

Adaptive XML Shredding: Architecture, Implementation, and Challenges

Juliana Freire* and Jérôme Siméon

Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA
{juliana, simeon}@research.bell-labs.com

1 Introduction

As XML data becomes central to business-critical applications, there is a growing need for efficient and reliable XML storage. Two main approaches have been proposed for storing XML data: native and colonial systems. *Native systems* (e.g., [9, 20]) are designed from the ground up specifically for XML and XML query languages. *Colonial systems* (e.g., [5, 7, 19]), on the other hand, attempt to reuse existing commercial database systems (DBMS) by mapping XML into the underlying model used by the DBMS. Colonial systems can thus leverage features, such as concurrency control, crash recovery, scalability, and highly optimized query processors available in the DBMS, making them an attractive alternative for managing XML data. However, several technical challenges need to be addressed in terms of architecture, algorithms, and implementation of these systems. In this paper, we described how these issues are addressed in the context of colonial systems that use relational databases as the underlying DBMS.

The mismatch between the XML and the relational models implies that one must first *shred* an XML tree-structured document so that it *fits* into flat relational tables. Therefore, a mechanism is needed to determine the appropriate storage configuration. Once a mapping is selected, the system must provide support for loading the XML data into the database, and to translate queries over the original document into queries over the mapped data. There are different approaches for these problems. For example, while commercial relational systems require users to manually define mappings [14, 15], techniques have been proposed to automatically derive XML-to-relational mappings that adopt either a fixed shredding strategy [19, 11] or that derive the best shredding for a given application [5, 4]. Different techniques have also been proposed for query translation [10, 6].

Although individual problems pertaining to colonial XML storage systems have been studied in isolation, to the best of our knowledge, the design and implementation of a complete colonial system has not been described in the literature. In this paper, we discuss the design and implementation of LegoDB [5], a colonial XML data management system. In particular, we present the complete architecture of the system, its implementation, and the describe underlying algorithms.

* Current address: juliana@cse.ogi.edu

The paper is organized as follows. Section 2 motivates the need for adaptive XML shredding. Section 3 presents the general architecture of a colonial system that supports adaptive shredding. Sections 4, 5 and 6 describe the components of such an architecture and their implementation in the LegoDB system.

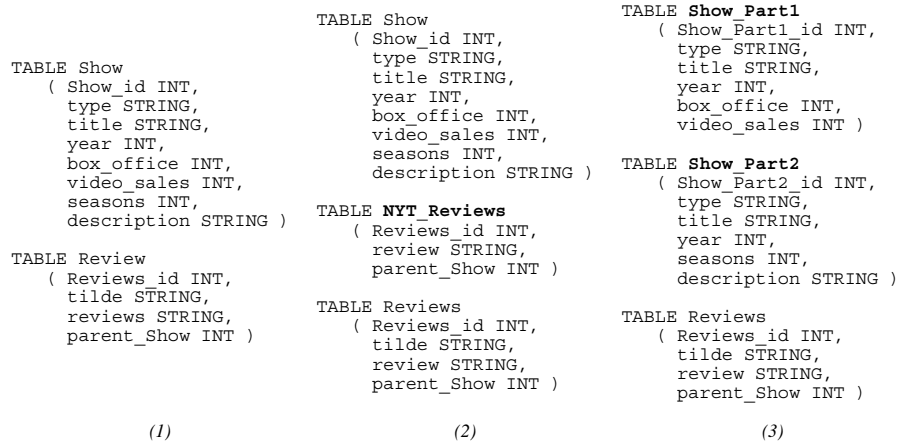


Fig. 1. Three storage configurations for the movie database

2 The Need for Adaptive Shredding

In this section, we motivate the need for adaptive shredding through a scenario inspired from the Internet Movie Database (<http://www.imdb.com>), which provides access to information about movies and television shows. A fragment of Document Type Definition (DTD) corresponding to this document is illustrated below.

```
<!ELEMENT imdb (show*, director*, actor*)>
<!ELEMENT show (title, year, reviews*,
                ((box_office, video_sales)
                 |(seasons, description, episode*)))>
<!ATTLIST show type CDATA #REQUIRED>

<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT review (#PCDATA)>
....
```

An IMDB document contains a collection of shows, movie directors and actors. Each show can be either a movie or a TV show. Movies and TV shows have some information in common (*e.g.*, title and year of production), but they also contain information that is specific to each (*e.g.*, movies have a `box_office` and `video_sales`, TV shows have `seasons`).

