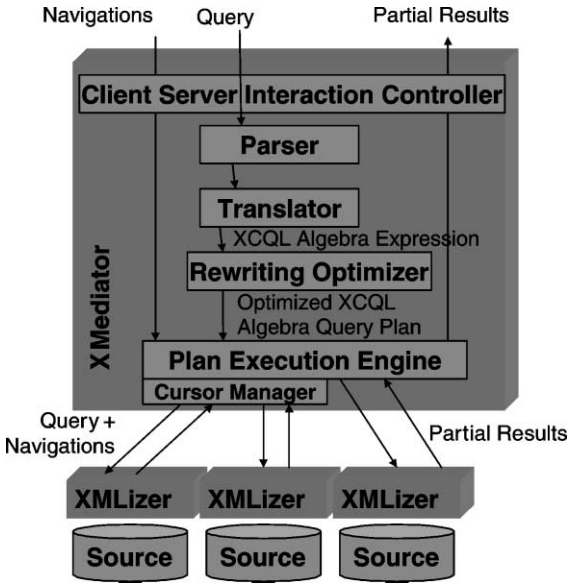


Fig. 2. The query processor architecture.

XML result of the input query. In practice, the query result object is not fully materialized immediately. Instead, query evaluation is driven by the client navigations, as explained below.

4.1. Navigation-driven evaluation and partial result evaluation

A large set of requirements posed by Web applications are covered by the iterator model of executing the operators of the algebra and the ability of the query plans to perform navigation-driven result evaluation. Web-based applications are often accessed by large numbers of users who generate correspondingly large numbers of requests to the mediator. In this context, good use of available resources (main memory, bandwidth, resources of underlying sources) is of high importance. In addition, result computations need to be client-driven and typically partial; the typical Web user tends to issue queries that have very large results. After navigating into a small part of the result the user may decide that she does not need to see the whole result. In order to avoid unecessary costs it is important that the mediator produces only what is being requested by the client.

The mediator allows "just-in-time" generation of the necessary (parts of the) query result, by integrating result object navigation and querying. In particular, the mediator server process (depending upon appropriate configuration by the client) may only send parts of a query result to the client process. A partial result has multiple lists of elements that are incomplete, as in Fig. 3. In place of the missing parts of the results, appropriate tokens are included that the client may send back to the server if it needs one of the missing parts. The tokens contain information needed for the mediator to produce missing parts. For example, if the client navigates to the second child of customer "John Smith", the `Token 2` is passed to the mediator and the mediator produces another partial result, which leads to a tree such as the one of Fig. 4.

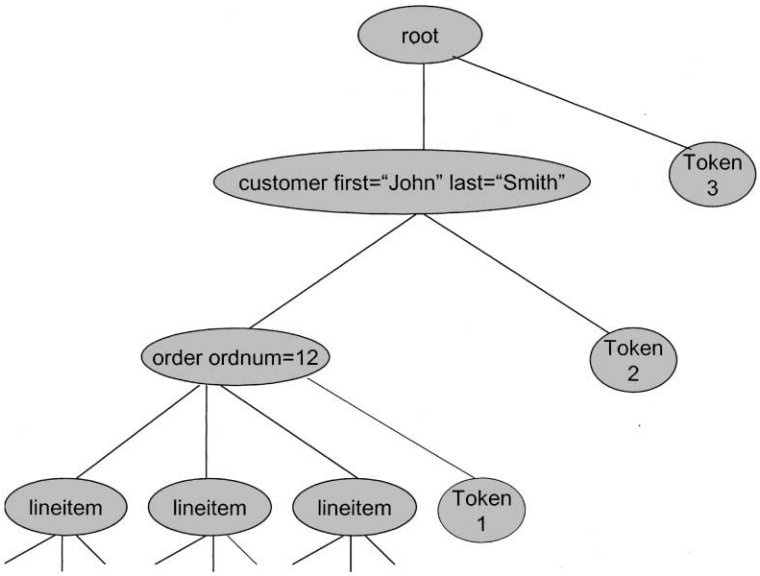


Fig. 3. A partial result.

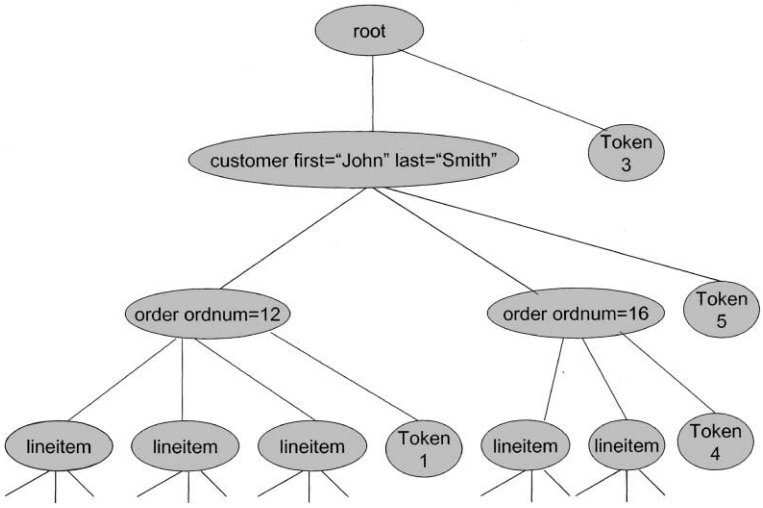


Fig. 4. A larger partial result.

