University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

January 2003

# XML Vectorization: A Column-Based XML Storage Model

Byron Choi
*University of Pennsylvania*

Peter Buneman
*University of Pennsylvania*

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Byron Choi and Peter Buneman, "XML Vectorization: A Column-Based XML Storage Model", . January 2003.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-03-17.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/45
For more information, please contact libraryrepository@pobox.upenn.edu.

# XML Vectorization: A Column-Based XML Storage Model

**Abstract**
The usual method for storing tables in a relational database is to store each tuple contiguously in secondary storage. A simple alternative is to store the columns contiguously, so that a table is represented as a set of vectors all of the same length. It has been shown that such a representation performs well on queries requiring few columns. This paper reviews the shredding scheme used in XMill, an XML compressor, which represents the document structure by using a set of files, consisting of a file describing the structure, and files describing the character data to be found on designated paths (corresponding to the column data). We consider such a shredding as a storage model –- XML vectorization –- by presenting an indexing scheme and a physical algebra associated with a detailed cost model. We study query processing on the XML vectorization, in particular the XML join queries. XML join queries are often translated into a few relational join operations in the relational-based XML storage systems. The use of columns enables us to develop a fast join algorithm for vectorized XML based on two hashbased join algorithms. The important feature of the join algorithm is that the disk access of the algorithm is mostly sequential and the data not needed are not read from disk. Experimental results demonstrate the effectiveness of the join algorithm for vectorized XML.

**Comments**
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-03-17.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/45

# XML Vectorization: a Column-Based XML Storage Model

Byron Choi
University of Pennsylvania
kkchoi@seas.upenn.edu

Peter Buneman
University of Pennsylvania
peter@cis.upenn.edu

## Abstract

The usual method for storing tables in a relational database is to store each tuple contiguously in secondary storage. A simple alternative is to store the columns contiguously, so that a table is represented as a set of vectors all of the same length. It has been shown that such a representation performs well on queries requiring few columns. This paper reviews the shredding scheme used in XMill, an XML compressor, which represents the document structure by using a set of files, consisting of a file describing the structure, and files describing the character data to be found on designated paths (corresponding to the column data). We consider such a shredding as a storage model – XML vectorization – by presenting an indexing scheme and a physical algebra associated with a detailed cost model. We study query processing on the XML vectorization, in particular the XML join queries. XML join queries are often translated into a few relational join operations in the relational-based XML storage systems. The use of columns enables us to develop a fast join algorithm for vectorized XML based on two hash-based join algorithms. The important feature of the join algorithm is that the disk access of the algorithm is mostly sequential and the data not needed are not read from disk. Experimental results demonstrate the effectiveness of the join algorithm for vectorized XML.

## 1 Introduction

Almost all commercial database systems only support N-ary storage model – storing tuple contiguously in the secondary storage. In the absence of any indexes, a relation query on a set of tables has, generally, to read all those tables at least once. However, most queries use only some attributes of a tuple. For such queries, a large fraction of the I/O is unnecessary. A simple alternative is to store the columns contiguously, so that a table is represented as a set of vectors all of the same length. Such a column-based storage model has been proposed [8, 2, 4, 1]. A good side of such a model is that data not needed are not read. Unfortunately, queries that use a few data vectors require additional work to join the data vectors together.

The amount of information encoded in XML has been tremendous. There is a need to store and query XML documents efficiently. Recent work has been proposed to reuse the relational databases [10, 3, 18, 7] to store XML. The general approach is to shred the XML document into a set of tuples and store them in a relational database. Unfortunately, XML is semi-structured. Depending on how an XML document is queried, there are many possible reasonable shreddings [3]. It is known that certain shredding schemes perform well on certain class of queries.

A shredding scheme (XMill shredding scheme) used in XMill [17], an XML compressor, is originally designed only for XML compression. In order to compress the XML document effectively, the character data and the tags are separated. The character data is stored in the data file according to the path where the data is found. Each data file is a *data column* (a *data vector*). Tags are replaced by integers. The resultant structure of these operations, namely the skeleton of the document, is essentially a sequence of integers. The data files and the skeleton are compressed separately. It is found that such organization of the document yields good compression ratio. This paper bridges the XMill shredding scheme and XML query processing. In particular, this shredding scheme works very well on an important class of queries in databases – the join queries.

Unlike XMill, this work applies compression with considerations on query processing. Compression is not applied on the data columns but indexes (eg. B+ trees) are built. The skeleton is indexed and compressed. In this paper, we name the storage model based on the XMill shredding scheme the *XML vectorization* and the corresponding shredded document the *vectorized XML*.

In this work, we study query processing on the XML vectorization. It has similar properties to the relational column-based storage. It performs well on queries using few vectors since the data not needed are not read. We study the XML join queries which appear in almost all queries in W3C XQuery User Cases section 1.4 – Access to Relational Data [6]. Such queries are often translated into

a few relational joins in the systems implemented on top of a relational database. In contrast, we developed a fast algorithm for evaluating the XML join queries. The algorithm always performs 4 scans on the skeleton of the document and 1 join on the participating data columns.

Our main contributions include:

- we propose a storage model based on the XMill shredding scheme. We develop an index structure, a cost model and a physical algebra for the storage model;
- we develop a fast join algorithm based on two hash-based join algorithms. The I/O required by the algorithm is mostly sequential;
- we experimentally illustrate the performance boundaries between the vectorization and some other XML storage schemes.

The rest of the paper is organized as follows. The next subsection reviews some XML storage schemes. Section 2 reviews the XMill shredding scheme, proposes the XML vectorization and presents an indexing scheme and a detailed cost model for simple path queries. We discuss a fast join algorithm for the vectorized XML and study the detailed cost model of the algorithm in Section 3. We provide the relevant physical algebra in Section 4. Section 5 validates the cost model experimentally and provides a comparison to some other XML storage schemes and the result of a performance of our join algorithm. We discuss the implementation experience for the vectorization in Section 6. The conclusions can be found in 7.

### 1.1 Background

In this subsection, we briefly review some storage models which are based on the reuse of relational database. These storage models are used in our performance study.

The *edge table* scheme [10] involves generating an identifier for each node in the original XML document. The edge table scheme stores the nodes in a single table, i.e. the edge table. Each node is represented by a 4-ary tuple: <the node id, the tag, the parent node id, a nullable data attribute>. If the target node is not a data node, the data attribute is null. Traditional indexes are built on some columns. Figure 1 shows an XML document of some soccer game statistics. Figure 2 shows some tuples of the document using the edge table scheme.