There are many different ways to store data that conforms with the IMDB DTD in a relational database. Figure 1 shows three alternatives. The first configuration results from inlining as many elements as possible in a given table, roughly corresponding to the shared strategy proposed in [19]. The second configuration is obtained from the first

by partitioning the `Reviews` table into two tables (one that contains New York Times reviews, and another for reviews from other sources). Finally, the third configuration is obtained from the first by splitting the `Show` table into `Show_Part1` for movies, and `Show_Part2` for TV shows.

Even though a given configuration can be efficient for one application, it may lead to poor performance for others. Thus, it is not possible to select the *best* configuration in isolation, without taking the application characteristics and *cost* into account. As an illustration, consider the following XQuery [3] queries:

```

Q1:
for $v in imdb/show
where $v/year = 1999
return ($v/title, $v/year,
        $v/nyt_reviews)

Q2:
for $v in imdb/show return $v

Q3:
for $v in imdb/show
where $v/title = c3
return $v/description

Q4:
for $v in imdb/show
return <result>
  { $v/title, $v/year,
    (for $e in $v/episode
     where $e/guest_director = c4
     return $e) }
</result>

```

Queries `Q1` and `Q2` are typical of a publishing scenario as in [10] (*i.e.*, to send a movie catalog to an interested partner). Queries `Q3` and `Q4` contain selection criteria and are typical of interactive lookup queries, such as the ones issued against the IMDB Web site itself. We then define two workloads, $W1 = \{Q1 : 0.4, Q2 : 0.4, Q3 : 0.1, Q4 : 0.1\}$ and $W2 = \{Q1 : 0.1, Q2 : 0.1, Q3 : 0.4, Q4 : 0.4\}$, where each workload contains a set of queries and an associated weight that reflects the importance of each query for the application.

	Configuration 1	Configuration 2	Configuration 3
Q1	1.00	0.83	1.27
Q2	1.00	0.50	0.48
Q3	1.00	1.00	0.17
Q4	1.00	1.19	0.40
W1	1.00	0.75	0.75
W2	1.00	1.01	0.40

Fig. 2. Performance of Different Configurations

Figure 2 shows the estimated costs for each query and workload, as returned by the LegoDB optimizer for each configuration in Figure 1 (costs are normalized by the costs of mapping 1). Only the first of the three storage mappings shown in Figure 1 can be generated by previous heuristic approaches. However, this mapping has significant disadvantages for either workload we consider. First, due to its treatment of union, it inlines several fields which are not present in all the data, making the `Show` relation wider than necessary. Second, when the entire `Show` relation is exported as a single document, the records corresponding to movies need not be joined with the `Episode` tables, but this join is required by the first two configurations. Finally, the large `Description` element need not be inlined unless it is frequently queried. The principle behind adaptive shredding is to automatically explore a space of possible relational configurations, and select the configuration that leads to the lowest cost for evaluating the application workload.

3 Principles and Architecture

A colonial XML storage system has two main components: storage design and run-time. The main task of the design component is to generate a target relational schema to store the input XML document. In adaptive shredding systems such as LegoDB, this component is rather complex. It takes into account information about the target application to both generate a space of possible mappings and evaluate the effectiveness of the derived mappings.

Figure 3(a) shows the architecture of the design component in LegoDB. The LegoDB storage engine takes as inputs: an XML Schema, an XQuery workload, and a set of sample documents. As output, it produces an *efficient* relational configuration (a set of relational tables) as well as a mapping specification. The modules of the storage design components are described below.

