Optimizing View Queries in ROLEX to Support Navigable Result Trees

P. Bohannon S. Ganguly H. F. Korth P.P.S. Narayan P. Shenoy

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

Abstract

An increasing number of applications use XML data published from relational databases. For speed and convenience, such applications routinely cache this XML data locally and access it through standard navigational interfaces such as DOM, sacrificing the consistency and integrity guarantees provided by a DBMS for speed. The ROLEX system is being built to extend the capabilities of relational database systems to deliver fast, consistent and navigable XML views of relational data to an application via a *virtual* DOM interface. This interface translates navigation operations on a DOM tree into execution-plan actions, allowing a spectrum of possibilities for lazy materialization. The ROLEX query optimizer uses a characterization of the navigation behavior of an application, and optimizes view queries to minimize the expected cost of that navigation. This paper presents the architecture of ROLEX, including its model of query execution and the query optimizer. We demonstrate with a performance study the advantages of the ROLEX approach and the importance of optimizing query execution for navigation.

1 Introduction

XML has gained widespread popularity as a standard for information representation and exchange. Infrastructure software for business hubs, supply-chain integration, and catalog management all use XML encodings. Standards bodies for business data exchange, such as RosettaNet [21] and Oasis-Open [18], are extremely active. The result is a tremendous focus on incorporating support for XML in application-development and data-management tools.

In some cases, an XML-based application may be developed from scratch, and perhaps require a storage facility for XML documents [3, 25]. However, in most cases the XML-based application must *interoperate* with existing SQL-centric 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 [5, 8] 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. Furthermore, commercial relational and object-relational databases are becoming "XML-enabled" [20] by integrating certain middleware functionality into the DBMS. This may entail, for example, supporting a modified SQL syntax that outputs XML or allowing XPath queries against an XML view of the database.

1.1 Caching and the "Back-Room" DBMS

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 [16] 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.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

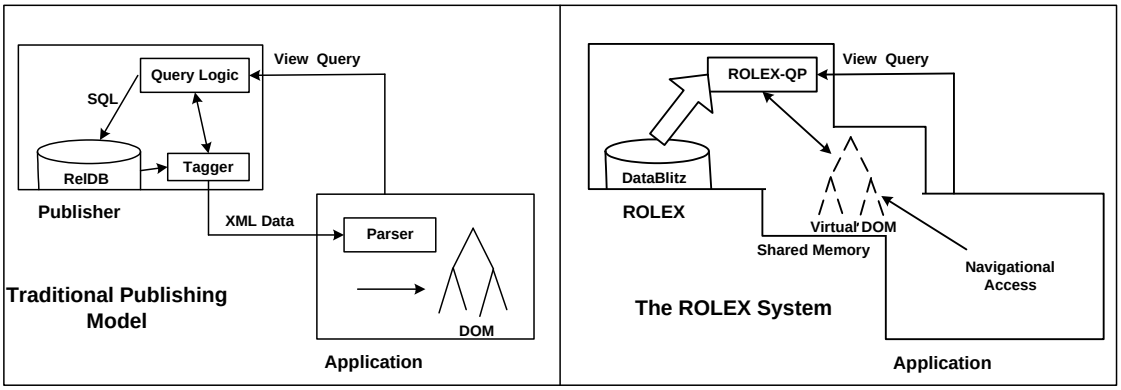

Figure 1: *Publishing architectures (a) current approaches (b)* ROLEX *approach*

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.

1.2 ROLEX System Architecture

ROLEX[1] is a research system for XML-relational interoperation [4]. 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) [14]. 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 [2]. 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. In fact, for simple queries, experimental results in this paper demonstrate that ROLEX can produce a fully traversed query answer in less time than it would take merely to parse that answer from text form. While our system is based on a particular main-memory database system, we expect the model can be used with any closely-coupled architecture, including database-aware caches [27].

In this paper, we focus on the ROLEX query execution and optimization framework and on one critical way in which we can capitalize on the virtual result tree: optimizing our execution plan for user and application *navigation patterns*. The 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.

1.3 Optimizing for Navigation

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.

To illustrate, consider a travel hub that supports two applications, *room browsing* and *convention planning*. The browsing application lets users examine hotels in a specific metropolitan area for an accommodation meeting their requirements. In the convention-planning application, the user tries to find a collection of hotels within an area that satisfy multiple aggregate criteria (such as total room availability or conference room capacity).

Consider first the scenario where both these applications have been developed using an XML view that conforms to an industry-standard schema. In this case, the room browsing application seldom, if ever, requests the information in the tree about a hotel's total conference-room capacity, while in the convention-planning application, this information may be accessed frequently. Clearly therefore, in the room-browsing application, it would be desirable for the

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

optimizer to use a lazy evaluation strategy for retrieving these data. ROLEX uses a *navigational profile* to represent the probability of the application navigating along edges in the DOM tree. In the example application, we would expect the probability of navigating to a conference room to be almost zero. An optimizer cognizant of navigational profiles is thus able to choose the lazy evaluation strategy, as desired.

Alternatively, the XML view may be defined by the application itself. For example, an application query may be "composed" with the view query to produce a new view query [9]. Such views conform closely to the actual needs of the application. Therefore, in the browsing application, computation of the total conference-room capacity would be eliminated by a well-written application view. One might expect this to obviate the need for a navigational profile; however, this expectation turns out to be incorrect. For instance, in the room-browsing scenario, a typical user is likely to navigate to a few hotels from the query result that satisfy certain user criteria. Therefore, use of a navigational profile can reduce resource utilization, since not all the query results that might be of interest are actually accessed. If a view query uses two relations **metroarea** and **hotel**, for example, a simple navigational profile may be constructed by tracking the fraction of hotels accessed among those in the given metropolitan area across multiple previous invocations of the application. If this fraction is small, the optimizer might choose to implement the query using a nested-loop join between these relations. On the other hand, if this fraction is large, the optimizer may materialize the join between these relations into a hash index that is used to support navigation. We argue that complex view queries contain many such tradeoffs; balancing them is part of the optimization space explored by ROLEX.

