SIGMOD 2003 Demo Proposal:
Relational On-Line Exchange with XML

Philip Bohannon Xin (Luna) Dong Sumit Ganguly Henry F. Korth Chengkai Li
P.P.S. Narayan Pradeep Shenoy

Lucent Technologies – Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974 USA
{bohannon,sganguly,hfk,ppsnarayan}@lucent.com
{luna,pshenoy}@cs.washington.edu cli@uiuc.edu

1 Introduction

As XML has gained widespread popularity, new applications are often XML-*based*, that is, they depend primarily on XML documents and the associated data model for data access and messaging. However, in most cases the XML-based application must *interoperate* with existing SQL-based applications. In the typical "shred-and-publish" approach to interoperation, incoming XML data is parsed (shredded) into relational tables and outgoing data is extracted by SQL engines and then formatted (published) as XML. For example, a database supporting an SQL-based hotel-reservation application may also be called on to support a web-site, or to exchange XML with a third party "hub" for the travel industry.

Maintaining the mapping between the relational data source and the associated XML documents is complex and error-prone. Fortunately, recently-developed middleware systems for XML publishing [3, 5] greatly ease this task by providing a declarative language in which a *view query* specifies the desired mapping. The view query is translated by the middleware into one or more SQL queries for execution on the underlying DBMS, and a *tagger* process constructs an XML document from the result.

Application-caching of database data is widespread, particularly in the web-facing applications that XML middleware systems are designed to support. Data is cached primarily for performance, and an experimental study by Labrinidis and Roussopoulos [8] of caching web data both in and out of the DBMS illustrates the problem. In almost every experiment, caching outside the DBMS offered *two orders of magnitude* better performance than caching within.

While caching may solve the performance problem, the application cache is undesirable for a number of reasons. First, multiple applications must each re-implement a portion of the functionality provided by the DBMS. Second, concurrency and data integrity among the caches and the relational DBMS must be managed by the application(s). This may lead to consistency problems when the underlying relational data is being accessed and updated by previously existing applications, while cached copies of this data are being used by e-business applications. Nevertheless, anecdotal evidence again indicates that this tradeoff is made frequently, leaving the DBMS in the "back room"—increasingly isolated from the bulk of web interactions.

ROLEX[1] is a research system for XML-relational interoperation [2]. In short, ROLEX seeks to provide the functionality of XML-relational middleware *at the speed of cached* XML *data*. To achieve this, ROLEX is integrated tightly with both the DBMS and the application, as shown in Figure 1(b). However, the integration with the application is through a standard interface supported by most XML parsers, the Document Object Model (DOM) [7]. Thus, in general, an application need not be modified to be used with ROLEX. To support our integration model and performance goals, ROLEX is built on the DataBlitz^{TM} Main-Memory Database System, allowing us to capitalize on extremely low-latency access to data while still providing advanced concurrency control and recovery features [1]. We expect ROLEX, when fully implemented, to be a compelling platform with the best of two worlds: the speed of cached XML files *and* the declarative data management tools and consistency guarantees of the DBMS.

In particular, contrast the ROLEX architecture, shown in Figure 1(b) with that of standard XML-relational middleware shown in Figure 1(a). As shown, the results of a ROLEX view query are provided to the application in the form of a *virtual* DOM tree rather than as a text document. Simply avoiding the cost of text generation and subsequent parsing is an important benefit of this approach. While our system is based on a particular main-memory database sys-

[1] ROLEX stands for Relational On-Line Exchange with XML.