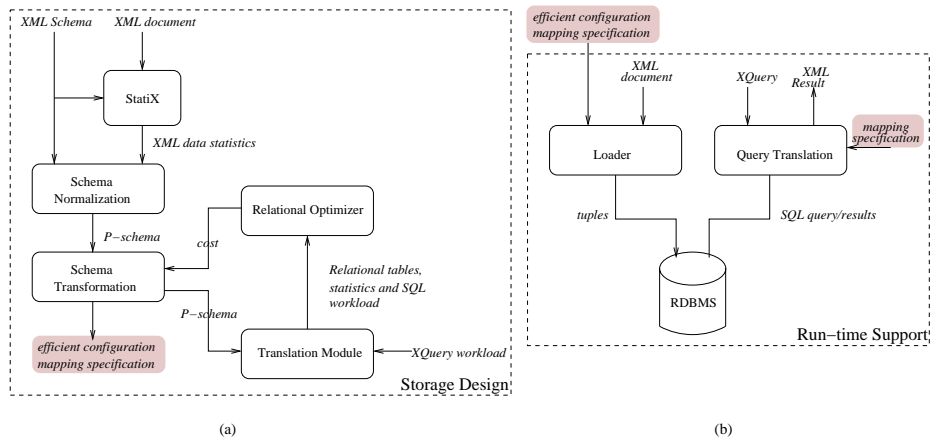


Fig. 3. Design and Run-Time Architectures

StatiX The first task in the system is to extract *statistical information* about both the values and structure of the input XML document. This information is used to derive *precise* relational statistics that are needed by the relational optimizer to accurately estimate the cost of the query workload. In LegoDB, this is done by the *StatiX* module, described in Section 5.

Schema Normalization The statistics together with the XML Schema are sent to the *Schema Normalization* module, which produces a *physical schema* (p-schema). P-schemas extend XML Schemas with statistical information, and have a structure such that each type defined in the schema can be directly mapped into a relational table (see Section 4).

Schema Transformation The system searches for an efficient relational configuration by repeatedly transforming p-schemas, *i.e.*, generating new p-schemas that are structurally different, but that validate the same documents. Each new p-schema corresponds to a possible relational configuration. XML Schema transformations and search algorithms are discussed in Section 6.

Translation Module For each p-schema, the *Translation Module* generates a corresponding relational schema, translates the XQuery workload into SQL, and derives the appropriate relational statistics.

Relational Optimizer LegoDB uses a standard relational optimizer for cost estimation as a black box. As a result, it is possible to use LegoDB with different databases whose optimizers have different cost-models. The quality of the derived configurations depends on the accuracy of the estimates computed by the optimizer.

The design module produces a specification for the mapping that has the lowest cost among the alternatives explored by LegoDB. This specification is then used by the Loader and Query Translation modules (see Figure 3(b)) to create and populate the corresponding tables in the RDBMS, and answer application queries.

Loader Given a mapping specification and an XML document, the Loader module populates the target relational database. As described in Section 4.3, the Loader extends an XML Schema validating parser to generate tuples a document is validated.

Query translator The *Query Translation* module, described in Section 4.2 is used to perform query translation on behalf of the target XML application.

Note that other tools for mapping XQuery to SQL (e.g., [10]) could be used in LegoDB.

4 XML-to-Relational Translation

4.1 Mapping schemas

LegoDB [5] generates a space of possible schema mappings by repeatedly transforming the original XML Schema, and for each transformed schema, applying a *fixed mapping* from XML Schema into relations. In this section, we describe how the fixed mapping is implemented. Details of the schema transformations and search algorithm are given in Section 6.

In order to guarantee the existence of a fixed mapping, we define the notion of *physical schema (p-schema)*. In a p-schema, each type name defines a structure that can be directly mapped to a relation. More precisely, this structure must be such that it contains only (possibly nested) singleton elements with a simple value; (possibly nested) optional elements with a simple value; and all complex regular expressions may only contain type names. This last condition ensures that complex regular expressions, such as union and repetition, do not contain actual values. As a result, a schema that verifies such criteria can be mapped to relations by creating a table for each type, and creating a column in that table for each element with a simple value. In the examples that follow, for simplicity, we use the type notation from the XQuery type system, described in [8]. A formal definition of p-schemas and the corresponding grammar appear in [5].