In summary, navigation profiles offer significant opportunities for optimization of query execution, regardless of whether the XML view is defined by a standard or by the application. In the absence of support for navigation, an application must either request all data that it *might* need, or it must submit multiple, distinct queries to the system. Both cases result in high processing overhead. By taking the navigational profile of the application into account, the ROLEX approach offers the promise of reduced resource utilization and lower response time.

1.4 Contributions

The contributions of this paper are three-fold. First, we describe the novel system model of ROLEX and its query modeling and execution framework. Second, we describe the modifications made to the design space and rule set of a VOLCANO-style [13] rule-based optimizer to implement optimization of ROLEX view queries. These modifications include a new operator representing navigation, a model of decorrelation for complex tree queries, and a new cost model that takes into account the application's navigation profile. A critical result is that the integration is straightforward and the impact on the optimizer is limited. Our third contribution pertains to optimization for the expected cost of navigating the result XML tree. In fact, we show that optimizing for expected navigation, even with a very simple navigational profile, can improve performance substantially when the application or user's navigation fits the profile, and in most cases this plan is robust if the navigation is somewhat different than expected.

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*

1.5 Outline of the Paper

The outline of the remainder of the paper is as follows. In Section 2, we introduce our running example and describe queries and navigation profiles. In Section 3 we introduce the virtual DOM tree and navigable query plans. Section 4 describes the space of decorrelation options we consider. In Section 5 we describe how a VOLCANO-style optimizer is extended to optimize ROLEX view-queries. The cost model used by the ROLEX optimizer to account for navigational profiles is presented in Section 6. Experimental results are presented in Section 7. Related work is discussed in Section 8, followed by our conclusion and a discussion of future work in Section 9.

2 Model for Queries and Navigation

This section introduces the view queries accepted by ROLEX and presents our model of navigation profiles.

2.1 Schema-Tree Queries

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 [9]. 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. The binding variable associated with a node is used in descendant nodes as a tuple variable ranging over the results of the tag query. 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**." (We subscript the tag query with the binding variable, in this case $\mathcal{Q}_m$.) This query defines a list of metropolitan areas, which

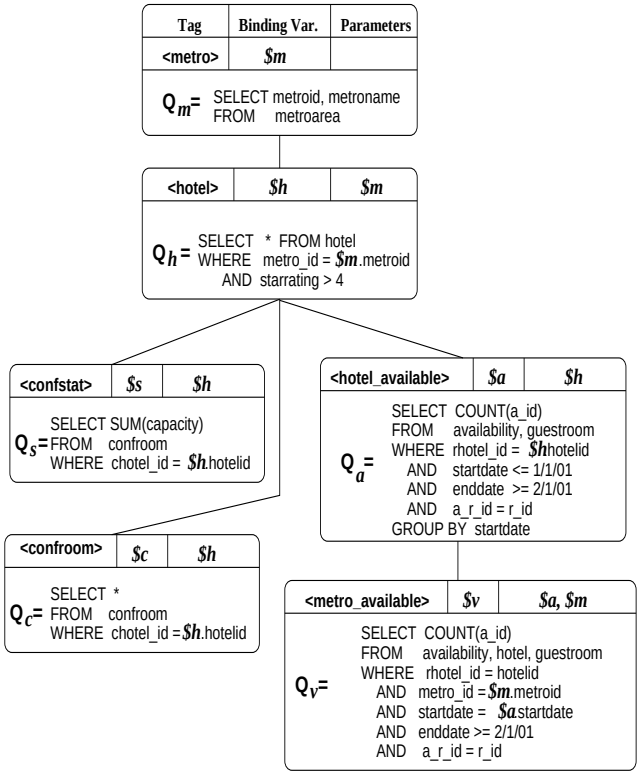

Figure 3: *An* XML *view query and its associated schema tree*

become sibling nodes in the resulting XML document, each tagged with the `<metro>` tag (a unique document root is implied). For simplicity of presentation, tags and binding variables are unique and mutually exclusive in this paper (in general, tags could be repeated).

As mentioned above, 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. Tag queries may be parameterized by zero or more parameters, associated with the same or different binding variables. We refer to the query which defines binding variable x as $\mathcal{Q}_x(s_1, s_2, \ldots, s_k)$, where $s_1, s_2, \ldots, s_k$ are the binding variables mentioned in the body of $\mathcal{Q}_x$.

The remainder of the view in Figure 3 defines the following. The tag query, $\mathcal{Q}_h(m)$ for `<hotel>` is parameterized by the tuple variable m running over metropolitan areas and gives a list of hotels in that metropolitan area. The tag query, $\mathcal{Q}_a(h)$, for `<hotel_available>` counts available rooms at the given hotel in a certain fixed time period, whereas the tag query, $\mathcal{Q}_v(m, h)$ for `<metro_available>` counts the total available rooms in the entire metropolitan area for that same time period. In separate branches of the schema-tree, summary and detail information about conference rooms is given by the nodes with tags `<confstat>` and `<confroom>` respectively.

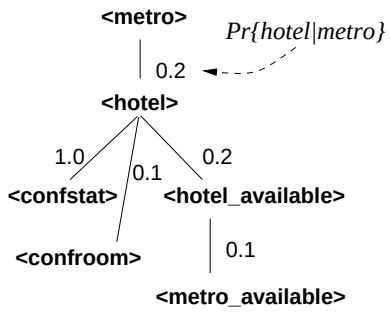

Figure 4: *A sample navigational profile*

2.2 Navigational Profile

As mentioned in the introduction, ROLEX 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 estimate this probability is by collecting the corresponding statistic at each schema-tree node during application navigation. Exploration of more sophisticated navigation profiles that might, for example, consider data values or navigation order, are left as future work. An example navigational profile for the query of Figure 3 is shown in Figure 4.

3 Virtual DOM

An application using ROLEX accesses data through a standard interface called the *Document Object Model* (DOM) [14]. The navigation functions implemented by DOM 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 next, is the mechanism used by ROLEX to support a virtual DOM tree.