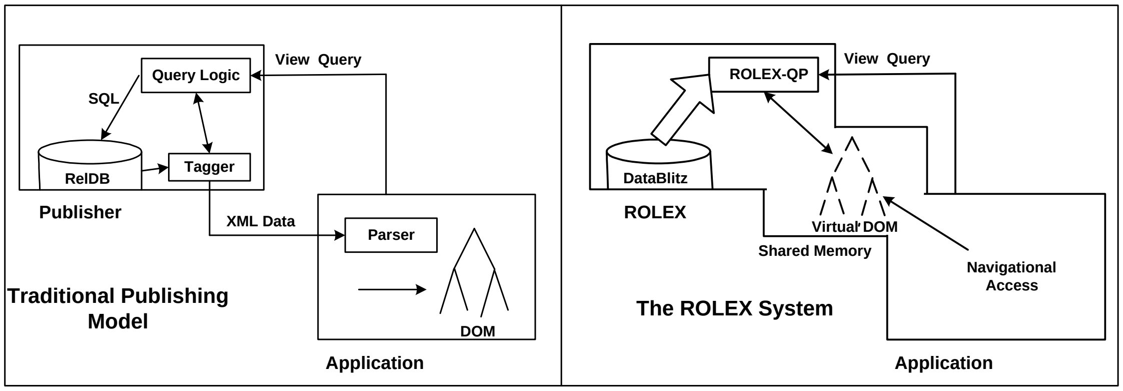


Figure 1: Publishing architectures (a) current approaches (b) ROLEX approach.

hotelchain(chainid, companyname, hqstate)
metroarea(metroid, metroname)
hotel(hotelid, hotelname, starrating, chain_id
    metro_id, state_id, city, pool, gym)
guestroom(r_id, rhotel_id, roomnumber, type, rackrate)
confroom(c_id, chotel_id, croomnumber, capacity, rackrate)
availability(a_id, a_r_id, startdate, enddate, price)

Figure 2: Hotel reservation schema.

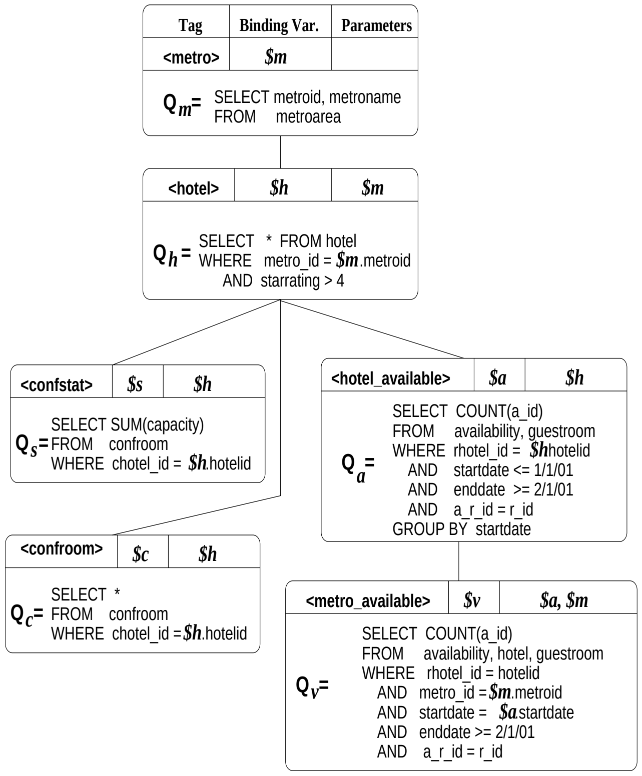
tem, we expect the model can be used with any closely-coupled architecture, including database-aware caches.

2 Overview

In this section, we introduce view-query specification in ROLEX using the example shown in Figure 3. This query format, referred to as a *schema-tree query*, is meant to capture a rich set of XML view queries, and is adapted from the intermediate query representation of [6]. This particular example defines an XML view on the tables of Figure 2 that supports conference planning by showing candidate hotels along with information about availability of rooms in the same metro area.

Each node in the schema-tree query includes a *tag*, a *tag query*, and a *binding variable*. Each tuple returned by the tag query becomes an element in the resulting XML document. Relational attributes can be mapped to XML attributes or subelements; however, these details are not shown. For example, the top-level node in Figure 3 has the tag `<metro>` and the tag query "$\mathcal{Q}_m$ = SELECT metroid, metroname FROM **metroarea**." This query defines a list of metropolitan areas that become sibling nodes in the resulting XML document, each tagged with the `<metro>` tag (a unique document root is implied). As shown in this example, the binding variable for a node may be used as a *parameter* when specifying tag queries of descendant nodes in the schema tree. For example, the variable $\$m$ associated with `<metro>` is used as a parameter in tag queries for `<hotel>` and `<metro_available>` to refer to the attribute $\$m$.metroid.