The *basic inlining* scheme [18] is an XML shredding scheme which requires a DTD of the XML document to be shredded. A relational schema is generated from the graph. The philosophy is that one would prefer to store (inline) as many nodes as possible in a table. Since an attribute of a table cannot be of collection types, a new table is generated whenever a "*" or "+" sign is encountered in a DTD. For the brevity of presentation, we skip the detail of generating a relational schema from a DTD. We illustrate the basic inlining by an example. Figure 3 shows the DTD for the `soccer.xml` document and Figure 4 shows the generated relation schema.

```
<soccer>
 <group>
  <name>B</name>
  <team>Spain</team><team>Paraguay</team>
  <team>S. Africa</team><team>Slovenia</team>
 </group>
 <stat>
   <team>Spain</team>
   <W>3</W><D>0</D><L>0</L>
   <F>9</F><A>4</A><PTS>9</PTS>
 </stat>
 <stat>
   <team>Paraguay</team>
   <W>1</W><D>1</D><L>1</L>
   <F>6</F><A>6</A><PTS>4</PTS>
 </stat>
 <stat>
   <team>S. Africa</team>
   <W>1</W><D>1</D><L>1</L>
   <F>5</F><A>5</A><PTS>4</PTS>
 </stat>
 <stat>
   <team>Slovenia</team>
   <W>0</W><D>0</D><L>3</L>
   <F>2</F><A>7</A><PTS>0</PTS>
 </stat>
</soccer>
```

Figure 1: The Input Document: soccer.xml

| node id | tag | parent node id | data |
|---|---|---|---|
| 0 | "soccer" | null | null |
| 1 | "group" | 0 | null |
| 2 | "name" | 1 | "B" |
| 3 | "team" | 1 | "Spain" |
| 4 | "team" | 1 | "Paraguay" |
| ... | ... | ... | ... |

Figure 2: Partial Edge Table of the soccer.xml Document

Similar to the basic inlining scheme, the *shared inlining* scheme [18] produces a relational schema from a DTD. The generated schema by using the basic inlining scheme often consists of many relations. The shared inlining scheme proposes that nodes that share the same name are stored in a single table. This often reduces the number of relations of the generated schema. The relation schema generated by the shared inlining scheme is shown in Figure 5.

There are more details on the inlining techniques. For instance, inlining techniques can handle recursions in a DTD, a hybrid inlining scheme has been proposed [18], and extra bookkeeping is used to preserve the order of nodes [20].

As a final remark of this subsection, these storage schemes are known to perform well on certain classes of XML queries and bad on the others. The performance mainly depends on the number of relational join operations required by the XML queries. Recently, an algorithm [3] is developed to generate a relation schema from an XML

```
soccer → group*, stat*
group → name, team*
stat → team, W, D, L, F, A, PTS
other nodes are PCDATA
```

Figure 3: The DTD for the soccer.xml Document

**table** soccer (node id: **integer**)
**table** soccer.group (node id: **integer**, parent id: **integer**, name: **string**)
**table** soccer.stat (node id: **integer**, parent id: **integer**, team: **string**, W: **string**, D: **string**, L: **string**, F: **string**, A: **string**, PTS: **string**)
**table** soccer.group.team (node id: **integer**, parent id: **integer**, value: **string**)

Figure 4: The Relational Schema Generated from the DTD shown in Figure 3 by Using the Basic Inlining Scheme

schema which is optimized for a given workload.

## 2 Column-Based Storage Model for XML Documents

In this section, we review the XMill shredding scheme and propose the XML vectorization. Two important features of the XMill shredding scheme are that there is a very clean separation between the structure and the data and possibly similar data is placed together. These features have been found [17] important for yielding a good compression ratio.

The XMill shredding scheme reads an XML document and produces a *skeleton* of the document, a collection of *data files*, and some dictionary structures. We skip the details on handling attributes and assume the carriage returns and white spaces between tags are not important. The shredding scheme is illustrated by the following example.

The input document is the soccer.xml shown in Figure 1. The output of the shredding is shown in Figure 6, which consists of 1. a tag map, 2. a skeleton of the document, and 3. a collection of data files, which is uniquely identified by the root-to-leaf path.

The document is shredded in a parse. During the parse, a stack is maintained to record the path from the root to the current node. The open tag is replaced with a tag identifier. The data nodes are replaced by a special marker "#". The data nodes are placed into a data file according to its path from the root. The close tags are replaced by another special marker "/". XMill then compresses the skeleton and the data files separately using traditional compressors.

The skeleton is essentially a sequence of integers. Its size is much smaller than the original document size (see Figure 16). Frequent structures in the document tree become frequent sequence of integers in the skeleton. LZ77 compression algorithm is used in this project and it, and also many other compression algorithms, performs well when the input document has a fair amount of frequent sequences. Also note that similar data are placed together. For instance similar statistics is grouped in data columns. One may expect that character data grouped as such can be

**table** soccer.stat (node id: **integer**, parent id: **integer**, W: **string**, D: **string**, L: **string**, F: **string**, A: **string**, PTS: **string**)
**table** team (node id: **integer**, parent id: **integer**, type: **integer**, value: **string**)

Figure 5: The (Partial) Relational Schema Generated from the DTD shown in Figure 3 by Using the Shared Inlining Scheme

```
tag map: soccer = 1, group = 2, name = 3,
         team = 4, stat = 5, W = 6, D = 7,
         L = 8, F = 9, A = 10, PTS = 11
skeleton: 1 2 3 # / 4 # / 4 # / 4 # / 4 # /
          / 5 4 # / 6 # / 7 # / 8 # / 9 # /
          10 # / 11 # / / 5 4 # / 6 # / 7 #
          / 8 # / 9 # / 10 # / 11 # / / 5 4
          # / 6 # / 7 # / 8 # / 9 # / 10 #
          / 11 # / / 5 4 # / 6 # / 7 # / 8
          # / 9 # / 10 # / 11 # / / /

/1/2/3  /1/2/4        /1/5/4     /1/5/6
''B''   ''Spain''     ''Spain''    ''3''
      ''Paraguay''  ''Paraguay''  ''1''
      ''S. Africa'' ''S. Africa'' ''1''
      ''Slovenia''  ''Slovenia''  ''0''

/1/5/7  /1/5/8  /1/5/9 /1/5/10  /1/5/11
''0''   ''0''   ''9''   ''4''    ''9''
''1''   ''1''   ''6''   ''6''    ''4''
''1''   ''1''   ''5''   ''5''    ''4''
''0''   ''3''   ''2''   ''7''    ''0''
```

Figure 6: The Output Produced by XMill Shredding Scheme

compressed effectively.

The *XML vectorization* is a storage model based on this shredding scheme. The data files are the "columns" or the "vectors" of the document. Index structures are added to both of the skeleton and the data columns and compression is applied only to the skeleton of the document.

### 2.1 Indexing the Skeleton of the Document

In general, a query processor for the vectorized XML performs the following three tasks when answering queries: 1. scans the skeleton, 2. finds the node satisfying the query during the scan and 3. fetches the data columns relevant to the query. Thus, scan on the skeleton is a fundamental operation on the vectorized XML. [1]

While scanning a skeleton, the query processor reads all nodes even the node could be irrelevant to a query. There is a need for indexing the skeleton. There are two challenges regarding indexing the skeleton. First, the skeleton is a very

[1] It has been demonstrated that there is a large fragment of XPath queries that can be evaluated in a single pass of the document [11]. We modified the lazy DFA implementation [11] to work on vectorized XML. We notice that the use of the vectorized XML leads to at least a threefold performance gain over flat documents. This is due to the speedup from the parsing time.

compact "encoding" of the document structure. The (both compressed and uncompressed) skeleton is small in size compared to the original document. The additional I/O for some index structures may exceed the size of (or a large chunk of) the skeleton. Second, a sequential scan on the skeleton only requires one explicit disk seek and rotational latency. A tree-based index structure introduces some random disk access.