Applications are shielded from the existence of tokens: a thin client DOM implementation is in charge of passing the appropriate tokens to the mediator server. The client navigates in the result fragment unaware that parts of the result are not yet in client memory. If the client happens to navigate into a missing part, the library will send the relevant token to the server and fetch the required fragment.

A key challenege in optimizing navigation-driven query evaluation is the choice of the size and shape of the produced fragments. At the one extreme, one may choose each node of the data model to be a fragment and encode the relevant token in the node itself. This approach, described in [28], is elegant but it is vert inefficient in the number of round-trips that will be needed between the client and the server. Moreover, it penalizes the server with unacceptable overhead in creating tokens. Instead, the mediator employs the *Client Server Interaction Controller* module to choose the size and shape of the fragment that will be returned to the client. The algorithm takes as input configuration parameters provided by the client.

The Mediator execution engine supports CLSIC through a pipelined, iterator-based execution model, which additionally allows the computation state of operators to be exported and imported. Each operator can respond to a call for the "next" tuple (as we discuss in the next section, XAlgebra is tuple-oriented). Moreover, each operator enables the production of tokens by being able to produce information about its state on command. This state information is encoded by CLSIC in tokens. Each operator is able to reproduce a prior state and continue its computation from that point on. Upon receipt of a token, CLSIC produces and imposes the appropriate state on each operator.

Note that, depending on the capabilities of the underlying sources for on-demand computation, the mediator and the XMLizers set up and invoke appropriate access methods of these sources, such as SAX calls or SQL cursors. For example, assume that a query produces `customer` elements, from the `customer` table of an underlying relational source. When the client issues the query, the corresponding XMLizer establishes a cursor, using the *cursor manager* module. As the

mediator requests more results the cursor reads more tuples. A semantic description of the cursor's status may even appear in tokens that are exported.

5. The XCQL query language

We introduce next the XCQL query language. We start with an example that illustrates the basics of the language and we proceed with examples that highlight its special features. We emphasize the support of NULL values.

An XCQL query consists of two parts or clauses, as described below:

- The WHERE clause specifies the elements expected in the input and the conditions they should satisfy. It also assigns or binds input elements and values to variables.
- The CONSTRUCT clause defines the arrangement of the variable bindings supplied by the WHERE clause in a new XML document. Nested queries are also allowed in the CONSTRUCT clause.

For our running example, let us assume that there are two sources, exporting "homes" and "schools" information, as illustrated in Figs. 5 and 6.

```
<db>
  <homes>
    <home id = "h1">
      <bedrooms> 4 </bedrooms>
      <zip> 92122 </zip>
      <descr> Duplex house, quiet neighborhood </descr>
      <ocean_view />
    </home>
    <home id = "h2">
      <bedrooms> 2 </bedrooms>
      <zip> 92122 </zip>
      <descr> Awesome beach house </descr>
      <ocean_view />
      <fireplace />
    </home>
    <home id = "h3">
      <bedrooms> 3 </bedrooms>
      <zip> 92123 </zip>
      <descr> House in the suburbs </descr>
      <fireplace />
      <washer />
      <dryer />
    </home>
  </homes>
</db>
```

Fig. 5. Sample of the "homes" source.

```
<db>
  <schools>
    <school id = "s1">
      <zip> 92122 </zip>
      <nts> 430 </nts>
    </school>
    <school id = "s2">
      <zip> 92123 </zip>
      <nts> 500 </nts>
    </school>
    <school id = "s3">
      <zip> 92123 </zip>
      <nts> 600 </nts>
    </school>
  </schools>
</db>
```

Fig. 6. Sample of the "schools" source.

XCQL queries can combine data from multiple sources with ease and create complex output structures. In the example of Fig. 7, a WHERE clause joins information from two sources, and a CONSTRUCT clause creates new elements and nests the source elements appropriately. In particular, the XCQL query accesses the sources homes and schools and finds homes that are in the same zip code as schools with a national test score $\geqslant 500$. For each such home, the query creates a "top_home" element that contains the specific home and the qualifying schools of the same neighborhood.

The WHERE clause consists of two tree patterns and two condition clauses. In general, a query may have multiple tree patterns, which syntactically resemble the tree structure of the input documents and are annotated with variables. Variables are denoted by the $ symbol. Intuitively, each tree pattern is matched against the XML source, real or virtual, and each variable is bound to an element or element content of the source. As a result, a tree pattern produces a list of all matching combinations of variables to source bindings. Each such combination is termed a binding tuple. The homes tree pattern of the example query produces all binding tuples

$$(\$H = h, \$Z = z_1)$$

such that the following conditions hold:

- The root of the XML document at homes is named db.
- The root contains a homes element.
- The homes element contains a home element, h.
- The home element h, in turn, contains a "zip" element.
- The content of zip is z_1.

The tree patterns generate binding tuples, and the condition clauses apply join and selection conditions, as in SQL. The first tree pattern produces all possible binding tuples ($\$H = h$, $\$Z1 = z_1$) for variables $\$H$ and $\$Z1$. The second tree pattern produces all binding tuples

```
CONSTRUCT
<homes_around_good_schools>
  <top_home>
    $H
    <schools>
      $S {$S}
    </schools>
  </top_home> {$H}
</homes_around_good_schools>

WHERE
% Start of ‘‘homes" tree pattern
<db>
  <homes>
    $H:<home>
      <zip> $Z1 </zip>
    </home>
  </homes>
</db>
in "homes.xml"
% End of tree pattern on ‘‘homes"
AND
% Start of ‘‘schools" tree pattern
<db>
  <schools>
    $S:<school>
      <zip> $Z2 </zip>
      <nts> $nts </nts>
    </school>
  </schools>
</db>
in "schools.xml"
% End of tree pattern on ‘‘schools"

% Condition clauses
AND $Z1 = $Z2
AND integer($nts) >= 500
```

Fig. 7. A typical join-and-construct query.