An application using ROLEX accesses data through a standard interface called the *Document Object Model* (DOM) [7]. The navigation functions implemented by DOM



```
Qm= SELECT metroid, metroname
    FROM   metroarea

Qh= SELECT  * FROM hotel
    WHERE  metro_id = $m.metroid
       AND  starrating > 4

Qs= SELECT SUM(capacity)
    FROM   confroom
    WHERE  chotel_id = $h.hotelid

Qa= SELECT  COUNT(a_id)
    FROM    availability, guestroom
    WHERE  rhotel_id =  $hhotelid
       AND   startdate <= 1/1/01
       AND   enddate  >= 2/1/01
       AND   a_r_id = r_id
    GROUP BY  startdate

Qc= SELECT  *
    FROM   confroom
    WHERE  chotel_id =$h.hotelid

Qv= SELECT  COUNT(a_id)
    FROM    availability, hotel, guestroom
    WHERE   rhotel_id = hotelid
      AND   metro_id =$m.metroid
      AND   startdate =   $a.startdate
      AND   enddate >= 2/1/01
      AND    a_r_id = r_id
```

Figure 3: An XML view query and its associated schema tree.

are as one would expect: parent-to-child, child-to-parent, and sibling-to-sibling. We also support navigating to the first child with a particular tag. A DOM interface to an XML view query supports all the DOM operations and behaves as if the user were navigating the XML document resulting from the query. For example, this might be accomplished by navigating the query results and building a DOM tree. A *virtual* DOM tree goes a step further by providing the same interface without creating the physical DOM tree. A *navigable query plan*, which we describe in the next section, is the mechanism used by ROLEX to support a virtual DOM tree.

A novelty of ROLEX is that it uses a navigational profile for a user or application when optimizing view-query plans. While navigational profiles can, in principle, be quite complex, we currently adopt a very simple model. If n is a node in the schema tree with parent p, the navigation profile stores $Pr\{n|p\}$, or the probability that some node in the DOM tree generated by n will be visited given that its parent, generated by p, has been visited. One simple way to gather this information is by collecting the corresponding statistic at each schema-tree node during view-query execution. While statistics gathering is not implemented in the demo, it is possible for the user to manually enter the profile probabilities with the query.

3 Demonstration Overview

The ROLEX prototype consists of three subsystems: the optimizer, the execution engine, and the virtual DOM layer. The execution engine and DOM interface operate on the tuple-layer interface of the DataBlitz^{TM} Main-Memory Database System. Note that, although the data is memory resident, many costs of a full-featured DBMS remain, including locking, latching, support for multiple data types, null handling, etc.

3.1 Interface

The demo has a web based interface allowing queries to be entered and edited. Once the user is satisfied, a query plan is generated as well as a graphical (postscript) representation of the plan. Query results are displayed using a standard browser.

3.2 Optimizer

A *navigable query plan* provides, for each node n in the schema tree of a view query, two entities: (1) a *subplan* for evaluating the tag query for n, and (2) a *navigation index*. The navigation index serves to materialize the output of the tag query and supports efficient lookup based on parameter values. The subplan may populate the navigation index lazily or eagerly as decided by the optimizer, and it may also materialize results to be used by other subplans.

The navigation index is distinguished from a normal (hash or tree) index by two additional features: (1) given a pointer to an entry in the index, the successor and predecessor matching the same key value can be reached efficiently, and (2) the index can record the fact that certain parameters produced empty results. These capabilities allow us to support DOM tree operations on the schema-tree view without explicitly generating the document; effectively implementing a virtual DOM interface.