We propose the algorithm described in Figure 8 (`organizeSkeleton`) to index a skeleton into subskeletons. The idea is to partition the skeleton by a path query $q$ [2], see Figure 7. The skeleton fragments that are relevant to answering $q$ are placed together forming the *skeleton view*. A special marker "*" is placed at the original skeleton when a skeleton fragment is moved to the skeleton view. We call the resultant structure the *complement skeleton view*. The goal is that if for a query $q'$, we find a prefix of $q'$ which is contained in $q$, and the task is *only* to locate the offset of nodes satisfying $q'$, then it is sufficient to scan only the skeleton view. Otherwise it is sufficient to scan only the complement skeleton view. This partitioning can be performed recursively until the gain of such partitioning is smaller than a threshold which represents the increase in random disk access.

The pseudo-code for linking the sub-skeleton is omitted for brevity. The sub-skeletons are stored in a search tree structure: the search key is a path query and data is a subskeleton. The search tree has small number of nodes since we limited the random disk access by the threshold value and the skeleton size is usually small.

Assume a query workload $Q$ is given. The algorithm shown in Figure 8 performs the below tasks. It iterates over the queries in the query workload $Q$ and estimates the cost for answering $Q$ if the skeleton is partitioned by a query. It picks the "best" query $q$ from $Q$ and partitions the skeleton $skel$ by $q$. This step is performed only when the increase in the disk random access is smaller than the gain from a partitioning. This process is repeated until there is no more gain from a partitioning. Hence, the skeleton is partitioned such that the I/O cost for answering $Q$ is the smallest.

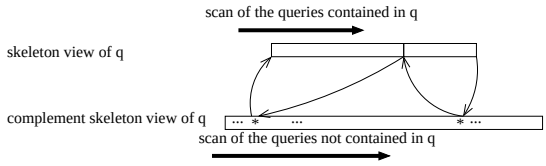


Figure 7: Partitioning the Skeleton by the Query q

We illustrate the `partition` function, which is essential in the algorithm by the following example. Consider the skeleton of the `soccer.xml` document, denote it as $skel$. The `partition(` `/soccer/stat`, $skel$) function returns the skeleton view and the complement skeleton view below.

[2]Path queries refer to the fragment of XPath queries whose containment problem can be decided efficiently by the algorithm presented in [9]

**Input**: $Q$ is a query workload (a set of simple path queries), $skel$ is a skeleton and $\theta$ is a threshold

```
organizeSkeleton(Q, skel, θ) {
  //estimate the cost to answer the query set Q if we partition
  //the skeleton by q
  for each q in Q
    |skel_q^+| = compute the size of skeleton view of q;
    denote α_q = |skel_q^+|/|skel|
    |Q_q^+| = |{q_i | q_i ⊆ q and q_i ∈ Q}|
    if (|Q_q^+| * |skel_q^+| < θ
    || (|Q| - |Q_q^+|) * (|skel| - |skel_q^+|) < θ)
    then cost_q = ∞
    else cost_q = |Q_q^+| * α_q + (|Q| - |Q_q^+|) * (1 - α_q)
  }
  find the q such that cost_q is the smallest
  //divide the skeleton and the query set by q
  Q^+ = {q_i | q_i ⊆ q and q_i ∈ Q} and Q^- = Q - Q^+
  (skel^+, skel^-) = partition(skel, q)
  organizeSkeleton(Q^+, skel^+, θ)
  organizeSkeleton(Q^-, skel^-, θ)
  //code to link the subskeletons
}
```

Figure 8: Algorithm 1. The Algorithm for Indexing a Skeleton

$skel_q^+$ (skeleton view): 5 4 # / 6 # / 7 # / 8 # / 9 # / 10 # / 11 # / / 5 4 # / 6 # / 7 # / 8 # / 9 # / 10 # / 11 # / / 5 4 # / 6 # / 7 # / 8 # / 9 # / 10 # / 11 # / / 5 4 # / 6 # / 7 # / 8 # / 9 # / 10 # / 11 # / /

$skel_q^-$ (complement skeleton view): 1 2 3 # / 4 # / 4 # / 4 # / 4 # / / * /

Suppose the query is `/soccer/group/name`. Since it is not contained in `/soccer/stat`, the query processor fetches only the complement skeleton view from the disk, scans the view and locates all node offsets satisfying the query.

**Example 1.** In this example, we illustrate that the algorithm shown in Figure 8. It picks the "best" query from a query workload and partitions the skeleton by that query. Consider a query workload $Q$ = {`/soccer`, `/soccer/stat`, `/soccer/stat/team`}. The algorithm computes this size of the skeleton view of each query in $Q$. The skeleton view of the three queries in $Q$ is the original skeleton, the skeleton view shown above and 4 # / 4 # / 4 # / 4 # /, respectively. We discuss the iteration regarding `/soccer/stat/team` in detail. $\alpha_q$ is 4 / 39 = 0.1. The number of queries contained in `/soccer/stat/team` in the workload is 0. The cost is 3 * 0.9 + 0 * 0.1 = 2.7. We had obtained 3.0 and 1.82 in the first and the second iteration respectively. Hence, assuming that the threshold is larger than 1.82, to answer the query workloads similar to $Q$, the algorithm partitions the skeleton by `/soccer/stat`.

### 2.2 A Detailed Cost Model for Simple Path Queries

We analyze the total I/O cost, $C$, for evaluating a simple navigation $P$ with a simple predicate with the selectivity $\tau$. We study the detailed I/O costs model [12] in which the cost is the sum of the seek time, the rotational latency and the transfer time.

$$C = N_S \times T_S + N_{I/O} \times T_L + N_X \times T_X$$

The constants used for $T_S$, $T_L$ and $T_X$ approximate those observed for the IBM Deskstar 60GXP hard disk drive [13], as described in Figure 9 and the symbols used in our analysis can be found in Figure 10.

Here we assume that a set of queries $q_1$, ..., $q_n$ are used to partition the skeleton $S$. Partitioning the skeleton by $q_i$ reduces the skeleton size by $\alpha_{q_i}$. Wlog, we assume that the skeleton view is always used. We assume that there is an optimizer picking the smallest skeleton $S'$ for answering the query. Assume that $S'$ is obtained by partitioning $S$ by $q_1$, ... $q_m$. Hence, we have $|S'| = \lceil (\prod_{i=1}^{m} \alpha_{q_i}) |S| \rceil$. Since compression reduces the I/O for the skeleton by a non-trivial amount, the cost model does not account for it. $p$ is the number of columns involved in a query and the columns are stored in the same disk. $Y(k, d, n)$ [22] is the function to estimate the number of blocks fetched to retrieve $k$ tuples out of $n$ tuples stored in $d$ blocks. $I$ is the input buffer size – the number of blocks to be fetched in a read operation.