We now sketch the *normalization* algorithm used to generate a *p-schema* from an XML Schema. The algorithm is applied top-down on the structure of the type, and for each type in the original schema. For a type definition `define type X { T }`, where `X` is the name of the type, and `T` is the structure defining that type, the normalization algorithm is first applied on `T`. This returns a new type `T'` along with a set of new type definitions `define type X1 { T1 } ... define type Xn { Tn }`, where all of the

type T' , T_1 , ..., T_n are normalized. A given type structure T is normalized, recursively, as follows:

- If the type is an atomic type (e.g., `string`), return the type unchanged.
- If the type is an (optional) element declaration, then return the element declaration with its content normalized.
- If the type is a sequence of two types, return the sequence of their normalized types.
- If the type is a repetition (e.g., `element a*`), then insert a new type name with the normalized content of the original type (e.g., `X1*` with `define type X1 { element a }`).
- If the type is a union (e.g., `element a | element b`), then create a new type name for each component of the union with the contents of the original type normalized (e.g., `X1 | X2` with `define type X1 { element a }` and `define type X2 { element b }`).

After the original schema is normalized, transformations can be applied on the resulting p-schema. These transformations always result in a p-schema which can then be mapped into a relational configuration. The algorithm to map a p-schema into a set of relations is as follows:

- Create one relation R_X for each type name x ;
- For each relation R_X , create a key that stores the id of the corresponding element;
- For each relation R_X , create a foreign key parent P_X to all the relations R_{P_X} such that P_X is a parent type of x ;
- Create a column in R_a for each element a inside the type x that contains a value;
- If the data type is contained within an optional type then the corresponding column can contain a null value.

4.2 Mapping queries

The LegoDB prototype implements a simple version of query translation for a fragment of XQuery which consists of: simple path navigation, selections, joins, and nested queries. More sophisticated query mapping techniques such as the ones proposed in [10, 6] can be integrated into the system.

Translating an XQuery query into SQL requires analysis of the XPath expressions contained in the query and information about the schema mapping in order to determine the relational tables and columns to be accessed. In LegoDB, the mapping of XQuery to SQL is done in two phases. The first phase rewrites an XQuery XQ in a normal form XQ_{nf} . For example, the query:

```
for $show in document("www.imdb.com/imdb.xml")/show
where $show/year >= 2000
return $show/title, $show/reviews, $show/box_office
```

is normalized into:

```
let $imdbdoc := document("www.imdb.com/imdb.xml")
for $show in $imdbdoc/show, $v_title in $show/title,
    $v_box in $show/box_office, $v_year in $show/year,
    $v_reviews in $show/reviews
where $v_year >= 2000
return ($v_title, $v_reviews, $v_box)
```

XQ_{nf} is then translated into an equivalent SQL query for the given p-schema:

- *SELECT clause.* For each variable v in the XQuery return clause, if v refers to a type in the p -schema, all attributes of the corresponding table are added to the clause. Otherwise, if v refers to an element with no associated type, the corresponding attribute is added to the clause.
- *FROM clause.* For each variable v mentioned in the XQuery (in both where and return clauses), if v refers to a type T in the p -schema, the corresponding table R_T is added to the clause.
- *WHERE clause.* Conditions in where clause are translated in a straightforward manner, by replacing variable occurrences with the appropriate column name. In addition, for each variable in the XQuery, if the path expression defining the variable includes elements in that are mapped into separate tables, a condition must be added to enforce the key/foreign-key constraint.

For example, given the third configuration in Figure 1, the query mapping algorithm generates the following SQL Query:

```
SELECT S1.title, S1.box_office, R.Reviews_id
FROM Show_Part1 S1, Reviews R
WHERE S1.Show_Part1_id = R.parent_Show AND S1.year >= 2001
```

4.3 Mapping data

After a mapping is selected by LegoDB, the next step is to create the database tables and load the XML documents. LegoDB extends an XML Schema validating parser to generate tuples as documents are validated.¹ Validation is a basic operation for XML documents. The most important property we use from XML Schema validation is the notion of type assignment: during validation each node in the XML tree is assigned a unique type name from the schema. Since the mappings are type-based, and one relation is created per type, the parser outputs a tuple for each instance of type encountered during validation. We describe below the process used in the current prototype to generate insert statements that populate the target database.