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*, NI_n. 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

procedure nav-to-child(c)
begin
Assume tag query for c is $\mathcal{Q}(s_1, s_2, \ldots, s_k)$
1. Extract parameter values for $s_1, s_2, \ldots, s_k$ from current tuples of c's ancestor nodes
2. Search NI_c for $s_1, s_2, \ldots, s_k$ parameter values
3. **if not** found **then**
4. Use $s_1, s_2, \ldots, s_k$ to initialize the subplan for c
5. Add results of plan to NI_c
6. **endif**
7. Use NI_c to support sibling navigation

end

Figure 5: *Navigation from parent to child node c*

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. For example, the actions taken on navigation from a parent to a child node are given as pseudo-code in Figure 5. Although not shown in the pseudo-code, if the subplan is pipelined, results from the subplan can be materialized into NI_c lazily during user navigation. Though it might be beneficial to continue executing the query plan "ahead" of the user while waiting for the next DOM-traversal step, we do not consider such "speculative" execution in this paper.

4 The Decorrelation Plan Space

Decorrelation has been studied in the context of generating equivalent executions for correlated SQL queries in [6, 11, 15, 24]. In all previous work of which we are aware, plans are decorrelated when possible on the heuristic assumption that the decorrelated execution can be optimized better. On the contrary in ROLEX, when the navigation profile indicates that a node will seldom be visited, *correlated* execution may be preferred. Various subsets of tag queries may be decorrelated, and the navigation profiles for which each is optimal obviously depends on the queries, database structures, and statistics. In this section, we discuss how we use standard decorrelation transformations to generate a space of equivalent plans for schema-tree queries. While these transformations are implemented at the plan level, they are more easily described at the SQL level, the approach taken in this section. The embedding of this plan-space in a VOLCANO-style optimizer is discussed in the next section.

4.1 Single-Parameter Decorrelation

In this section, we describe the basic transformation used to decorrelate a single parameter from a node query. Consider a tag query, $\mathcal{Q}_x(y)$, which defines the variable x and has a single parameter y and takes the following form:

procedure nav-to-child-dec(c)
begin
Assume tag query for c is $\mathcal{Q}(s_1, s_2, \ldots, s_k)$
Assume decorrelated plan for c is $\mathcal{Q}^{s_1 s_2 \ldots s_j}(S)$, where $S = (s_{j+1}, \ldots, s_{k'})$ may be empty
1. Extract parameter values for $s_1, s_2, \ldots, s_{k'}$ from current tuples of c's ancestor nodes
2. Search NI_c for $s_1, s_2, \ldots, s_k$ parameter values
3. **if not** found **then**
4. Use $s_{j+1}, \ldots, s_{k'}$ to initialize the subplan for c
5. Add results of plan to NI_c
6. **endif**
7. Use NI_c to support sibling navigation

end

Figure 6: *Navigation to child c, with decorrelated plan*

SELECT *select-list* FROM *from-list*
WHERE *preds-list* GROUP BY *group-by-list*
HAVING *having-list* ORDER BY *sort-list*

where y is defined by an ancestor tag query, $\mathcal{Q}_y$. We denote by $\mathcal{Q}_x^y$ the query corresponding to $\mathcal{Q}_x(y)$ that has been decorrelated with respect to $\mathcal{Q}_y$. $\mathcal{Q}_x^y$ is defined by the following SQL query:

SELECT *select-list* FROM *from-list*, $\mathcal{Q}_y$ as **temp**
WHERE *new-preds-list* GROUP BY *group-by-list*
HAVING *new-having-list* ORDER BY *sort-list*

where the list of relations in the FROM clause includes the definition of the query $\mathcal{Q}_y$ renamed as a new relation **temp**. We obtain *new-preds-list* and *new-having-list* from their previous counterparts by replacing each occurrence of the parameter y by **temp**. Note that if the query $\mathcal{Q}_y$ had a parameter z, i.e., $\mathcal{Q}_y(z)$, then the above decorrelated query would also be parametrized by z, that is $\mathcal{Q}_x^y(z)$. The decorrelated query $\mathcal{Q}_x^y(z)$ can then be decorrelated further to eliminate the parameter z and thereby obtain $\mathcal{Q}_x^{yz}$. Each of these decorrelated queries may be a candidate for further query transformations like those described in [15]. Since these transformations are standard in the literature on query processing, we do not present details here.

4.2 Multi-Parameter Decorrelation

The extension of the technique above to tag queries with multiple parameters is straightforward. Consider a tag query $\mathcal{Q}_x(s_1, s_2, \ldots, s_k)$, parameterized by k binding variables, $s_1, s_2, \ldots, s_k$. The idea is to treat $\mathcal{Q}_x$ as a query with a single parameter, corresponding to the binding variable for the schema-tree node that is *lowest*, or closest to $\mathcal{Q}_x$, in the tree. We decorrelate this query to remove one variable, say s_i, and possibly add several more; however, these variables are defined higher in the tree than s_i. We continue this process until a fully decorrelated query is obtained. The *decorrelation space* for $\mathcal{Q}_x(s_1, s_2, \ldots, s_k)$ is the set of queries obtained during this process. In the example shown in Figure 3, the `<metro_availability>` tag query, $\mathcal{Q}_v(a, m)$, has two parameters, a.startdate and m.metroid. Based on this discussion, the decorrelation

space for $\mathcal{Q}_v(a,m)$ is $\{\ \mathcal{Q}_v(a,m),\ \mathcal{Q}_v^a(h,m),\ \mathcal{Q}_v^{ah}(m),\ \mathcal{Q}_v^{ahm}()\ \}$.

When the final plan chosen to implement a node, say c, is not completely decorrelated, then the parameters used to initialize the plan differ from those used as a key for c's navigation index, NI_c. A modified version of the algorithm given in Figure 5 that handles this case is shown in Figure 6.

4.3 The Ups and Downs of Decorrelation

Typically the result of a decorrelation step is the elimination of the subquery. As noted above, ROLEX deviates from this by retaining the correlated subplan, since it may be better to use the correlated subplan for nodes with lower navigation probabilities. However, there is a more striking difference in the ROLEX approach: as seen above, the result of a decorrelation step in ROLEX is to replace the *child* plan, while leaving the parent plan intact. When viewed in the context of a schema-tree query, this transformation is a "down" decorrelation, since parts of queries always move down the tree, as opposed to the "up" decorrelation which is standard.