($\$S = s, \$Z2 = z_2, \$nts = n$) for variables $\$S$, $\$Z2$, and $\$nts$. Together, they produce all possible tuples of the form

$$(\$H = h, \$Z1 = z_1, \$S = s, \$Z2 = z_2, \$nts = n)$$

by combining every tuple for the first tree pattern with every tuple for the second. Finally, the two condition clauses select only the tuples for which $z_1 = z_2$ and $n \geqslant 500$. Using these binding tuples, the CONSTRUCT clause generates a "homes_around_good_schools" root element, which contains a list of "top_home" elements. The group-by construct {$H} requires that one "top_home" element is constructed for each binding h of $H. This is specified by the {$H} construct after the

```
<homes_around_good_schools>
  <top_home>
    <home id="h3">
      <bedrooms>3</bedrooms>
      <zip>92123</zip>
      <descr>House in the suburbs</descr>
      <fireplace/>
      <washer/>
      <dryer/>
    </home>
    <schools>
      <school id="s2">
        <zip>92123</zip>
        <nts>500</nts>
      </school>
      <school id="s3">
        <zip>92123</zip>
        <nts>600</nts>
      </school>
    </schools>
  </top_home>
</homes_around_good_schools>
```

Fig. 8. Result of the example query.

closing of the top_home element in the CONSTRUCT clause. In other words, the {$H} construct groups top_home elements by *h*. Within each top_home element, along with the binding for $H, a schools element is also generated. In a schools element, all bindings, *s*, of $S for the specific binding of $H are listed. Fig. 8 illustrates the XML result.

5.1. Optional patterns and the use of NULLs in XCQL

Optional tree patterns allow for matching with sources that have irregular structure, characterized by missing elements or missing element content. Given that structure irregularities is a prime feature of semistructured data and XML, it is obvious that the feature is very useful. Like regular tree patterns, an optional tree pattern matches elements and their components to variables, if matching elements are present in the data. A regular tree pattern requires that all variables in the pattern are bound to some content or element in the source before it produces a binding tuple. However, an optional pattern produces binding tuples even when the specified pattern is missing from the data. In such cases, the variables in the pattern receive empty (NULL) bindings. Optional patterns are annotated with a question mark (?).

The example of Fig. 9 illustrates regular tree patterns. The user collects all home ID's from the homes source regardless of the presence of an ocean view. At the same time, if a home has an ocean view, the "ocean_view" object should be included in the output. In other words, the ocean_view subelement is optional.

Before we proceed with explanations on optional patterns let us note that we used the attribute pattern "`id = $hid`" in the WHERE part and a syntactically identical construct was used in the

```
CONSTRUCT
<ans>
  <home id = $hid>
    $ocean_view {$ocean_view}
  </home> {$hid}
</ans>
WHERE
<db>
  <homes>
    <home id = $hid >
      $ocean_view : <ocean_view/> ?
    </home>
  </homes>
</db>
in "homes.xml"
```

Fig. 9. Example of optional tree patterns.

```
<ans>
  <home id = "h1">
    <ocean_view />
  </home>
  <home id = "h2">
    <ocean_view />
  </home>
    <home id = "h3"/>
</ans>
```

Fig. 10. Result of optional tree pattern query.

CONSTRUCT clause. Attribute patterns in the WHERE clause are used to place conditions on XML element attributes and bind attribute names and values to variables, as is the case with XML-QL, as well. Attribute patterns in the CONSTRUCT clause are used to include attributes in the results.

The result of the query is illustrated in Fig. 10. The optional pattern "<ocean_view /> ?" is used to signify that the pattern does not have to necessarily match any elements in the source. If the <ocean_view/> element is encountered within the <home> element, the variable $ocean_view is bound to it. Otherwise, the variable binds to NULL.

Remark on passing NULLs to predicates and functions: Most of the predicates in the standard predicate library of the XMediator will turn false if one of their arguments is NULL. For example, "5 > *NULL*" will return false and, in SQL spirit, "NULL = NULL" will also return false. However, one can build predicates that can return "true" even if they are given NULLs. The "isnull()" predicate is one of them.

The XCQL iterators do not construct elements that correspond to tuples that have NULL bindings for the variables of the iterator. In other words, if a binding tuple has a NULL binding for one of the variables in a group-by list, no element is generated for the particular tuple.

6. The XCQL algebra

In this section we describe the XCQL algebra, which is used to evaluate the queries in the XMediator. Unlike the XML Query Algebra and the XQuery Core, which are algebras based on functional languages, the XCQL algebra is tuple-oriented and draws on the relational and nested relational algebras. Tuple orientation allows the construction of an iterator model, which extends the iterator model of relational databases and enables navigation-driven partial evaluation. In addition, it enables a better fit with the underlying relational databases, which naturally return tuples. Finally, it introduce join operators, which open the gates for cost-based optimization.