- Validation is performed top down starting at the root of the document.
- For each type name in the schema, a structure that represents a tuple is kept in memory to store the current part of the type which is validated. This structure is created based on the fixed mapping described in Section 4.
- Each time validation reaches a value that corresponds to an attribute of a tuple, the tuple is populated accordingly.
- When the tuple is full, it is flushed on disk as an insert statement.
- This process is repeated until the document is fully validated.

Note that currently this phase is done in LegoDB using a batch file. It is also possible to directly populate the database using ODBC, or to use more efficient bulk-loading facilities.

¹ In LegoDB, both data loading and statistics gathering are performed during document validation (see Section5).

5 Statistics and Cost Estimation

LegoDB uses a standard relational optimizer [18] for cost estimation. But in order to derive accurate cost estimates, the optimizer needs accurate statistics. In LegoDB, the StatiX module is responsible for gathering statistics about the input XML document. As described in [12], it is important that these statistics capture both value and structural skews present in the data, and that they contain enough information so that accurate relational statistics can be derived as transformations are applied to the XML Schema.

Similar to data loading, statistics gathering is performed simultaneously with document validation. StatiX gathers statistics on a *per-type* basis. As a document is validated, globally unique identifiers (IDs) are assigned to all instances of the types defined in the schema; and together with these ID assignments, the system keeps track of the cardinality of each edge in the XML Schema type graph. Using this information, *structural histograms* constructed which use the IDs to summarize information about how elements are connected. Note that, while StatiX uses histograms in a novel way to summarize structural information, histograms are also used in a more traditional sense: *value histograms* can be built for types that are defined in terms of base types (e.g., Integer).

Statistics gathering proceeds as follow. First, each distinct type in the XML Schema is assigned a unique type ID off-line. Then, for each type ID, we maintain a structure which contains: (1) a counter for the next available ID for that type, and (2) a set of all parent IDs that have been processed for that type. During document validation, the validation function is given the current parent ID. Whenever a new instance of the type is encountered, it is assigned a local ID using the sequential counter associated with the type, and the current parent ID is added to the parent set for the corresponding type. The global ID of the element is then simply obtained by concatenating the associated type id and its local id. The modified part of the standard validation procedure is given in pseudo-code below.

```
/* Type struct is a counter plus parent types */
type TypeStructure = { counter : integer; parents : [ID] }

/* We maintain a type structure for each type */
AllTypes := [ (typename, type_structure) ]

/* Maintains the type structure and returns the new type ID */
fun add'to'type(T : Type, ParentID : ID)
{ TS := find'type'structure(AllTypes, T); /* get the type structure for T */
  NewID := build'id(T, TS.counter); /* creates new ID */
  TS.counter := TS.counter+1; /* increments the counter */
  TS.parents := add(ParentID, TS.parents) /* adds parent ID */
  return NewID; }

/* Here is a part of the validation function */
fun validate'sequence(S : Sequence, R : RegExp, ParentID : ID)
{ Deriv := derivatives(R); /* computes the derivatives */

  Node := first(S); /* get the first node */
  (NodeType, R') := match(Deriv, Node); /* find the node type for the node */

  CurrentID := add'to'type(NodeType, ParentID); /* get the ID of the node */

  validate'node(Node, NodeType, CurrentID); /* Validate the children of the node */
  validate'sequence(rest(S), R', ParentID); /* validates the rest */ }
```

The structural information gathered during validation can be summarized using standard histogram techniques. A variety of histogram constructions have been described in the literature [17] – the most common are *equi-width* histograms, wherein the domain range covered by each histogram bucket is the same, and *equi-depth* histograms, wherein the frequency assigned to each bucket is the same. Since it has been shown in [16] that equi-depth histograms result in significantly less estimation error as compared to equi-width histograms, we have implemented the former in StatiX.

6 Searching

In this section, we describe how a space of alternative shreadings is constructed and techniques to search for the best shredding for a given application.