The amount of disk transfer is the sum of the skeleton and the $p$ vectors.

$$N_X(P,\tau) = |S'| + \sum_{i=1}^{p} IY(\tau||V_i||, |V_i|/I, ||V_i||)$$

To reconstruct a subtree, the disk head moves between the physical locations of the columns. It is fair to assume that each data block fetched costs a disk seek. Vectorization performance deteriorates roughly proportionately to the increase in the number of columns $p$ involved in a query result. The average seek cost can be estimated as one third of the maximum seek cost [19] and we have $\lceil 3\,|S'| / D \rceil$ seek for reading the skeleton.

$$N_S(P,\tau) = \lceil 3|S'|/D \rceil + \sum_{i=1}^{p} Y(\tau||V_i||, |V_i|/I, ||V_i||)$$

The number of disk I/O latency is the sum of the I/O latency for the skeleton and the $p$ vectors.

$$N_{I/O}(P,\tau) = \lceil |S'|/I \rceil + \sum_{i=1}^{p} Y(\tau||V_i||, |V_i|/I, ||V_i||)$$

The skeleton is a very small structure and hence can often be entirely kept in the memory. We assume the following inequality to hold: $M \geq |S'| + pI$

| Symbol | Value | Meaning |
|---|---|---|
| $D$ | 2000000 | Number of blocks in a disk device. |
| $M$ | 131072 (512MB) | Size of main memory in terms of blocks. |
| $I$ | 512 | Number of blocks in an input buffer. |
| $B$ | 4Kbytes | Disk block size, in bytes. |
| $T_S$ | 8.5 | Time for an average disk seek (ms). |
| $T_L$ | 4.27 | Average rotational latency (ms). |
| $T_X$ | 0.6 (6.30MB/sec) | Block transer time (ms). |

Figure 9: Table of System Constants

| Symbol | Meaning |
|---|---|
| $\|S\|$ | Number of blocks in the skeleton. |
| $\|V\|$ | Number of blocks in vector $V$. |
| $\|\|V\|\|$ | Number of entries in vector $V$. |
| $\tau$ | Semijoin selectivity, i.e. the proportion of the entries in a vector that participate in the join. Or the path query selectivity. |
| $N_S$ | Number of seeks in an algorithm. |
| $N_{I/O}$ | Number of I/O requests in an algorithm. |
| $N_X$ | Number of block transfer in an algorithm. |

Figure 10: Table of Symbols

## 3 Query Evaluation on the Vectorized XML

Join is one of the fundamental operations in the relational databases. In this section, we discuss the natural equi-join queries in XML. We also present the cost model of our algorithm for the vectorized XML.

```
<result>
{
  for $x in P_x, $y in P_y
  where $x/Q_x = $y/Q_y
  return $x/R_x, $y/R_y
}
</result>
```

Figure 11: A Join in XML

```
<result> {
  for $x in /soccer/group,
      $y in /soccer/stat
  where
    $x/team = $y/team
  return $x/name, $y
} </result>
```

Figure 12: Example Join Query on soccer.xml

### 3.1 The Join Operation

The general structure of the XML join queries is shown in Figure 11. Many of the queries in the W3C XQuery User Case document section 1.4 – Access to Relational Data [6] exhibit this structure. However, the XML join queries are translated into a number of relational join queries in the systems implemented on top of a relational database. The performance of such systems depends on the number and the size of the intermediate result of the relational joins.

By using the columns (vectors) in XML, we develop the Hybrid-Hash-Jive Join (HHJJ) algorithm for the vectorized XML, which is based on the hybrid hash join and the Jive join algorithm [16]. The algorithm always performs 4 scans on the skeleton, fetches the participating vectors only and 1 join for an XML join query.

The HHJJ algorithm consists 1. the hybrid hash join phrase and 2. the Jive join phrase. 1. We bridge the vectorization and the relational column-based storage model by the following step: associate the node offset to vectors participating the join query. Each vector becomes a binary relation. We apply the hybrid hash join algorithm and build the join index [21] which is necessary for the Jive join algorithm. 2. The relational Jive join algorithm constructs the answer by scanning the participating tables once. We modify the Jive join algorithm so that the XML result subtrees are constructed by two sequential scans on the skeleton. Since the skeleton is often very small and compressed efficiently and the data not needed are not read, the result subtree construction is fast.

### 3.2 Example of the HHJJ Algorithm

In this section we illustrate the details of the HHJJ algorithm by an example. The input document is shown in Figure 1. The query is shown in Figure 12. The pseudo-code of the HHJJ algorithm is shown in the Appendix.

The key steps of the HHJJ algorithm are two. First, it scans the skeleton twice and associates node offsets of $\$x$ ($/soccer/group$) and $\$y$ ($/soccer/stat$) to the data entries found by following the paths $\$x/team$ and $\$y/team$, respectively. After these two scans, we obtain two binary tables shown in the left hand side of Figure 13. The tables are produced as two streams of binary tuples during the scans. The relational hybrid hash join algorithm is applied to two binary relations during the two scans. A binary table of node offsets $\$x$ and $\$y$, shown in the right hand side of Figure 13, is produced. The binary table is then sorted by $\$x$ offsets.

The first step can be performed offline and the corresponding structure of this binary relation in the relational context is named the join index [16].[3] We also refer the binary relation produced in this step as the join index.

| $x | $x/team |
|---|---|
| 3 | Spain |
| 3 | Paraguay |
| 3 | S. Africa |
| 3 | Slovenia |

⋈

| $y | $y/team |
|---|---|
| 19 | Spain |
| 42 | Paraguay |
| 65 | S. Africa |
| 88 | Slovenia |

=

| $x | $y |
|---|---|
| 3 | 19 |
| 3 | 42 |
| 3 | 65 |
| 3 | 88 |

Figure 13: The Two Binary Tables Obtained During the Scans in the Hybrid Hash Join Phrase

The second step is the Jive join phrase. A random access on the skeleton is undesirable since some work is required to resume the root to current node path and some counters. The $\$y$ offsets of the join index are often not sorted and it requires some work to reconstruct the $\$y$ subtrees efficiently. The Jive join phrase consists of two scans on the skeleton to reconstruct the $\$x/name$ and $\$y$ subtrees. Suppose we have two partitions: 1. $\$y < 50$ and 2. $\$y \geq 50$. In practice, the statistics of columns is maintained by offline scans on data. After the first scan of the Jive join phrase, we obtain the partitioned $\$x/name$ subtrees and $\$y$ offsets as shown in the upper part of Figure 14. The subtrees and the offsets are stored in the $Res$ buffer and the $Temp$ buffer respectively. The other scan is required to construct the result $\$y$ subtrees. One pass is sufficient to reconstruct $\$y$ subtrees because all the offsets in $Temp_m$ is greater than those in $Temp_n$ if $m > n$. We scan each $Temp$ partition and, for each partition, we sort the $\$y$ offsets. The scan resumes at where the scan of the last partition stopped. After a scan, we obtain the tables in the lower part of Figure 14. It only requires some straightforward processing to produce the final result.

| $x/name | $y offsets |
|---|---|
| 3 "B" / | 19 |
| 3 "B" / | 42 |

Res$_{x,1}$ Temp$_1$

| $x/name | $y offsets |
|---|---|
| 3 "B" / | 65 |
| 3 "B" / | 88 |

Res$_{x,2}$ Temp$_2$

| $y offsets | $y |
|---|---|
| 19 | 5 4 "Spain" / 6 "3" / 7 "0" / 8 "0" / 9 "9" / 10 "4" / 11 "9" / / |
| 42 | 5 4 "Paraguay" / 6 "1" / 7 "1" / 8 "1" / 9 "6" / 10 "6" / 11 "4" / / |

Temp$_1$ Res$_{y,1}$

| $y offsets | $y |
|---|---|
| 65 | 5 4 "S. Africa" / 6 "1" / 7 "1" / 8 "1" / 9 "5" / 10 "5" / 11 "4" / / |
| 88 | 5 4 "Slovenia" / 6 "0" / 7 "0" / 8 "3" / 9 "2" / 1o "7" / 11 "0" / / |

Temp$_2$ Res$_{y,2}$

Figure 14: The Temporary Result Produced in the Jive Join Phrase.