A close relative of the XCQL algebra is the XMAS algebra [28]. There are some obvious differences between the XCQL algebra and XMAS, stemming from the fact that the latter addresses "abstractions". For example, it omits direct support of attributes. However, there are also substantial differences that affect query processing in the mediator:

1. The XCQL algebra has a "relational query" operator that is responsible for turning a part of the plan (which typically corresponds to the `FOR` and `WHERE` parts of the XQuery) into an SQL query that returns tuples of bindings.
2. The XCQL algebra supports and OQL-like group-by and nested plans.
3. The XCQL algebra supports NULLs. Its ability to support NULLs becomes useful when we evaluate query language constructs that explicitly involve NULLs, such as optional patterns and outerjoins. It is also useful in the evaluation of other language constructs, such as nested queries, which can be translated into constructs involving null operators—and such a translation may lead to more efficient plans.

The input and output of most operators is a set of tuples $\{b_i | i = 1, \ldots, n\}$. Each tuple $b_i \equiv [\$var_1 = val_1^i, \ldots, \$var_k = val_k^i]$ consists of variable–value pairs, also referred to as bindings. We say that the variable $\$var_j$ is bound to the value val_j^i in the tuple b_i if the pair $\$var_j = val_j^i$ appears in b_i. All input (resp. output) tuples of an operator have the same list of variables and no variable appears more than once in a tuple. Each value v_j^i can either be a constant, NULL, a single element, a list of elements or a set of tuples, recursively.

In the rest of the paper $b_i.\$x$ is used to represent the value to which the variable $\$x$ is bound to in the tuple b_i. The notation $b_j = b_i + (\$v = w)$ means that the tuple b_j contains all the bindings from the tuple b_i and the binding $\$v = w$, and $b_k = b_i + b_j$ means that the tuple b_k contains all the bindings from the tuples b_i and b_j. The notation $l[e_1, \ldots, e_k]$ is used to describe an XML element l with subelements $e_1, \ldots, e_k$.

A subset of the operators of the XCQL algebra is described next:

- $source_{\&srcid,[\$V_1,\ldots,\$V_n],query?}()$. The *&srcid* parameter identifies the source/wrapper from which bindings for the variables $[\$V_1, \ldots, \$V_n]$ will be obtained. The number of variables in the list is dependent on the type of the source. In the simplest case, which is the case of an XML file source, the list of variables is a single variable $[\$V]$ and the result is the singleton list of tuples $\{[\$V = r]\}$, where r is the root node of the XML file.
  In the XML file case the *query* is missing. Note that whenever we include the symbol "?" next to a parameter we mean that the parameter may be missing. In general, the *plan* is an algebra

expression that is evaluated by the underlying wrapper and returns a list of tuples of the form $[\$V_1 = v_1, \ldots, \$V_n = v_n]$. For example, in the case of a relational database source the *plan* can be an expression which corresponds to a combination of selections and joins.

- $getD_{\$V.p\to\$X,d,o}(I)$. The "get descendants" operator is used to obtain bindings for the fresh variable \$X by navigating using the path expression p into the bindings of the variable $\$V$ of the input I. The parameters d (called "NULL disqualify") and o (called "optional") are binary flags that guide the behavior of the operator when the path p cannot be followed or when the binding of $\$V$ is NULL. In the basic case, where $d = 1$ and $o = 0$ the output of the $getD_{\$V.p,\$X,1,0}$ operator on input $I = \{b_i | i \in 1, \ldots, n\}$ is as follows: Suppose $b_i = [\$V = v_i, \ldots]$. Let $Y_i = \{y_{ij}\}$, where y_{ij} is reachable from the node v_i by a path p' such that the labels on this path satisfy the path p. Then the output is the set of binding lists (tuples) defined by

  $$getD_{\$V.p\to\$X,1,0}(I) = \{b_i + (\$X = y) \mid b_i \in I, y \in Y_i\}$$

  Basically this means for every tuple in the input, for the node n bound to the variable $\$V$, we find the set of nodes reachable from n by a path satisfying the regular expression p, and these nodes are the bindings for the variable $\$X$. Every pair of bindings for $\$V$ and $\$X$ is inserted into a new output tuple and bindings for all the other variables in the corresponding input tuple (the tuple from which we took the value of $\$V$) are copied into the output tuple.
  Note that if $\$V = NULL$ then the corresponding tuple is disqualified. Similarly, if Y_i is the empty set the corresponding input tuple b_i is disqualified. The other cases are defined similarly.

Example 6.1. Consider the set of input tuples

$$\begin{aligned} B = \{ \quad & [\$H : home[descr[neat], descr[clean], beds[2], \ldots]], \\ & [\$H : home[bed[3], \ldots]], \\ & [\$H : NULL] \\ \} & \end{aligned}$$

Then it is

$$\begin{aligned} getD_{\$H.descr\mapsto\$X,1,0}(B) = \{ \quad & [\$H : home[descr[neat], descr[clean], beds[2], \ldots], \$X = descr[neat]], \\ & [\$H : home[descr[neat], descr[clean], beds[2], \ldots], \$X = descr[clean]] \\ \} & \end{aligned}$$

- $assign_{f(\$V_1,\ldots,\$V_n)\mapsto\$X}$. In the ideal case, for each input tuple $b_i = [\$V_1 = v_1, \ldots, \$V_n = v_n, \ldots]$, the assignment operator outputs the tuple $b_i + (\$X = f(v_1, \ldots, v_n))$, where f is some function understood by the query processor.
  However, it is often the case that f cannot accept NULL values for some (or even all) of its inputs. It may also be the case that f may not be defined for some of its inputs. The function may respond in one of two ways in such cases:
  - The function f returns the special result "FAIL", which in practice is actually communicated to the mediator by a Java exception. The assign operator disqualifies b_i from the output.
  - The function f returns NULL. Then the tuple $b_i + (\$X = NULL)$ is included in the output.