One example of how we take advantage of the virtual DOM output is that we optimize our execution plan for user and application *navigation patterns* [2]. When query results are navigable, patterns of access to the document tree may be user- or application-specific. Using knowledge of these patterns, the ROLEX query optimizer selects execution plans that are expected to outperform, during navigation, plans optimized to generate the entire document. The

Table	Tuple Size (Bytes)	Cardinality
hotel	54	1000
metroarea	128	50
phone	24	3000
guestroom	20	40000
confroom	20	10000
availability	20	800000

Table 1: Table Cardinalities for Experimental Queries.

navigation opportunities for a user on an SQL query are typically limited to the use of bi-directional cursors. However, XML views of relational data can be large and complex, and even considering a subset of DOM functionality, user navigation on the result is potentially far more complex and more frequent than for relational results.

3.3 Engine

The execution engine has been built to serve as a general in-memory relational query-execution engine, as well as the execution engine for ROLEX. The engine handles a variety of join techniques, group-by and aggregates, and the materialization options discussed in [2]. In the implementation, the engine is decoupled from the optimizer, and an XML plan representation is used to communicate between the two.

3.4 Updates

We have implemented simple update functionality which demonstrates the potential of updating the relational data through the DOM interface. This implementation is part of ongoing work in defining updates through XML views [4]. The architecture consists of two modules, the information collection module and the view-update execution module. The information-collection module collects XML-view and relational schema information when the view-definition is parsed and sets up a "view-relationship graph" describing cardinality relationships between node pairs in the view. The view-relationship graph is then translated into update plans that are persisted in the system and later used at run time. The view-update execution module provides the interface for deletion, insertion, movement, and replacement on a given XML DOM node at run time. The execution module interacts with the relational database and the DOM interface to access the underlying data for the XML view. The two modules are connected through the persisted update plans that provide the necessary update translation and propagation information.

3.5 XSLT Stylesheets

We also demonstrate XSLT stylesheets running on the DOM result of view queries. However, this demonstration does not currently allow all features of XPATH and XSLT, since the processor internally copies the tree in order to add order information it uses to handle certain features, especially involving sibling axes in XPATH. We expect to also demonstrate a preliminary version of an XSLT *view composition* algorithm, which is the subject of on-going research [9].

```
CREATE VIEW view1 AS
<hotel>
  ( $h =
    SELECT hotelid, hotelname, starrating, state_id
    FROM hotel
  )
    <avail>
    ( $a =
      SELECT rhotel_id, startdate, rhotel_id, roomnumber
      FROM availability, guestroom
      WHERE type > 5 AND rhotel_id = $h.hotelid
      AND startdate > 12/15/02 AND r_id = a_r_id
    )
    </avail>
</hotel>;
```

Figure 4: XML view query for experiments.

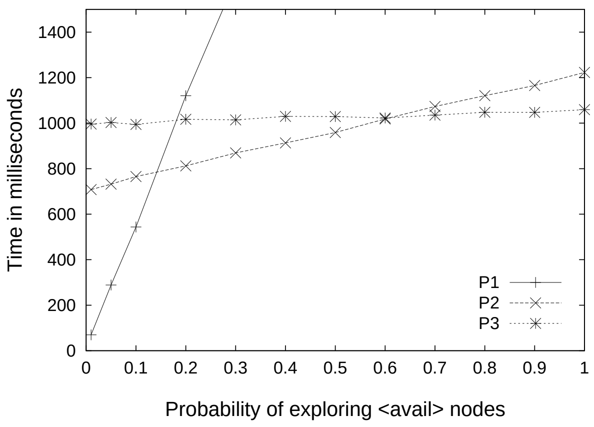


Figure 5: Performance of plans $P1$ through $P3$ as a function of navigation probability for the view query in Figure 4.

4 Performance

4.1 Impact of Navigation Profiles

We observe that, within the parameters of our system, generating an execution plan for all probabilities set to 1.0 most closely approximates a plan optimized for document export. Similarly, a plan optimized for low (but non-zero) probabilities at nodes lower in the tree most closely approximates the heuristic of attaching all child plans to their parents by outer joins. The general approach of our experiments is to compare these two "extreme" plans to the plan chosen by the ROLEX optimizer, across a range of probabilities, with our contention being that neither "extreme" plan performs well across the range.