An obvious drawback of this approach is duplication; for example, a complicated expression near the root of the tree may be duplicated in a number of leaf nodes, due to "down" decorrelation. However, this problem is alleviated by three factors. First, a significant simplicity arises from the fact that we do not need transformations for outer-join operations. Second, the resulting size of the decorrelation space for "down" decorrelation is no larger than the number of nodes in the tree times the height of the tree, while the number of possible "up" plans is exponential in the number of nodes in the tree (since any subset of the children of a node can be decorrelated with it). The third advantage of performing "down" decorrelation is an artifact of the optimizer [22] we are extending. Since that optimizer was designed for multi-query optimization, it is particularly good at factoring the common expressions generated by "down" decorrelation.

5 Optimizing Navigable Query Plans

The ROLEX query optimizer is built on top of a rule-based query optimizer designed for multi-query optimization [22] that implements many of the features of VOLCANO [13]. ROLEX could alternatively employ a bottom-up approach as in [23] but we do not consider that possibility here. In this section, we review the VOLCANO data structures, describe how the ROLEX plan space is implemented in this framework and discuss our materialization strategy for subplans.

5.1 The VOLCANO AND-OR DAG

In the VOLCANO AND-OR DAG, each OR node (also called an equivalence class) represents alternative ways to evaluate a subexpression, say E, of the original SQL query. Each OR node has one or more AND node children, where each child represents the top relational-algebra operator of

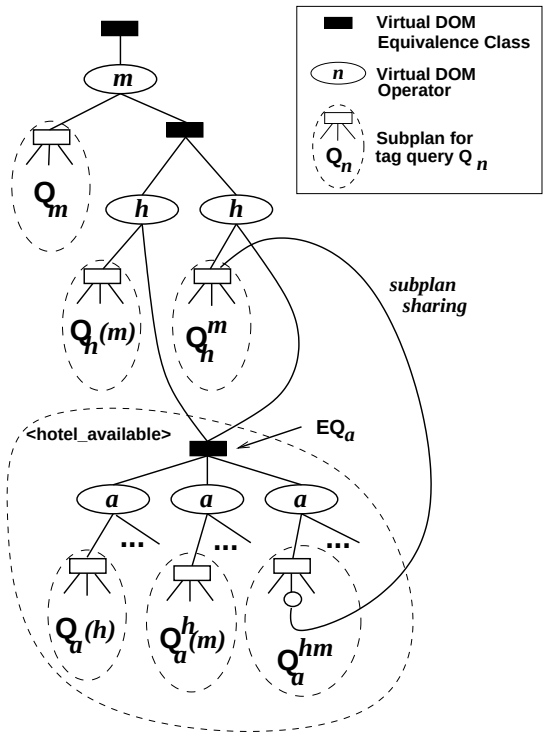

Figure 7: *Virtual* DOM AND-OR *nodes*

a subplan implementing E. For example, if an OR node represents $A \bowtie B \bowtie C$, it may have two join-operator children, one, say j_1 representing the join between A and $(B \bowtie C)$ and the other, say j_2 representing the join between $(A \bowtie B)$ and C. The children of AND nodes are, in turn, the OR nodes corresponding to the subqueries of its operands. For example, the j_1 operator in the above example has an OR-node child representing different ways of computing $(B \bowtie C)$.

Common subexpressions appear once in the DAG structure; for example, all operators with an input equivalent to $A \bowtie B \bowtie C$ point to a single equivalence class for this expression. At an abstract level, the optimization proceeds as follows: once all the logical transformation rules are applied to expand the DAG (e.g. join reordering), a branch-prune pass is made to find, in a bottom-up manner, the best (cheapest) physical execution strategy for each AND node, and by extension, the OR node with which it is associated. For more details, see [22].

5.2 Logical Operators for Navigation

To represent our execution space, we add two new types of nodes, an OR node called the *virtual*-DOM *equivalence class*, and an AND node called the *virtual*-DOM *operator*. We explain the function of these new nodes with the example DAG shown in Figure 7. This figure shows a portion of the AND-OR DAG representing the decorrelation space of the schema-tree query from Figure 3. We associate a virtual-DOM equivalence class, say EQ_n, with each schema-tree node n, to represent decorrelation alter-

natives. For example, the node labeled EQ_a is the equivalence class for the decorrelation space of $\mathcal{Q}_a(h)$, the tag query for the <hotel_available> node. Each child, o_a (labeled "a" in Figure 7) of EQ_a is a virtual-DOM operator implementing one decorrelation strategy for that schema-tree node. During plan execution, these operators perform the nav-to-child-dec procedure from Figure 6. The first (leftmost) input of a virtual-DOM operator o_n, is the OR-node of the relational subplan for node n. The remaining inputs are virtual-DOM equivalence classes that correspond to the children of the schema-tree node n in the schema-tree query. These edges represent the navigation options of the application query.

The fact that the VOLCANO DAG structure consolidates common subexpressions ensures efficient optimization of the ROLEX decorrelation plan space. In particular, when a query $\mathcal{Q}_x(y)$ is decorrelated with $\mathcal{Q}_y(z)$ to produce $\mathcal{Q}_x^y(z)$, the entire subplan for $\mathcal{Q}_y(z)$ is included in the plan for $\mathcal{Q}_x^y(z)$. This is shown in Figure 7 by the edge labeled "*subplan sharing*." Of course, as the logical DAG is expanded by other transformation rules, the plan for $\mathcal{Q}_x^y(z)$ explores, for example, join orders not usable in the $\mathcal{Q}_y(z)$ plan. However, those subplans in common are shared and optimized once.

5.3 Opportunistic Materialization

Each equivalence node and operator (not just virtual-DOM nodes) is labeled by the set of parameters that appear in the DAG rooted at that node. We use this information to materialize subplans *opportunistically*, so that a given physical operator is executed only as many times as required by its bindings. In particular, we mark as *materialized* any operator whose parent parameter set is a proper superset of its parameter set. This subplan is re-initialized whenever a binding variable in its parameter list changes. Note that this follows the approach taken in [19] in which subexpressions with exactly zero bindings are executed only once. We generalize this to materialize at the appropriate level for the bindings present. For example, selection on **hotel** for a particular m.metroid appearing in the subplan for $\mathcal{Q}_a^h(m)$ in Figure 7 is executed exactly once for each metro area, rather than once for each <hotel_available> node.