### 3.3 The Detailed I/O Cost Model for the HHJJ Algorithm

The total cost of the HHJJ algorithm is the seek time, the rotational latency and the transfer time from the hash join phrase, the sorting of the join index and the Jive join phrase.

$$C_{vect} = C_{hash} + C_{sort} + C_{jive}$$

We will discuss $C_{hash}$ and $C_{jive}$ equations, shown below, in details. The cost model for sorting is skipped for brevity. The I/O cost model of the relational hybrid hash join and the Jive join [12, 16] is adapted to the vectorized XML.

$$C_{hash} = N_S^h \times T_S + N_{I/O}^h \times T_L + N_X^h \times T_X$$

$$C_{jive} = N_S^j \times T_S + N_{I/O}^j \times T_L + N_X^j \times T_X$$

**Hybrid Hash Join Phrase.** The hybrid hash join is implemented in the `scan_hash` operator shown in Figure 15. The memory requirement and the optimal values for the input and output buffers are similar to the relational hybrid hash join. We obtained the cost model by including the cost of scanning the skeleton to the formulae.

We assume that the memory size is $M$ and is greater than $\sqrt{f_x |V_x|}$ where $f_x$ is the fudge factor, which includes the increase in size of associating $\$x$ offset to the data entry

[3] Join index is a binary relation containing tuple IDs which involved in the result of a join. The join index is sorted by the first column and is usually stored in a separate disk.

and the overhead from in-memory hash table, and, wlog, $V_x$ is the smaller vector. In the partition phrase, the vectors are read $I_1$ blocks at a time, $O$ is the output buffer size for each partition, $K$ is the number of partitions excluding the ones in the main memory. The size of the join index $J$ obtained in this phrase is denoted as $|J|$. $J$ is written to disk $O_0$ at a time. In the probing phrase, the vector $V_y$ is read $I_2$ at a time. Hence the working space for the in-memory hash tables $WS = M - K \times O - I_1 \geq 0$ and the following inequality holds: $K(M - I_2 - O - O_0) + WS \geq f_x|V_x|$.

Given $I_1$, $I_2$, $O_0$ and $O$, the optimal value for $K$ is follow.

$$K = \lceil (f_x|V_x| - (M - I_1))/(M - I_2 - O - O_0) \rceil$$

The size of the in-memory hash table of $V_x$ is $|V_x^0| = \lfloor WS / f_x \rfloor$

Denote the size of the vector $x$ to be written back to the disk as $|V_x'|$ which equals to $|V_x| - |V_x^0|$.

The size of $|V_y|$ will be reduced roughly proportionately and we have $|V_y'| = \lceil |V_y| \times (1 - |V_x^0| / |V_x|) \rceil$

The number of blocks transfers for hybrid hash join is as follow.

$$N_X^h = |S'| + |V_x| + |V_y| + 2f_x|V_x'| + 2f_y|V_y'| + |J|$$

In the partition phrase, the skeleton and the join vectors are read. Then the reduced vectors are written back. In the probing phrase, a fetch on each partition costs an I/O latency and the reduced vector $V_y$ is fetched back to the main memory. The result is written to the disk once.

The number of seeks is as follow.

$$N_S^h = \lceil 3|V_x|/D \rceil + \lceil 3|V_y|/D \rceil + \lceil 3f_x|V_x'|/D \rceil + \lceil 3f_y|V_y'|/D \rceil +$$

$$\lceil 3|S'|/D \rceil + \lceil f_x|V_x'|/O \rceil + \lceil f_y|V_y'|/O \rceil + 2K + \lceil |J|/O_0 \rceil$$

The number of seek includes those for reading the two vectors, reading the skeleton, writing and reading the vectors to and from the secondary storage, the explicit seek for reading the K partitions and writing the binary offset table back to the disk.

And the number of I/O is as follow.

$$N_{I/O}^h = \lceil |S'|/I \rceil + \lceil |V_x|/I_1 \rceil + \lceil |V_y|/I_1 \rceil + \lceil f_x|V_x'|/O \rceil +$$

$$\lceil f_y|V_y'|/O \rceil + K + \lceil f_y|V_y'|/I_2 \rceil + \lceil |J|/O_0 \rceil$$

The number of I/O includes those for reading the skeleton, reading the vectors, writing the reduced vectors back, fetching the $K$ partitions and the reduced $y$ vector and writing the result to the disk.

The output of the hash join phrase is a binary relation of the $\$x$ and $\$y$ offsets. We assume that good sorting algorithm is used for sorting the output relation by $x$ offsets, which costs $C_{sort}$.

**Jive Join Phrase.** The Jive join is implemented in the `scan_jive` operator shown in Figure 15. The difference between the Jive join phrase and the relational Jive join is that the former performs on two sequential scans the skeleton and one sequential scans on relevant data vectors for result reconstruction and the latter performs a sequential scan on the two relations participated in the join. We obtain the cost model for `scan_jive` by the cost model for evaluating simple paths over vectorized XML (see section 2.2) and that of the relational Jive join algorithm [16].

We assume that the binary relation from the hash join phrase is divided into $y$ partitions. There are *output file buffer* of the size $O_1$ blocks and *temporary file buffer* of the size $O_2$ blocks associated to a partition. Therefore, the following equality must hold: $y(O_1 + O_2) \leq M$.

Let $|Res_x|$ be the size of the $\$x/R_x$ subtrees involved in the answer.

$$N_S^j = \lceil 3|J|/D \rceil + N_S(P_xR_x, \tau_x) + N_S(P_yR_y, \tau_y) +$$

$$\lceil |J|/(2O_1) \rceil + \lceil |Res_x|/O_2 \rceil$$

The total number of disk seek is to scan the result from the hash join phrase, to reconstruct $|Res_x|$ and $|Res_y|$, to write the $x$ offsets and the $|Res_x|$ to the disk.

$$N_{I/O}^j = 2y + \lceil |J|/(2O_1) \rceil + \lceil |Res_x|/O_2 \rceil +$$

$$N_{I/O}(P_xR_x, \tau_x) + N_{I/O}(P_yR_y, \tau_y)$$

The total number of I/O requests is $2y$ for partition switch, $|J|/(2O_1)$ for flushing the temporary files, $|Res_x|/O_2$ for $\$x/R_x$ subtrees, and the sum of the I/O requests for reconstructing the $\$x/R_x$ subtrees and the $\$y/R_y$ subtrees.

$$N_X^j = |J| + N_X(P_xR_x, \tau_x) + N_X(P_yR_y, \tau_y) + |Res_x|$$

The total block transfer is $|J|/2$ for writing the temporary buffer files, $|J|/2$ for reading the temporary buffer files, and the I/O for reading the result and writing the $P_xR_x$ result back to disk.