6.1 Transforming P-Schemas

There are many different schemas that validate a given document. For instance, different but equivalent regular expressions (e.g., $(a(b|c^*))((a,b)|(a,c^*))$) can describe the contents of a given element. In addition, the presence or absence of a type name does not change the semantics of the XML Schema.

By transforming regular expressions that define XML elements, and by adding and removing types, LegoDB derives a series of equivalent schemas. These schemas have *distinct type structures* that when mapped into relations using the fixed type-to-relation mapping (Section 4), lead to distinct relational configurations. There is large number of possible transformations that can be applied to XML Schemas, and we describe some below. For a more comprehensive discussion on transformations used in LegoDB, the reader is referred to [5].

Inlining/Outlining. In an XML Schema, associating a type name to a given nested element (outlining) or nesting its definition directly within its parent element (inlining) does not change the semantics of the schema, *i.e.*, the modified schema validates the same set of documents. However, rewriting an XML Schema in that way impacts the mapped relational schema by inlining or outlining the corresponding element within its parent table. Inlining is illustrated below:

```

define type TV {
  element seasons { int },
  type Description,
  type Episode*
}
define type Description {
  element description { string }
}

define type TV {
  element seasons { int },
  → element description { string },
  type Episode*
}

```

At the relational level, this rewriting corresponds to the following transformation:

```

TABLE TV
( TV_id INT,
  seasons STRING,
  parent_Show INT)

TABLE Description
( Description_id INT,
  description STRING,
  parent_TV INT)

TABLE TV
( TV_id INT,
  seasons STRING,
  → description STRING,
  parent_Show INT)

```

Note that inlining is the basis for the strategies proposed in [19]. It reduces the need for joins when accessing the content of an element, but at the same time it increases the size of the corresponding table and the cost of retrieving individual tuples. In the example above, the benefits of inlining or outlining `description` element within the `TV` type depend both on the access frequency for this element in the workload as well as its length.

Union Factorization/Distribution. Union types are often used to add some degree of flexibility to the schema. As queries can have different access patterns on unions, *e.g.*, access either parts together or independently, it is essential that appropriate storage structures are derived. LegoDB uses simple distribution laws on regular expressions to explore alternative storage structures for union. The first law $((a, (b|c)) == (a, b | a, c))$ allows distribution of a union within a regular expression. The second law $(a[t1|t2] == a[t1] | a[t2])$ allows to distribute a union over an element (crossing element boundaries). For example, in Figure 1, applying these two laws on configuration 1 leads to configuration 3. The union transformations highlight the advantages of working directly at the XML Schema level. The horizontal partitioning of the relational schema derived from the configuration (3) in Figure 1 would not be easily found by a relational physical-design tool, since the information about the set of attributes involved in the union would not be available in the relational setting.

6.2 Searching for the Best Shredding

By repeatedly applying schema transformations, LegoDB generates a space of alternative *p-schemas* and corresponding relational configurations. Since this space can be very large (possibly infinite), it is not feasible to perform an exhaustive search. As shown in Algorithm 6.1, LegoDB uses a greedy heuristic to prune the search space. The algorithm begins by deriving an initial configuration *pSchema* from the given XML Schema *xSchema* (line 3). Next, the cost of this configuration, with respect to the given query workload *xWkld* and the data statistics *xStats* is computed using the function *GetPSchemaCost* which will be described in a moment (line 3). The greedy search (lines 5-16) iteratively updates *pSchema* to the cheapest configuration that can be derived from *pSchema* using a single transformation. Specifically, in each iteration, a list of candidate configurations *pSchemaList* is created by applying all applicable transformations to the current configuration *pSchema* (line 7). Each of these candidate configurations is evaluated using *GetPSchemaCost* and the configuration with the smallest cost is selected (lines 8-14). This process is repeated until the current configuration can no longer be improved. Note the use of the function *ApplyTransformations*, which applies the schema transformations described above.