6 Cost Estimation

Since ROLEX explicitly considers a space of both correlation and decorrelation options, as opposed to attempting to maximize the amount of decorrelation, it is important to cost complex correlated plans with reasonable accuracy, including the "opportunistic materialization" discussed in Section 5.3.

This section shows how we estimate the key components of our cost model: (1) the number of visits to a node and (2) the number of tuples produced by such a visit. The number of visits to a node is somewhat complex to compute since it depends on the number of unique values generated for the node's parameters during the execution. In ROLEX, the cost for each node represents the estimated contribution of that node to the *total* cost of the user navigation. A node that contains a parameter may be executed multiple times, and the cost of that node includes the expected number of executions, the cost to materialize these executions if needed, and the cost to use the materialized versions the appropriate number of times.

The cost model presented in this section is implemented by extending the cost model of a traditional SQL optimizer and additionally uses information about functional dependency and foreign-key constraints over the database to make estimations more accurate [12]. Extending the model to handle navigation profiles is accomplished relatively easily, supporting the claim that traditional query-optimization techniques can be modified easily to optimize for navigational profiles.

6.1 Estimating Schema-Tree Statistics

Two basic components of our cost model for schema-tree nodes and their associated tag queries are *visits* and *unit-size*. The expected unit size of a node n, denoted $EUSize(n)$ is the expected number of tuples produced by a single "unit" call to n's tag query with representative parameter values. This number is estimated using traditional size estimation techniques for SQL queries. For example, consider the tag query $\mathcal{Q}_m$, associated with metro in Figure 3. Using the cardinalities of Table 1, $EUSize(\texttt{metro}) = EUSize(\mathcal{Q}_m) = 50$. And similarly, if we assume a) uniform distribution of hotels in the metro areas, and b) the star ratings for hotels (**hotel**.starrating) range uniformly from 1 to 8 (hence our query has 0.5 selectivity), $EUSize(\texttt{hotel}) = EUSize(\mathcal{Q}_h(m)) = (1000/50) \times 0.5 = 10$. (While the discussion and examples assume uniform distribution for simplicity, the cost model in this section generalizes in a straightforward manner to use histogram information if available.) We define $EUSize(root) = 1$ where $root$ is the implied root of the schema-tree query. We overload $EUSize$ to the fields that appear in the output of a query as follows: if fld is a field in the output of the query at node n, then $EUSize(\mathsf{fld}, n)$ is the expected number of distinct values of fld resulting from a single call to n.

$EVis(n)$ denotes the expected number of "visits" to node n during a traversal that obeys the navigational profile. If all probabilities in the navigational profile are 1, $EVis(n)$ corresponds to the number of DOM nodes generated by n in a full result document. If $p(n)$ is the parent of n, we calculate $EVis(n)$ recursively as:

$$EVis(root) = 1$$
$$EVis(n) = Pr\{n \mid p(n)\} \times EVis(p(n)) \times EUSize(p(n))$$

That is, $EVis(n)$ is the expected number of $p(n)$ tuples (computed as the product of the expected number of visits to $p(n)$ and the expected number of tuples generated per visit) times the probability that a $p(n)$ visit leads to a visit to n. In our example, $EVis(\texttt{hotel}) = 50$ (once for each metro) if navigation probabilities are all 1, but

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*

given the navigation profile shown in Figure 4, in which $Pr\{\texttt{hotel} \mid \texttt{metro}\} = 0.2$, $EVis(\texttt{hotel})$ would be only 10. Note that $EVis(n)$ estimates the number of lookups made to the navigation index for n.

6.2 Binding Variables and Parameters

Since the navigation index ensures that a subplan is not called with duplicate parameters, the number of unique parameter bindings seen at a node n is the *expected unique calls* of the subplan for n, which we denote as $EUniqCalls(n)$. Consider the `metro_available` node in Figure 3. The query for this node, $\mathcal{Q}_v(a, m)$, is parameterized by m.metroid and a.startdate. We need to estimate the number of unique (m.metroid, a.startdate) pairs that we expect will be *seen* at the `metro_available` node in order to estimate $EUniqCalls(\texttt{metro\_available})$.

In some cases, like the `hotel` node of Figure 3, $EUniqCalls(n) = EVis(n)$. Since $Pr\{\texttt{hotel} \mid \texttt{metro}\} = 0.2$ in the navigational profile shown in Figure 4, we can estimate that $EUniqCalls(\texttt{hotel}) = 10$, and we say that 10 unique `metro`.metroid values are "visible" at `hotel`. However, in general, $EUniqCalls(n)$ and $EVis(n)$ numbers can differ; for example, $EUniqCalls(\texttt{metro\_available})$ has far more visits than unique calls since it depends on the metro area and the startdate and thus its contents may be the same for several hotels in that metro area.

In order to compute $EUniqCalls$, we need two auxiliary statistics DV and $EUniqBind$. For a virtual DOM node a, $DV(\mathsf{fld}, a)$ is an upper bound of the expected number of distinct values of fld produced by the query at node a over all visits to that node. Since we assume **hotel**.starrating range uniformly from 1 to 8 in our running example, $DV(\mathsf{starrating}, \texttt{hotel})$ is 4 (0.5 selectivity). And since hotelid is the key for **hotel**, we get $DV(\mathsf{hotelid}, \texttt{hotel}) = 1000 \times 0.5 = 500$.

$EUniqBind(x.\mathsf{fld}, n)$ is the estimated number of distinct bindings (or values) that the parameter x.fld takes at a node n, given that x is defined at node a and n is either a or a descendant of a. The calculation for $EUniqBind$ is defined recursively as follows:

$$EUniqBind(x.\mathsf{fld}, n) = \begin{cases} \min(EUniqCalls(a) \times EUSize(\mathsf{fld}, a), DV(\mathsf{fld}, a)) & \text{(n=a)} \\ \min(EUniqBind(x.\mathsf{fld}, p(n)), EVis(n)) & \text{otherwise} \end{cases}$$

where $p(n)$ is the parent node of n. In the $n = a$ clause, we estimate the value of $EUniqBind(x.\mathsf{fld}, a)$, at node a where x.fld is generated, as the lower of two upper bounds. The first bound is obtained by assuming that distinct parameter bindings result in disjoint output sets; whereas the second one is the output-domain size, $DV(\mathsf{fld}, a)$. The "otherwise" clause asserts that for v values of some field to appear at a node or any of its descendants, there must be at least v visits to that node. This last point can be seen by considering the evaluation of $EUniqBind(m.\mathsf{metroid}, \texttt{hotel})$, which is limited to 10 by the number of visits to `hotel`.