The mediator provides to a developer functions functionality that facilitates the implementation of each one of the two behaviors above.

Example 6.2. Consider the addition function +:

$$assign_{+(\$V_1,\$V_2)\mapsto\$X}(\{[\$V_1 = 1, \$V_2 = 2, \$V_3 = a], [\$V_1 = 0, \$V_2 = 3, \$V_3 = b]\}$$
$$= \{[\$V_1 = 1, \$V_2 = 2, \$V_3 = a, \$X = 3], [\$V_1 = 0, \$V_2 = 3, \$V_3 = b, \$X = 3]\})$$

- $I_1 \cup I_2$. The union operator is identical to union (without duplicate elimination) of relational systems, in the case where the "schemas" of I_1 and I_2 are identical. However, if the tuples of I_1 are of the form

  $$b_1 = [\$V_1 : v_1, \ldots, \$V_k : v_k, \$V_1^1 : v_1^1, \ldots, \$V_l^1 : v_l^1]$$

  and the tuples of I_2 are of the form

  $$b_2 = [\$V_1 : v_1, \ldots, \$V_k : v_k, \$V_1^2 : v_1^2, \ldots, \$V_l^2 : v_m^2]$$

  then each tuple b_1 of I_1 is propagated in the output after it has been expanded with NULL bindings for the variables $\$V_1^2, \ldots, \$V_l^2$, i.e., the following tuple is output

  $$b_1 + (\$V_1^2 : NULL, \ldots, \$V_l^2 : NULL)$$

  We treat the tuples of I_2 similarly.
- $\pi_{[\$V_1,\ldots,\$V_n]}(I)$. The projection operator π is identical to the non-duplicate-removing duplicate elimination operator of relational systems. For every tuple $b = [\$V_1 : v_1, \ldots, \$V_n : v_n, \ldots] \in I$, the operator outputs exactly one tuple $[\$V_1 : v_1, \ldots, \$V_n : v_n]$.
- $\sigma_\theta(I)$ The output of the select operator σ_θ is the set of input tuples that satisfy the condition θ.

  $$\sigma_\theta(I) = \{b | b \in I, \theta(b) \equiv true\}$$

  The condition θ is a boolean expression involving built-in functions and predicates, such as $=, \neq, >, <, \geqslant, \leqslant$, and possibly other functions and predicates that have been interfaced to the XMediator. The arguments of the functions and the predicates are constants or variables that appear in the schema of I.
  The evaluation of the condition θ when NULL bindings are present is identical to the evaluation of SQL WHERE clauses in the corresponding situation. In particular, predicates may return one of the three values T (true), F (false), or $\perp$ (uncertain). The built-in predicates $=, \neq, >, <, \geqslant, \leqslant$ always return $\perp$ when one or both of their arguments are NULL. Other predicates may exhibit other behavior. Indeed, some predicates have input flags that control their response in the case that their argument(s) is (are) NULL.
  The boolean operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$), evaluate the boolean expression using the following list of equations, which is identical to the way SQL evaluates boolean expressions.

$\wedge$	T	F	$\perp$
T	T	F	$\perp$
F	F	F	F
$\perp$	$\perp$	F	$\perp$

$\vee$	T	F	$\perp$
T	T	T	T
F	T	F	$\perp$
$\perp$	T	$\perp$	$\perp$

$\neg$	
T	F
F	T
$\perp$	$\perp$

  Eventually, if the boolean expression results into T the corresponding tuple qualifies. It it results into a F or $\perp$ the boolean expression is disqualified.

- $join_{\theta}^{M?}(I_1, I_2)$ The $join_{\theta}$ operator has a single parameter θ which is a boolean predicate. The join operator is like the relational join operator except that it operates on binding lists instead of relational tuples, with variables taking the place of attributes. So the output of the $join_{\theta}(I_1, I_2)$ operator is

  $$join_{\theta}(I_1, I_2) = \{b_1 + b_2 \mid b_1 \in I_1, b_2 \in I_2, \theta(b_1, b_2) \equiv true\}$$

  The condition θ is of the form $\$v_1$ *op* $\$v_2$ where v_1, v_2 are variables and *op* is one of $=, <, \leqslant, >, \geqslant$. The condition is true for a given pair of binding lists b_1, b_2 if $b_1.\$v_1$ is bound to a leaf node whose value is x_1, $b_2.\$v_2$ is bound to a leaf node whose value is x_2, and x_1 *op* x_2 is true.
  The optional annotation M dictates one of the following join execution methods:
  - $M = m$ is applicable on equijoins only. It assumes that the inputs I_1 and I_2 are sorted on the join attribute and a merge join is performed at the XMediator. Note that there is no sort-merge join operator that does both the sorting and the merging, but nevertheless a sort-merge joins are done by inserting order-by operators that sort the arguments of the merge join. The reason that the sort is decoupled from the merge is that in this way the optimizer has the freedom to delegate the order-by to the underlying source, if such a delegation is supported by the underlying source and reduces the overall cost.
  - $M = i$ is applicable when θ involves equality and inequality conditions. The XMediator builds an index on I_2, based on the attribute(s) of I_2 involved in the join.
  - $M = l$ is applicable to all kinds of conditions θ. The XMediator stores I_2 and matches incoming tuples of I_1 against the I_2 copy.
- A semijoin operator is defined in the obvious way.
- $outerjoin_{\theta}^{M?}(I_1, I_2)$. The outerjoin operator is defined in the same way that is defined in relational systems. It is very useful to integration applications for two reasons: First, it is an operation that naturally arises when one fuses together data that arrive from distributed databases where there is no guarantee that information on entity x found in the first source will find matching information on x in the second source. Second, the outerjoin operator provides a rewriting of queries that involve nested plans, as we discuss later.