In our first experiment, we consider the view query shown in Figure 4. For this view query, the ROLEX optimizer finds three optimal plans ($P1$, $P2$, and $P3$) as the navigation probability is varied from 0.01 to 1.0 and estimates that they are optimal in the ranges $[0.0, 0.15)$, $[0.15, 0.25)$, and $(0.25, 1.0]$ respectively. Due to lack of space we do not show the plans in this paper. We see that the high probability plan is the decorrelated plan, where the query of the `<avail>` node is re-written to do a join with the query of the `<hotel>` node. This join is evaluated only once; the first time any `<avail>` node is visited. Hence the high cost of plan $P3$ at low probabilities.

The performance of each of these plans as a function of navigation probability is shown in Figure 5. The figure shows that three plans $P1$, $P2$, and $P3$ are actually optimal in the ranges $[0, 0.15)$, $[0.15, 0.6)$, and $[0.6, 1]$ respectively. Execution time of the plan $P1$, which is optimal at low probabilities, grows linearly with increasing probability, and executes in 5.1 seconds for a probability of 1.0. This is not shown in Figure 5. The error in the probability cutoff, attributed to cost-model variances, leads to a 5% to 15% sub-optimal execution.

However, Figure 5 emphasizes that there exist distinct optimal plans for different regions of the probability space. The experimental results confirm that executing a plan optimized for very low probability values such as $P1$ is highly sub-optimal at high probability values and vice-versa. Since the plan for probability of 1.0 corresponds, in our model, to the scenario of full document export, we conclude that such plans are sub-optimal at the lower end of the navigation probability spectrum.

5 Conclusion and Future Work

Increasingly, relational databases support simultaneous "OLTP" access via SQL and XML interfaces. ROLEX provides a novel approach to resolving this duality by offering the ability to access live, non-materialized XML views of relational data, directly and efficiently, through a navigable virtual DOM interface. As a result, the system avoids the overhead of tagging and parsing that limits the performance of existing middleware systems.

In this paper, we propose to demonstrate the working ROLEX prototype, which includes a query language, optimizer and runtime with web-based interface, as well as limited support for execution of XSLT transforms and updates on the DOM tree which get mapped to the relational model. We note that the ROLEX architecture provides a promising platform for integrating database functionality (query language, integrity constraints) with popular application platforms while providing high performance.

References

[1] J. Baulier et al. DataBlitz storage manager: Main memory database performance for critical applications. In *Proc. of the ACM SIGMOD Int'l. Conf. on the Management of Data*, 1999. Industrial track paper.

[2] P. Bohannon, S. Ganguly, H. Korth, P. Narayan, and P. Shenoy. Optimizing view queries in rolex to support navigable result trees. In *Proc. of VLDB 2002*, 2002.

[3] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *Proc. of the Third Int'l. Workshop on the Web and Databases*, 2000.

[4] X. L. Dong, P. Bohannon, H. Korth, and P. Narayan. Updating XML views of relational data. In *(submitted for publication)*, 2002.

[5] M. Fernández, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 2001.

[6] M. Fernández, D. Suciu, and W. Tan. SilkRoute: Trading between relations and XML. In *Proc. of the WWW9*, 2000.

[7] A. L. Hors, P. L. Hegaret, G. Nicol, J. Robie, M. Champion, and S. Byrne (Eds). "Document Object Model (DOM) Level 2 Core Specification Version 1.0". W3C Recommendation, Nov. 2000. http://www.w3.org/TR/DOM-Level-2-Core/.

[8] A. Labrinidis and N. Roussopoulos. WebView materialization. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 2000.

[9] C. Li, P. Bohannon, H. Korth, and P. Narayan. Composing XSLT stylesheets with XML publishing views. In *(submitted for publication)*, 2002.