We now estimate $EUniqCalls(n)$ recursively, by utilizing the statistic $EUniqBind(x.\mathsf{fld}, n)$. Suppose the query at node n is $\mathcal{Q}_x(y_1, y_2, \ldots, y_k)$. We first classify the parameters $y_1, y_2, \ldots, y_k$ into two sets, *independent parameters* and *dependent parameters*. A parameter y_j is said to be dependent if it is functionally dependent on some subset of the remaining parameters. For example, if a query in Figure 3 used both h.hotelid and m.metroid, we use schema information to infer that the parameter h.hotelid determines m.metroid, and thus only h.hotelid should be considered for $EUniqCalls$. Assume, without loss of generality, that the first k' parameters, $y_1, y_2, \ldots, y_{k'}, k' \leq k$, comprise the set of independent parameters. In this case, the calculation of $EUniqCalls(n)$ is

$$EUniqCalls(n) = min\left(\prod_{i=1}^{k'} EUniqBind(y_i, n), EVis(n)\right)$$

Clearly, each of the operands to the min operator above are upper bounds to $EUniqCalls(n)$.

6.3 Operator Costs and Optimization

In this section, we discuss how we assign costs to the AND nodes in the VOLCANO AND-OR DAG and compute the optimal plan (by assigning costs to the OR nodes) during a bottom-up pass of the DAG.

Our approach is to extend the notion of $EUniqCalls$ to each OR node. We label each OR node, EQ, with the set of parameters $y_1, y_2, \ldots, y_k$ that are used in the subtree rooted at EQ, and overload $EUniqCalls(EQ)$ to denote the expected number of unique calls as a result of the bindings for parameters $y_1, y_2, \ldots, y_k$. Note that EQ may be shared across query plans for multiple schema-tree nodes. Let $M = \{m_1, m_2, \ldots, m_l\}$ be the set of schema-tree nodes whose query plans use EQ. As before, we first obtain the independent subsequence of parameters $y_1, y_2, \ldots, y_{k'}$. Then,

$$EUniqCalls(EQ) = min\left(\prod_{i=1}^{k'} \left(\sum_{a \in M'} EUniqBind(y_i, a)\right), \sum_{b \in M} EVis(b)\right)$$