6.1. Grouping and nested plans

- $groupBy_{[\$G_1,\ldots,\$G_n],[\$P_1,\ldots,\$P_m]\to\$P}(I)$. The group-by operator partitions the input I into sets of tuples that agree on the values of the variables in the group-by list $[\$G_1, \ldots, \$G_n]$. (It is easy to see the influence of OQL's group-by.) One partition is created for each unique binding $[\$G_1 : g_1, \ldots, \$G_n : g_n]$ of the variables $\$G_1, \ldots, \$G_n$. The output consists of one tuple $[\$G_1 : g_1, \ldots, \$G_n : g_n, \$P : p]$ for each partition, where p is called the partition set is the tuple set defined by

  $$\pi_{\$P_1,\ldots,\$P_m}\sigma_{\$G_1=g_1\wedge\cdots\wedge\$G_n=g_n}(I)$$

  Typically partition sets are passed to the "apply" operator, described next, which applies algebra expressions on them.
- $apply_{\$P,\$plan,\$X\to\$L}(I)$. The "apply" operator has three parameters: A plan *plan*, a variable $\$P$ that binds to a partition set and an output variable $\$L$. The apply operator is well defined only when the binding p of $\$P$ is a set of tuples. Furthermore, it is assumed that the result $plan(p)$ of

$$
\begin{array}{l}
crElt_{\texttt{homes\_around\_good\_schools},\$4\mapsto\$Result}(\\
\ apply_{\$3,p_1\mapsto\$4}(\\
\ \ groupBy_{[]\mapsto\$3}(\\
select_{int(\$NTS)\geq 500}(\\
\ join_{\$Z1=\$Z2}(\\
\ \ getD_{\$H,zip/\_\mapsto\$Z1}(\\
\ \ \ getD_{\$1,db/homes/home\mapsto\$H}(\\
\ \ \ \ source_{\texttt{'homes.xml'},[\$1]})), \\
\ \ getD_{\$S,nts/\_\mapsto\$NTS}(\\
\ \ \ getD_{\$S,zip/\_\mapsto\$Z2}(\\
\ \ \ \ getD_{\$1,db/schools/school\mapsto\$S}(\\
\ \ \ \ \ source_{\texttt{'schools.xml'},[\$2]})))))))
\end{array}
\qquad
p_1 : \left\{
\begin{array}{l}
listify_{\$11\mapsto\$4}(\\
\ crElt_{\texttt{top\_home},\$10\mapsto\$11}(\\
\ \ cat_{\$9,\$8\mapsto\$10}(\\
\ \ \ crList_{\$H\mapsto\$9}(\\
\ \ \ \ crList_{\$7\mapsto\$8}(\\
\ \ \ \ \ crElt_{\texttt{schools},\$6\mapsto\$7}(\\
\ \ \ \ \ \ apply_{\$5,p_2\mapsto\$6}(\\
\ \ \ \ \ \ \ groupBy_{[\$H]\mapsto\$5}(\\
\ \ \ \ \ \ \ \ nestedSrc_{\$3})))))))))
\end{array}
\right\}
\qquad
p_2 : \left\{
\begin{array}{l}
listify_{\$S\mapsto\$6}(\\
\ nestedSrc_{\$5})
\end{array}
\right\}
$$

Fig. 11. The XCQL algebra for the query of Fig. 7.

applying the plan on p is a single tuple with a single attribute. Let us assume this attribute is named $\$A$. Typically, the *plan* has a "listify" operator on top, which guarantees that the output is a single tuple with a single attribute. Formally,

$$apply_{\$P,\$plan\to\$L}(I) = \{b + (\$L = res) | b \in I, res = plan(b.\$P)\}$$

- $listify_{\$X\mapsto\$L}(I)$. The "listify" operator collects all bindings of the variable $\$X$. The result is a single tuple, having the single variable $\$L$. Typically, the variable $\$L$ is annotated to be the variable that has the result of the algebraic plan. The listify operator requires the variable $\$X$ to bind to nodes. It discards all NULL bindings.

$$listify_{\$X\mapsto\$L}(I) = \{[\$L : \text{list}[\{e \mid [\$X : e, \ldots] \in I, e \neq NULL, e \neq \text{list}[\ldots]\}]]\}$$

- $nestedSrc_{\$x}()$. The "nested source" operator has only one parameter, and may be used only in nested plans, i.e. plans which appear as a parameter to an apply operator in some other plan. It functions mainly as a placeholder. The output of the $nestedSrc_{\$x}$ operator is the set of binding lists to which the variable $\$x$ is bound to in the current tuple in the outer plan (the plan in which the corresponding *apply* operator appears).

Finally, the construction operators *crElt*, *cat* and *crList* are responsible for creating news elements, concatenating lists, and turning single elements into lists respectively. The operators are illustrated in the following example.