We now determine the optimal value for $y$, $O_1$ and $O_2$. $c(y, O_1, O_2) \approx (|J|/(2O_1) + |Res_x|/O_2)T_S + (2y + |J|/(2O_1) + |Res_x|/O_2)T_{I/O}$. With the inequality above, we obtained the optimal value for $y$, $O_1$ and $O_2$ as follow.

$$O_1 = M^2/(L(1 + \sqrt{2|Res_x|/|J|}))$$

$$O_2 = M^2/(L(1 + \sqrt{|J|/2|Res_x|}))$$

$$y = L/M where L = |J|/2 + \tau_y|Res_y|$$

## 4 Physical Algebra on the Vectorized XML

From the implementation perspective, the XML vectorization consists of some basic operators (a set of algebra). Since the vectorized XML is often accessed sequentially, the operators are scan operators and are implemented as iterators. We show the set of physical algebra (Figure 15) presented in earlier sections.

The `scan` operator performs a few tasks. Given the input path query $P$, it transverses the index by checking if

| Name | Input | Output | Cost | Meaning |
|---|---|---|---|---|
| scan | a path expression $P$ | a list of node offsets | $C$ | locate the nodes by following $P$ (projection) |
| scan_hash | path expressions $P_x, Q_x, P_y, Q_y$ | a binary table of node offsets | $C_{hash}$ | the path expressions refer to the path expression in Figure 11. It implements the hybrid hash join algorithm. |
| scan_jive | a binary relation of node offsets $J$, path expressions $R_x, R_y$ | for the first (second) node offset in a tuple in $J$, follow $R_x$ ($R_y$) and reconstruct the subtree | $C_{jive}$ | the path expressions refer to the path expression in Figure 11. It implements the jive join algorithm. |
| fetch | column ID $i$ | the next entry in the column | the amortized cost is $\|V_i\|$ | Iterates over the column entries |

Figure 15: The Physical Algebra for the Vectorized XML Presented in this Work

the query of a node $P'$ such that some prefix of $P \subseteq P'$. After the search, it fetches the corresponding compressed subskeleton and decompresses it on-the-fly. An automaton is constructed for the path $P$. The skeleton is passed to the automaton and the offset of the nodes satisfying $P$ is reported.

The `scan_hash` operator and the `scan_jive` operator implement the hybrid hash join phrase and the Jive join phrase respectively. Since the join index is a relatively small structure, the join index of common join queries are precomputed offline by `scan_hash`. The join operator implements the follow basic steps. It checks if a join index exists, executes the coresponding `scan_hash` operator if necessary and then executes the `scan_jive` operator.

The `fetch` operator is an iterator for a data column in the absence of indexes. An index is built on a data column by the following step. 1. A binary table of the data entry and the corresponding node offset is constructed. 2. A B+-tree is constructed on this table.

# 5 Experimental Evaluation and Cost Model Validation

Based on the cost model, one can design use cases in which the column-based storage out-performs. The experiments are designed for understanding the performance of the XML vectorization so that use cases on the border-lines are picked. We also validate the cost model by experiments.

## 5.1 Test Environment

Our experiments were performed on a 1.5GHz, 512M RAM, 6.30M/sec hard disk with a 2MB buffer, computer running Linux OS. Since we implement the system on top of a software layer, the operating system, the disk is *not* at its optimal speed. The page size is 4096 bytes. To ensure disk access, we flush the main memory before each experiment unless otherwise specified. Since the hard disk offers a 2MB buffer, the input and output buffer sizes are set to be at least 2MB. We use a commercial relational database which implements N-ary storage model. Almost all the main memory is assigned to the database buffer. Most documents and query sets used in this experiment are intended to be simple. This helped to reduce the complex interaction between the hardware, the operating system and the database. The buffer management offered by the OS and the hard disk speed up the disk writes by a non-trivial amount. We do not know of an effective way to control these effects and hence write time is not included in our validation of the cost models. And we do not measure the time for CPU cost and memory stalls in the validation. No index is built on the skeleton unless otherwise specified.

## 5.2 Compression

This experiment validates that the skeleton can be compressed effectively and is a compact representation of the structure of the original document. We use a few real XML documents, shown in the first row of Figure 16. Shakespeare is the concatenation of all Shakespeare plays [5], DBLP document [15] is the XML document of the computer science bibliography. Swissport [14] contains experiment data on proteins. We generate two documents which both conform to the DTD below.

$Root \rightarrow A*$ $\quad A \rightarrow (B, C, D, E, F)$

$B, C, D, E, F$ nodes are $PCDATA$.

The first and second synthetic document contains 1,000,000 and 4,000,000 $B$-$C$-$D$-$E$-$F$ tuples respectively.

Figure 16 shows that the size of the skeleton is much smaller than the original document size and the skeleton can usually be compressed efficiently.

The time for fetching the skeleton and the time for fetching and decompressing the compressed skeleton are measured. The decompression overhead pays off when the skeleton is large as shown in the fifth row of table 16. We find that the decompression time is related to the content of the compressed skeleton.

## 5.3 Experiment on Simple Path Queries

### 5.3.1 Column-Based versus Relational-Based Storage Model

This experiment illustrates roughly two boundaries between the column-based storage model and the relational-based storage model by testing their performance on some simple queries. 1. We demonstrate that the performance of a relational-based XML storage scheme deteriorates when the tuple width is large. 2. Similarly, we demonstrate that the performance of a column-based XML storage scheme deteriorates when the queries involve a few of columns.

| **Documents** | Shakespeare | DBLP | Swissport | Synthetic | Synthetic 2 |
|---|---|---|---|---|---|
| **Document Size** | 7.6M | 103M | 457.4M | 60.4M | 241.8M |
| **Skeleton Size (I/O time)** | 539K (0.02s) | 7.8M (0.35s) | 32M (1.03s) | 18M (0.57s) | 72M (2.22s) |
| **Compressed Size (decompression time)** | 31K (0.04s) | 240K (0.44s) | 7.9M (0.21s) | 45K (0.29s) | 178K (0.88s) |
| **Compression Ratio** | 17.4 | 32.6 | 4.05 | 400 | 404 |

Figure 16: Relationship of the Skeleton Compression and I/O Time

A document `c10.xml` conforming to the DTD below is generated:
$Root \rightarrow (C_1, C_2, ... C_{10})$*
$C_1, ..., C_{10}$ nodes are $PCDATA$.

`c10.xml` contains 100,000 "tuples" and the character data in `c10.xml` are random integers. The query set used is shown in Figure 17. We measure the time to answer these queries but ignore the time for printing.