where, for any set of schema-tree nodes M, M' is the set of nodes in M whose parent nodes are not included in M. The second term is the upper bound of all visits to EQ from schema-tree nodes that use it in their query plans. The above is a conservative estimate of the unique calls to EQ, which may lead to sub-optimal plans sometimes; a better heuristic of estimating the calls will be addressed in future work.

```
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 8: XML *view query for experiments*

For every AND node (relational or virtual DOM operator), o that is a child of OR node EQ, we use standard query processing techniques to estimate its cost $ExecCost(o)$. To compute the aggregate cost of node o across invocations with different bindings, we multiply $ExecCost(o)$ by $EUniqCalls(EQ)$.

The bottom-up process of costing is also an optimization algorithm, since for each of the OR nodes, we keep the cost corresponding to the minimum-cost child among all its children. The process just outlined gives the plan for the query for each tag. Common subexpressions in the resulting plan are materialized; incorporating a greedy algorithm to consider the benefit of potential materialization as proposed in [22] is left for future work.

7 Performance

In this section, we present the results of our performance study on the ROLEX prototype. After describing our implementation and experimental settings, we investigate the utility of optimizing plans for navigation profiles and the impact of view-query complexity on the number of distinct plans produced by differing navigation profiles. Finally, we return to the higher-level issue of the overall performance potential of the virtual DOM approach.

7.1 Implementation

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.

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 Section 5.3.

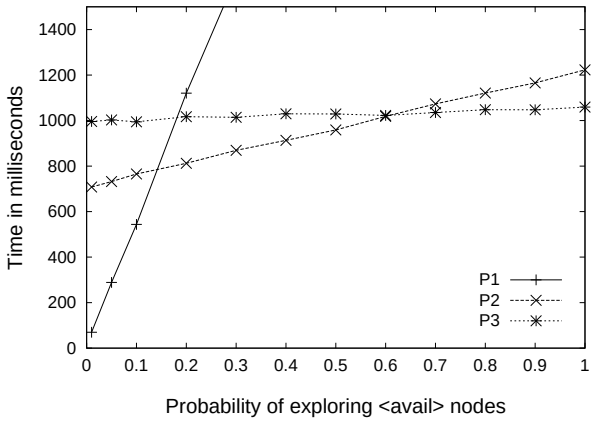

Figure 9: *Performance of plans $P1$ through $P3$ as a function of navigation probability for the view query in Figure 8*

7.2 Experimental Setting

In our experiments, plans were generated by the ROLEX optimizer, and executed on a Sun Ultra 60 with two 295MHz CPUs running SUNOS 5.7. The schema is as shown in Figure 2, with record sizes and tuple cardinalities as given in Table 1. Though the database is entirely resident in RAM the bigger tables are significantly larger than CPU cache.

Presently, the DOM interface layer has been developed only as a proof-of-concept independent of the execution engine. Thus, for these experiments, a driver was built on the execution engine to simulate an in-order scan that obeys a set of navigation probabilities – that is, a schema child of a node is visited only if the probability specified for that child is met by a random test. As a result, re-traversal cost of already-computed results is not measured.

7.3 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 8. 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. The actual performance of each of these plans as a function of navigation probability is shown in Figure 9. The figure shows that the three plans $P1$, $P2$, and $P3$ actually are optimal in the

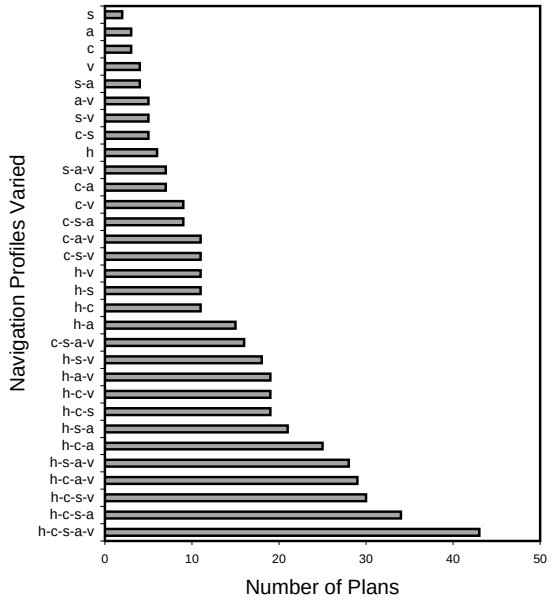

Figure 10: *Number of plans from navigation profiles*

ranges $[0, 0.15)$, $[0.15, 0.6)$, and $[0.6, 1]$, respectively. 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. 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 9. The error in the probability cutoff, attributed to cost-model variances, leads to a 5% to 15% sub-optimal execution.

However, Figure 9 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.

7.4 Number of Plans Generated

As seen in the above experiment, relatively few plans are generated by varying the navigational profile on a small query. So, in our next experiment, we show how the number of plans found changes as the complexity of the query increases. For this, we varied the navigational probabilities of a set of nodes in the view query of Figure 3, while keeping the other nodes at probability 1.0 (or 0.0). At each instance of these probabilities, we optimized the view query in ROLEX, and thereafter counted the number of different plans obtained. The results of this experiment, when the probability of the other nodes was kept at 1.0, are shown in Figure 10. Similar results were seen when the probabilities of the other nodes were set at 0.

In this figure, the binding variables shown in Figure 3 are used to indicate which navigational probabilities are being varied. For example, we varied the probabilities of exploring `<hotel>`, `<confstat>` and `<hotel_available>`, and the result is labeled as "h-s-a" in Figure 10. The probabilities were varied such that the number of visits to a node from its parent ranged exponentially from $0, 1, 2, 4, \ldots, n$, where n is the maximum number of visits (probability 1.0). Since we varied the probabilities of 5 nodes, about 3200 samples were generated.

From this experiment, we see that the number of distinct execution plans generated for a given view query can be large. For Figure 3, we found 43 different plans when we varied the navigational profiles of all the 5 nodes. The experiment demonstrates that many distinct plans can be generated when just a few probabilities are varied. For example, when the navigational profile of only `<hotel>` was varied, we got upto 6 different execution plans. This experiment supports the idea that, as view queries become complex, optimizing for specific navigational profiles will become increasingly important.

7.5 The Virtual-DOM Approach

In our final experiment, we deviate from our focus on query optimization and use the ROLEX prototype to evaluate the potential of the virtual DOM approach to compete with application caches of XML data. In particular, we compare the time taken for *execution and traversal* of a ROLEX query to the time taken to *parse* the result of the same query in the most mature C++ XML parser available that supports DOM, Xerces V1.6 [26]. To perform this experiment, we use a less selective version of the view query shown in Figure 8 (the date cutoff is changed to "10/05/02") so that larger results can be obtained. We vary the navigation probability as in the first experiment to vary the size of the result (from about 2000 elements to 125386 elements), and produce a file containing these elements for parsing. Furthermore, for each tuple returned, we include only the ROWID of the tuple in the output element, as the parse would take longer for additional attributes or subelements in the output. The parser was compiled in its "optimized" mode, the native transcoder was used, and no DTD-validation was performed. We used both the traditional DOM implementation and newer "IDOM" variant provided by Xerces. The results of this experiment are shown in Figure 11.

Since this experiment compares two very dissimilar activities, parsing and query execution, fine conclusions should not be drawn from the results. Our conclusions are simple: 1) ROLEX is a viable alternative to caching XML files in the application-tier and 2) the virtual DOM approach is likely to dominate the performance of XML middleware supporting only an XML text interface. Of course, the application can consider caching their documents as memory-resident DOM objects, which can be traversed very quickly. We plan to compare the in-memory performance of DOM

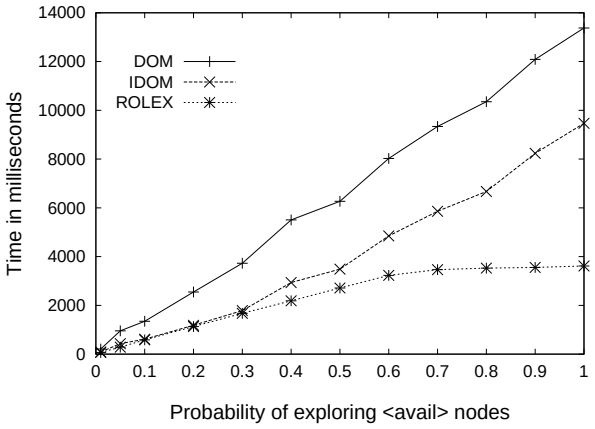

Figure 11: ROLEX *vs. parsing of view query in Figure 8*

to retraversal of an evaluated query in future work, but anticipate that DOM-like pointer structures might be required to match this performance. Finally, we note that complex (e.g., aggregate) queries with small output can always be constructed such that query execution would be arbitrarily worse than parsing (but ROLEX will not necessarily be worse than middleware solutions which also need to execute the query). However, such queries are typically part of OLAP workloads, not the OLTP workloads at which ROLEX is targeted.

8 Related Work

This work is closely related to previous work on publishing relational data as XML views [5, 9]. These middleware systems propose a view-definition language to specify the mapping betwen XML and relational data, and compose application queries with the view definition, a model that we have adapted in ROLEX. In terms of the schema-tree notation, these middleware systems transform application queries into SQL queries using outer unions among the tag queries corresponding to sibling nodes in the schema-tree and outer joins between queries corresponding to parent and child nodes. The optimization of this query is then left to the underlying database optimizer, and the tuple stream obtained as the output of the query is tagged to produce the XML result document.