GetPSchemaCost computes the cost of a configuration given a *pSchema*, the XML Query workload *xWkld*, and the XML data statistics *xStats*. First, *pSchema* is used to derive the corresponding relational schema (Section 4). This mapping is also used to translate *xStats* into the corresponding statistics for the relational data (Section 5), as well as to translate individual queries in *xWkld* into the corresponding relational queries in SQL (Section 4). The resulting relational schema and the statistics are taken as input by a relational optimizer to compute the expected cost of computing

Algorithm 6.1 Greedy Heuristic for Finding an Efficient Configuration

```
Procedure GreedySearch
Input:  xSchema : XML Schema,
        xWkld  : XML query workload,
        xStats : XML data statistics
Output: pSchema : an efficient physical schema
1 begin
  minCost = ∞;
  pSchema = GetInitialPhysicalSchema(xSchema)
  cost = GetPSchemaCost(pSchema, xWkld, xStats)
5  while (cost < minCost) do
    minCost = cost
    pSchemaList = ApplyTransformations(pSchema)
    for each pSchema' ∈ pSchemaList do
      cost' = GetPSchemaCost(pSchema', xWkld, xStats)
10     if cost' < cost then
       cost = cost'
       pSchema = pSchema'
    endif
  endfor
15 endwhile
  return pSchema
end.
```

a query in the SQL workload derived as above; this cost is returned as the cost of the given $pSchema$.

Note that the algorithm does not restrict the kind of optimizer used (transformational or rule-based, linear or bushy, etc. [13]); although, obviously, the optimizer should reflect the actual costs in the target relational system. As a result, different database systems that use different optimizers and cost-models can easily be connected to LegoDB. Also note that physical design tools such as DB2 Advisor [21] and Microsoft's Tuning Wizard [2, 1] are complementary to LegoDB.

The first prototype of LegoDB implements a greedy search over inline and outline transformations, and experiments show this strategy is both efficient and effective in practice [5].

7 Discussion

In this paper, we discussed the main issues involved in building a colonial XML storage system based on adaptive shredding. We presented the architecture of LegoDB, described the implementation of its various components, and the underlying algorithms. The prototype implementation described here has been demonstrated at VLDB 2002 [4].

Initial experiments have shown that LegoDB is able to derive efficient relational configurations in a reasonable amount of time. We are currently investigating alternative search strategies that, in addition to inlining and outlining, also considers other transformations (e.g., union distribution, repetition split, etc). When additional transformations are considered, there is an explosion in the size search space, and a simple greedy strategy may miss interesting configurations.

In our prototype, we use the optimizer developed by Roy et al [18] to obtain cost estimates. In a *real* application, the optimizer should reflect the actual costs in the target relational system. We are currently investigating how to connect LegoDB to commercial RDBMS so that cost estimates can be obtained from these systems.

Acknowledgments. We would like to thank Jayant Haritsa, Maya Ramanath, Prasan Roy and Philip Bohannon for their invaluable contributions to the LegoDB project.

References

1. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. of VLDB*, 2000.
2. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Materialized view and index selection tool for microsoft sql server 2000. In *Proc. of SIGMOD*, 2001.
3. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. W3C Working Draft, June 2001.
4. P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Legodb: Customizing relational storage for xml documents. In *Proc. of VLDB*, 2002.
5. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of ICDE*, 2002.
6. M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as xml documents. In *Proc. of VLDB*, 2000.
7. A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with STORED. In *Proc. of SIGMOD*, 1999.
8. D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. The XQuery 1.0 formal semantics, March 2002. W3C Working Draft.
9. Excelon. <http://www.exceloncorp.com>.
10. M. Fernandez, W. C. Tan, and D. Suciu. Silkroute: trading between relations and XML. *Computer Networks*, 33(1-6):723–745, 2000.
11. D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML in a relational database. Technical Report 3680, INRIA, 1999.
12. J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: Making XML count. In *Proc. of SIGMOD*, 2002.
13. G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proc. of ICDE*, 1993.
14. IBM DB2 XML Extender. <http://www-3.ibm.com/software/data/db2/extenders/xmlxt/library.html>.
15. Oracle XML DB. <http://technet.oracle.com/tech/xml>.
16. G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of SIGMOD*, 1984.
17. Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2000.
18. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD*, 2000.
19. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, 1999.
20. Tamino. <http://www.softwareag.com/tamino>.
21. G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proc. of ICDE*, 2000.