We enumerate a similar experiment using a relational database. We create a $R$ table which contains $C_1$ - $C_{10}$ attributes. All attributes are 4 bytes and contain random integers as in the previous experiment. We alter the table width by setting the size of all attributes to be 8 bytes, 16 bytes, respectively.[4] We use the query: `select sum(`$C_1$`), sum(`$C_2$`), ... from` $R$ to enumerate the query set shown in Figure 17. The result is shown in Figure 18.

| | |
|---|---|
| Q1 | $/R/C_1$ |
| Q2 | $/R/(C_1 \mid C_2)$ |
| Q3 | $/R/(C_1 \mid C_2 \mid C_3)$ |
| Q4 | $/R/(C_1 \mid C_2 \mid C_3 \mid C_4)$ |
| Q5 | $/R/(C_1 \mid C_2 \mid C_3 \mid C_4 \mid C_5)$ |

Figure 17: Query Set for the Projection Experiment

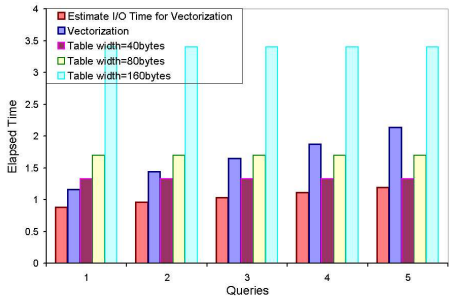


Figure 18: The Elapsed Time and the Columns Involved in a Query on the Vectorization and Tables with Different Width

The relational database spends the same time to answer $Q1$-5 since the database system implements the N-ary storage model, i.e. all tuples are fetched once. The horizontal lines in Figure 18 indicate that the performance of the relational database deteriorates as the table width increases.

[4]In general, a relational-based shredding of XML document may yield numerous tables which can contain numerous attributes. The experiment assumes that it is not valid to build indexes on all attributes of all tables.

The performance of the vectorization deteriorates roughly proportionately to the increase in the number of columns involved in a query. Consider $Q1$ ($Q5$), it is required to fetch the skeleton and 1 (5) vector(s), another 9 (5) vectors are not needed and are not read.

This validates that when most queries require a small portion of the data, a column-based storage model is preferred while when the width of data is small, a relational-based storage model is preferred.

### 5.3.2 Queries on Document Structure

We measure the elapsed time of evaluating some simple path queries on the vectorization, the edge table scheme, the basic inlining scheme and shared inlining scheme. We only consider the elapsed time to find the node offsets (or ids for the relational database implementation) by following the query path. This is because the results of some queries are trees and implementation based on a relational database requires tremendous overhead on the subtree reconstruction. For the vectorization, we do not fetch any vectors since they are not required for locating the result nodes. There is no index built on the skeleton in this experiment. We generated documents conforming to the following two DTDs.

$R \rightarrow (A_1, A_2)^*$ $A_x \rightarrow (B_1, B_2)^*$
$B_x \rightarrow (C_1, C_2)^*$ $C_x \rightarrow (D_1, D_2)^*$
$D_x \rightarrow (E_1, E_2)^*$ $E_x \rightarrow PCDATA$
where $x$ in $\{1, 2\}$.

The root node of the document tree is $R$. Each $R$ node has 8 $A_1$ nodes and 8 $A_2$ nodes. Each $A_x$ node has 8 $B_1$ nodes and 8 $B_2$ nodes and so on. We obtained a document with 1M nodes. The second DTD is shown below.

$R \rightarrow A^*$ $A \rightarrow B^*$ $B \rightarrow C^*$
$C \rightarrow D^*$ $D \rightarrow E^*$ $E \rightarrow PCDATA$

$R$ is the root node of the document and it contains 1M $A$ nodes. Each $A$, $B$, $C$, $D$ contains 1 $B$, $C$, $D$ node, respectively. The character data of the two documents is filled with random integers. The query sets on the first and the second document are shown in the LHS and the RHS of Figure 19.

| | | | |
|---|---|---|---|
| Q1 | /R | | |
| Q2 | /R/A1 | Q2' | /R/A |
| Q3 | /R/A1/B1 | Q3' | /R/A/B |
| Q4 | /R/A1/B1/C1 | Q4' | /R/A/B/C |
| Q5 | /R/A1/B1/C1/D1 | Q5' | /R/A/B/C/D |
| Q6 | /R/A1/B1/C1/D1/E1 | Q6' | /R/A/B/C/D/E |

Figure 19: Simple Path Query Set

Extensive indexes are built on the edge map and inlining schemes. We also assume that an XML query optimizer translating a path query to SQL queries without incurring overheads. The result is shown in Figure 20.

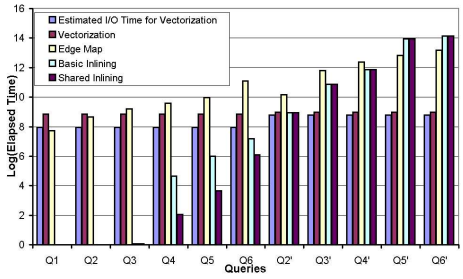


Figure 20: Simple Path Query Experiment on the Vectorized XML and Other Storage Schemes

The edge map and inlining techniques perform the best for simple queries since it requires joins on relatively small tables. However, the query processing time of these storage models increases as the number of relational joins required by the query increases. The query processing time of the XML vectorization is a constant but not as good for small documents. It indicates that a scan is relatively expensive for simple queries. The vectorization becomes competitive when the path queries are complex or when the queries require joins on relatively large tables (see Q2' - Q6').

A scan on the skeleton reproduces all the information of the structure of the document, e.g. a single scan is good enough for reconstructing the entire document. We measure the time for reconstructing the entire document ("/" in XPath syntax) and the result is shown in Figure 21. The relational-based storage requires numerous relational joins and the order of node is often not preserved. [5] Hence the vectorization outperforms other storage schemes when the query requires all information on the structure and the relatively expensive scan pays off.

| **Storage Scheme** | **Elapsed Time** |
|---|---|
| Vectorization | 47s (Estimate I/O time: 9.8s) |
| Edge Map * | 546s |
| Shared Inlining * | 69s |
| Basic Inlining * | 182s |

Figure 21: Time for Reconstructing the Document (* means that our implementation of such scheme does not preserve the order of nodes.)

Character data is accessed in document order in the document reconstruction. Since data is organized in a different order in the XML vectorization, each fetch operation (see Figure 15) induces a memory stall.[6] The memory activity is not included in our model and hence there is a notable difference between the elapsed time and the estimated I/O time.

Finally, we issue the query set in the presence of an index structure. We consider only the time for locating the relevant node offsets. The goal is to optimize the processing time of query set shown in Figure 19. The algorithm in Figure 8 first partitions the skeleton by Q1, which divided the skeleton roughly into two subskeletons with the same size. Then the subskeleton is partitioned by Q2, Q3, Q4 and Q5. The subskeleton is divided into two subskeletons of roughly the same size in each partitioning. Figure 22 shows that the speedup is roughly proportionate to the reduce of the size of the skeleton. The speedup is smaller than the estimated one when the size of the skeleton becomes comparable to the size of the query processor itself.