Our approach differs substantially from [5, 9]. First, the results produced by the above middleware systems almost invariably require parsing by the application and are cached by the application. In the ROLEX approach, support for virtual DOM eliminates the need for tagging and parsing, and has the potential of providing database consistent views of the data more easily. Second, these systems assume that the result of the query is output in full and do not consider DBMS mechanisms to support navigation over the output document. In our approach, the optimizer is cognizant of navigation profiles, and the optimized plan has lower expected resource utilization.

In [10], the authors present an optimization model for declaratively specified web-sites. This work conceptualizes a web-site as a graph, and associates a parametric query with each arc. It models the probability distribution of reaching each node, and the conditional probability of traversal of an arc given that the parent has been traversed. While from a different domain, their work bears on transforming and optimizing XML-view queries, and like our work models navigation probabilities. However, the system described in [10] is built outside the DBMS and based on heuristics; thus it does not define an optimization space.

Virtual mediators (or information integrators over heterogenous backends) proposed in [17] translate client navigation into navigations on lower-level mediators or wrapped sources. This can be thought of as the client-side counterpart of the virtual DOM mechanism discussed in the paper. A major distinction of our work from [17] is that they focus on lazy execution (rather than optimization) for heterogeneous sources while we optimize XML-view queries against local relational data taking navigational profiles into account. In [1, 7], XML documents are mapped to an object-oriented data model, and the DOM interface is directly supported by the database, which provides persistence, transactions, indexing, etc. However, these systems do not provide relational interoperability and do not optimize queries for navigational profiles.

9 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.

Through its support for navigational access, ROLEX is able to return DOM subtrees lazily as the application executes. Further, ROLEX accepts a navigational profile associated with a view query and uses this profile in a cost-based optimizer to choose a best-cost *navigational* query plan. The novel optimization plan-space includes a variety of correlated and decorrelated executions of each subquery, using VOLCANO's common sub-expression detection to prevent a blow-up in optimization complexity. Further, the optimizer aggressively materializes sub-expressions across repeated calls, and this is reflected in our cost model for deeply nested, navigable, correlated queries. The current ROLEX system prototype was used in an experimental study to show that accounting for navigation can lead to far better plans than assuming full materialization, and that plans optimized for a given probability work reasonably well at "nearby" probabilities. We also evaluated how navigational profiles interacted with query-tree complexity by optimizing a more complex query over a large space of such profiles. As probabilities were varied along more edges, the number of "best" plans found by the optimizer grew substantially, suggesting that the importance of optimizing for navigation will grow along with the complexity of XML ap-

plications.

Although the focus of this paper is on optimization and performance, the ROLEX architecture has other benefits as well – benefits to be exploited fully as part of future research. Most noteworthy among these is the ability to facilitate the maintenance of data consistency despite access by both relational and XML applications to the data. Since ROLEX views reference the relational data itself, the concurrency control of the relational database system can be employed to enforce whatever isolation level is deemed appropriate for the application. Any updates generated by an application using an XML view face the semantic issues of view update that face any relational system, but avoid the data-currency problems that arise in current cache-based XML-publishing systems. In addition to addressing update consistency, we plan to address 1) more complex navigational profiles, 2) multi-query optimization strategy in complex XML views, and 3) the potential benefits of pushing functionality from an XSLT processor working on the ROLEX virtual DOM interface into the query engine.

Finally, initial experimental results comparing execution of ROLEX queries to parsing the results of those queries from XML show that ROLEX has the ability to eliminate caches of XML data drawn from the DBMS by supporting optimized views. In doing this, ROLEX has the potential to bring database technology to the front-line of electronic-commerce implementations.

Acknowledgements
The authors would like to thank Prasan Roy for support of the VOLCANO-style optimizer on which ROLEX is based.

References

[1] S. Abiteboul et al. XML repository and active views demonstration. In *Proc. of 25th Int'l. Conf. on Very Large Data Bases*. Morgan Kaufmann, 1999.

[2] 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.

[3] P. Bohannon, J. Freire, P. Roy, and J. Simeón. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of Int'l Conf. on Data Engineering*, 2002.

[4] P. Bohannon, H. Korth, and P. P. S. Narayan. The table and the tree: On-line access to relational data through virtual XML documents. In *Proc. of WebDB*, 2001.

[5] 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.

[6] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. of Int'l Conf. on Very Large Data Bases*, 1987.

[7] eXcelon Corporation. An XML data server for enterprise web applications. (White Paper) www.exceloncorp.com/products/white_papers.html.

[8] 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.

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

[10] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web sites. In *Proc. of the Int. Conf. on Very Large Data Bases*, 1999.

[11] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queried revisited. In *Proc. of 19th ACM SIGMOD Conf. on the Management of Data*, May 1987.

[12] E. Gelenbe and D. Gardy. The size of projections of relations satisfying a functional dependency. In *Proc. of the Int'l Conf. on Very Large Data Bases*, 1982.

[13] G. Graefe and W. McKenna. Extensibility and search efficiency in the Volcano Optimizer Generator. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, 1993.

[14] 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/.

[15] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, Sept. 1982.

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

[17] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven evaluation of virtual mediated views. *Lecture Notes in Computer Science*, 1777, 2000.

[18] OASIS-OPEN. Oasis, the organization for the advancement of structured information standards. http://www.oasis-open.org. OASIS-Open web site.

[19] J. Rao and K. A. Ross. Reusing invariants: a new strategy for correlated queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 1998.

[20] M. Rhys. State-of-the-art XML support in RDBMS: Microsoft SQL Server's XML features. *Bulletin of the Tech. Com. on Data Engineering*, 24(2):3–11, June 2001.

[21] Rosetta-Net. Rosettanet: Lingua franca for business. http://www.rosettanet.org. RosettaNet web site.

[22] P. Roy. *Multi Query Optimization and Applications*. PhD thesis, Indian Institute of Technology, Bombay, 2001.

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database system. In *Proc. of ACM-SIGMOD Int'l Conf. on Management of Data*, 1979.

[24] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proc. of the 12th International Conference on Data Engineering*, 1996.

[25] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the Int'l. Conf. on Very Large Databases*, 1999.

[26] The-Apache-Software-Foundation. Xerces C++ parser. http://xml.apache.org.

[27] The-Times-Ten-Team. In-memory data management in the application tier. In *Proc. of the 16th Int'l. Conf. on Data Engineering (ICDE' 00)*, 2000.