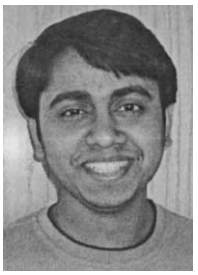
Example 6.3. The algebra for the query/view of Fig. 7 appears in Fig. 11. Notice the special variable/attribute $\$Result$ that carries the query result in its unique binding. The main plan has a bottom part that corresponds to the `WHERE` clause and an upper part that corresponds to the `CONSTRUCT` clause. All the bindings of the `WHERE` clause are grouped into a single group, which binds to \$3 and becomes the input of the nested plan p_1. The nested plan p_1 groups its input by homes ($\$H$). For its group it creates a list of schools \$6 using the nested plan p_2. Then by using the construction operators it organizes the bindings of $\$H$ and \$6 into `top_home` and `schools`.

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, Reading, MA, 1995.
[2] F. Banchilhon, S. Cluet, C. Delobel, A query language for the o_2 object-oriented database system, in: DBPL, 1989, pp. 122–138.
[3] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. in: Proceedings of ACM SIGMOD, 1996.
[4] D. Chamberlin et al., XQuery 1.0: An XML query language. Available at <http://www.w3.org/TR/xquery/>.
[5] M.J. Carey et al., Towards heterogeneous multimedia information systems: the garlic approach, in: Proceedings of RIDE-DOM Workshop, 1995, pp. 124–131.
[6] Callixa Inc. Callixa's Integration Server. Available at <http://www.callixa.com/>.
[7] V. Christophides, S. Cluet, G. Moerkotte, Evaluating queries with generalized path expressions, in: H.V. Jagadish, I.S. Mumick (Eds.), Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data Montreal, Quebec, Canada, 4–6 June, 1996, ACM Press, New York, 1996, pp. 413–422.
[8] V. Christophides, S. Cluet, G. Moerkotte, J. Simeon, On wrapping query languages and efficient xml integration, in: ACM SIGMOD Conference, 2000, pp. 141–152.
[9] S. Cluet, C. Delobel, J. Simeon, K. Smaga, Your mediators need data conversion! in: Proceedings of the 1998 ACM SIGMOD international conference on management of data, 1998, pp. 177–188.
[10] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi, Rewriting of regular expressions and regular path queries, in: ACM PODS Conference, 1999, pp. 194–204.
[11] S. Cluet, G. Moerkotte, Nested queries in object bases, in: C. Beeri, A. Ohori, D. Shasha (Eds.), Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages—Object Models and Languages, Workshops in Computing, Springer, Berlin, 1993, pp. 226–242.
[12] A. Deutch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, XML-QL: A query language for XML. Submission to W3C. Latest version available at <http://www.w3.org/TR/NOTE-xml-ql>.
[13] D. Draper, A.Y. Halevy, D.S. Weld, The Nimble integration engine, in: ACM SIGMOD Conference, 2001.
[14] P. Fankhauser et al., XQuery 1.0 formal semantics. Available at <http://www.w3.org/TR/query-semantics/>.
[15] J. Frohn, R. Himmeröder, P.-Th. Kandzia, G. Lausen, Christian Schlepphorst. FLORID: A Prototype for F–Logic, in: Proceedings of ICDE Conference, 1997.
[16] D. Florescu, A. Levy, D. Suciu, Query containment for conjunctive queries with regular expressions, in: ACM PODS Conference, 198, pp. 139–148.
[17] M. Fernandez, A. Morishima, D. Suciu, Efficient evaluation of xml middle-ware queries, in: ACM SIGMOD Conference, 2001.
[18] M. Fernandez, A. Morishima, D. Suciu, W. Chiew Tan, Publishing relational data in xml: the silkroute approach, IEEE Data Engineering Bulletin 24 (2) (2001) 12–19.
[19] L. Galanis et al. Following the paths of XML data: an algebraic framework for XML query evaluation. Available at <http://www.cs.wisc.edu/niagara/>.
[20] H. Garcia-Molina, J. Ullman, J. Widom, Database Systems: The Complete Book, Prentice Hall, Englewood Cliffs, NJ, 2001.
[21] A. Gupta, Integration of Information Systems: Bridging Heterogeneous Databases, IEEE Press, New York, 1989.
[22] A. Halevy, Answering queries using views: A survey, VLDB Journal (2001).
[23] L. Haas, J. Freytag, G. Lohman, H. Pirahesh, Extensible query processing in Starburst, in: Proceedings of ACM SIGMOD Conference, 1989, pp. 377–388.
[24] L. Haas, D. Kossman, E. Wimmers, J. Yang, An optimizer for heterogeneous systems with non-standard data and search capabilities, IEEE Data Engineering Bulletin 19 (1996) 37–43, Special issue on query processing for non-standard data.
[25] L. Haas, D. Kossman, E. Wimmers, J. Yang, Optimizing queries across diverse data sources, in: Proceedings of VLDB, 1997.
[26] H.F. Korth, M.A. Roth, Query languages for nested relational databases, in: Nested Relations and Complex Objects in Databases, Springer-Verlag, Berlin, 1989, pp. 190–204.

[27] W. Litwin, L. Mark, N. Roussopoulos, Interoperability of multiple autonomous databases, ACM Computing Surveys 22 (1990) 267–293.
[28] B. Ludascher, Y. Papakonstantinou, P. Velikhov, Navigation-driven evaluation of virtual mediated views, in: Proceedings of EDBT Conference, 2000.
[29] A. Levy, A. Rajaraman, J. Ordille, Querying heterogeneous information sources using source descriptions, in: Proceedings of VLDB, 1996, pp. 251–262.
[30] A. Levy, A. Rajaraman, J. Ullman, Answering queries using limited external processors, in: Proceedings of PODS, 1996, pp. 227–237.
[31] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, A database management system for semistructured data, SIGMOD Record 26 (3) (1997) 54–66.
[32] MetaMatrix Inc. Enterprise information integration. Available at <http://www.metamatrix.com/l3_whtpprs.jsp>.
[33] Y. Papakonstantinou, S. Abiteboul, H. Garcia-Molina, Object fusion in mediator systems, in: Proceedings of VLDB Conference, 1996.
[34] Y. Papakonstantinou, A. Gupta, L. Haas, Capabilities-based query rewriting in mediator systems, in: Proceedings of PDIS Conference, 1996.
[35] Y. Papakonstantinou, A. Gupta, L. Haas, Capabilities-based query rewriting in mediator systems, Distributed and Parallel Databases 6 (1998) 73–110.
[36] Y. Papakonstantinou, V. Vassalos, Query rewriting for semistructured data, in: ACM SIGMOD Conference, 1999, pp. 455–466.
[37] M. Petropoulos, V. Vassalos, Y. Papakonstantinou, XML query forms (XQForms): declarative specification of XML query interfaces, in: Proceedings of WWW10 Conference, 2001.
[38] D. Quass, A. Rajaraman, S. Sagiv, J. Ullman, J. Widom, Querying semistructured heterogeneous information, in: Proceedings of DOOD, 1995, pp. 319–344.
[39] M.A. Roth, H.F. Korth, A. Silberschatz, Extended algebra and calculus for nested relational databases, ACM Transactions on Database Systems 13 (1988) 389–417.
[40] A. Rajaraman, Y. Sagiv, J. Ullman, Answering queries using templates with binding patterns, in: Proceedings of PODS Conference, 1995, pp. 105–112.
[41] V.S. Subrahmanian et al., HERMES: a heterogeneous reasoning and mediator system. Available at <http://www.cs.umd.edu//projects/hermes/publications/abstracts/hermes.html>.
[42] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk, Querying XML views of relational data, in: VLDB Conference, 2001, pp. 261–270.
[43] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald, Efficiently publishing relational data as XML documents, in: VLDB Conference, 2000, pp. 65–76.
[44] G. Thomas et al., Heterogeneous distributed database systems for production use, ACM Computing Surveys 22 (1990) 237–266.
[45] V. Vassalos, Y. Papakonstantinou, Expressive capabilities description languages and query rewriting algorithms, Journal of Logic Programming 43 (1) (2000) 75–122.
[46] G. Wiederhold, Intelligent integration of information, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1993, pp. 434–437.
[47] L. Xyleme, A dynamic warehouse for XML data of the web, in: IDEAS, 2001, pp. 3–7.

Yannis Papakonstantinou serves on the Faculty of Computer Science and Engineering at the University of California, San Diego, since 1996. His research is in the intersection of database and Internet technologies. Yannis has published over 40 research articles in scientific conferences and journals, given tutorials at major conferences, and served on journal editorial boards and program committees (as member or chair) for numerous internatinal conferences and symposiums. In 1998, Yannis received the NSF CAREER award for his work on integrating heterogeneous data. In 2000 Yannis founded Enosys Markets, Inc., which provides software for XML-based querying and integration of distributed sources. Yannis holds a Diploma of Electrical Engineering from the National Technical University of Athens and M.S. and Ph.D. in Computer Science from Stanford University, 1997.



Vinayak Borkar works an XML and mediator query processing at Enosys Markets Inc. He holds a Bachelors of Engineering in Computer Science and Engineering from University of Mumbai, India and a Masters of Technology from the Indian Institute of Technology, Bombay.

Max Orgiyan received a B.S./M.S. in Computer Science and Engineering from UCSD. His M.S. thesis was in fault-tolerant TCP/IP. He has worked on molecular simulations at Accelrys, underwater data acquisition systems at Scripps Institution of Oceanography, parallel programming libraries at SDSC, fault-tolerant TCP/IP at AT&T research, and data integration at Enosys Markets.

Kostas Stathatos is a Director of Engineering at Enosys Markets, Inc. He has previously been a research scientist at Telcordia Technologies. Kostas holds a Diploma of Electrical Engineering from the National Technical University of Athens and M.S. and Ph.D. in Computer Science from Univeristy of Maryland at College Park, where he performed research in the intersection of networks and databases.

Lucian Suta currently works on the Enosys query processor. Previously, he worked on different aspects of the execution engine. Before joining Enosys, he worked on several projects at Qualcomm Inc. including one for the Software Research and Development group. He holds a B.S. and an M.S. in Computer Science and Engineering from UCSD, where he concentrated on distrubuted and fault-tolerant computing.

Vasilis Vassalos is an assistant professor in the Information Systems Department of the Stern School of Business at New York University. Vasilis has written numerous articles and participated in the committees of major databases conferences. In 2000 Vasilis founded Enosys Markets, Inc., which provides software for XML-based querying and integration of distributed sources. He joined Stern in September 1999, after obtaining a Diploma of Electrical Engineering from the National Technical University and M.S. and Ph.D. in Computer Science from Stanford University.

Pavel Velikhov is the chief architect of the Enosys query processor. He holds a B.S. and an M.S. in Computer Science and Engineering from UCSD. His thesis was on navigation-driven evaluation of XML queries.