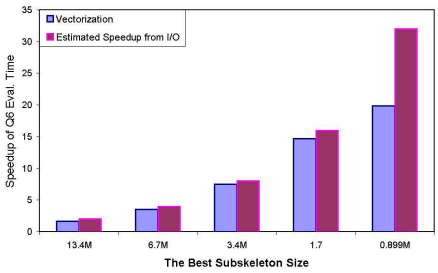


Figure 22: Speedup of Q6 by Recursively Partitioning the Skeleton

### 5.4 Experiment on XML Join Queries

This experiment validates the detailed cost model presented in section 3.3. We also present the result of a performance comparison on join queries with edge map storage scheme on synthetic data sets.

#### 5.4.1 Validating the I/O Cost of the Hybrid Hash Phrase and the Jive Join Phrase

To validate the cost model, we compare the elapsed I/O time and the estimated I/O time. We generate two documents which both conform to the following DTD.
$A \rightarrow B*$ $\quad B \rightarrow C\ D\ E$
$C$, $D$, $E$ nodes are $PCDATA$.

The first and second synthetic documents have 1,000,000 and 2,000,000 $B$ tuples respectively. Character data is filled with integers such that the join query shown in Figure 23 becomes a full participation of one-to-one join query.

```
<result>{
  for $x in /A/B, $y in /A/B
  where $x/C = $y/D
  return $x/D, $y/E
} </result>
```

Figure 23: The Join Query Used in the Cost Model Validation

[5] Recently, [20] proposed an order preserving inlining scheme.

[6] A simple solution here is to keep a copy of data in document order. We implemented this solution and notice a twofold increase in performance. In such implementation, the I/O time is the dominant factor of the performance.

The skeleton size $|S|$ of the first document (the second document) is 3000 blocks (6000 blocks). The vector size $|V_x|$ = $|V_y|$ = 1720 blocks (3750 blocks). $||V_x||$ = $||V_y||$ = 1,000,000. The buffers $I$, $I_1$, $I_2$ are set to be 500 blocks. (The size of output buffers is omitted because the disk write time is measured.) The fudge factor $f_x$ ($f_y$) is 1.8 (1.9). To control the memory usage, we varied the value of the working space $WS$.

Figure 24 and 25 show the real I/O time of the hash join phrase and the corresponding estimated I/O time. The x-axis is the amount of memory used to keep the hash tables. On one side, there is no in-memory hash table and on the other side, all partitions are kept in the main memory. Both figures indicates that the I/O time for the hash join phrase is estimated well. The accuracy of the estimation decreases as the number of partitions on the disk increases. This is because the seek time and the I/O latency for a Linux formatted disk are larger than the values appeared on the hard disk specification [13].

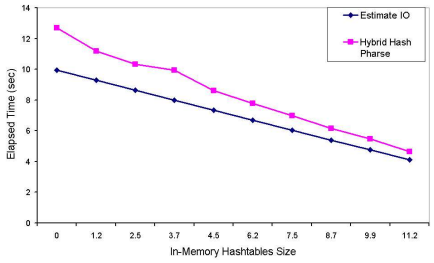


Figure 24: The I/O Time of the Hybrid Hash Join Phrase verse the Memory Allowed by the In-Memory Hashtable Size. (Doc.: 1st Synthetic Doc. Query: See Figure 23)

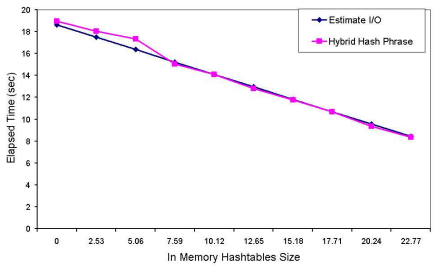


Figure 25: The I/O Time of the Hybrid Hash Join Phrase verse In-Memory Hashtable Size. (Doc.: 2nd Synthetic Doc. Query: See Figure 23)

Similarly we compare the estimated and the elapsed I/O time of our jive join implementation. The join index size $|J|$ of the join query with full participation is 2000 (4000). We vary the selectivity $\tau$ of the $\$x$ nodes. The size of the participating vector is 1720 blocks. Figure 26 shows that the cost model accurately predicts the elapsed time.

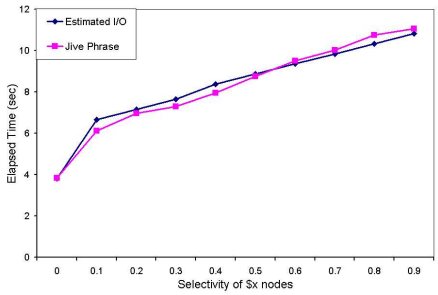


Figure 26: The I/O Time of the Jive Join Phrase verse the Selectivity of the Query Shown in Figure 23.

### 5.4.2 Performance Comparisons with the Edge Map Scheme

The final experiment compares the join queries on the vectorization and the edge map storage scheme. The synthetic document 1 and 2 conforms to the DTD shown in section 5.4.1 with 430K and 210K $B$ tuples. Index structures have been built on the edge map table.

The experiment result is shown in Figure 27. Indeed, the synthetic documents are designed such that the intermediate join result is large. This experiment shows that a small number of joins with many participating tuples will be more expensive than a few scans on the skeleton of the document plus a join on data columns.

| Doc. | Query | Storage | Elapsed Time |
|---|---|---|---|
| Syn. Doc. 1 | Twig Query 1 (4 scans + 1 join) | Vect. | 9.472s |
| Syn. Doc. 1 | Twig Query 1 (2 joins) | Edge Map | 201.8s |
| Syn. Doc. 2 | Twig Query 2 (4 scans + 1 join) | Vect. | 4.45s |
| Syn. Doc. 2 | Twig Query 2 (2 joins) | Edge Map | 85.6s |

Figure 27: Performance Evaluation on Synthetic Data Sets

## 6 Discussion

**Reuse of Relational Databases Technology.** Vectorization is implemented from scratch so that we have better understanding on the I/O of our join algorithms. However, this storage scheme is not against the reuse of relation databases. Vectorized XML can be maintained by a relational database.

**Update.** Vectorization inherits the update inefficiency from the column-based storage model. Inserting or deleting an N-ary tuple requires disk access on $N$ columns and the column headers. Similarly, such an update on the vectorized XML touches a few columns and the column headers and a scan on the skeleton.

**Cache Conscious Data Placement.** Initial experiment shows that the query evaluation time exceeds the estimated I/O time by a noticeable amount. Our implementation experience shows us that this is mostly due to the memory

stalls of the instruction and data cache. Since the skeleton and a vector are often small, some in memory structures can be built to speedup the memory performance.

## 7 Conclusions

In this paper, we consider the shredding scheme used in XMill, an XML compressor, as an XML storage model – the XML vectorization. The XML vectorization considers both the compression and the query processing of XML documents. The skeleton of the document is indexed and compressed. The character data is stored in columns and traditional indexes are built on them. We study the detailed cost model of querying the vectorized XML. We have showed that

- XML vectorization performs well on queries which require much information on the document structure. We have experimentally verified some good and bad query sets for the vectorization and some other shredding schemes.
- XML vectorization naturally supports XML join queries by using the modified hybrid hash join and the Jive join algorithm. Almost all of the I/O performed is sequential and data not needed by a query is not fetched. The algorithm requires 4 scans and 1 join on participating columns for any XML join queries;
- an index structure and a physical algebra are consistent to this nature.

**Acknowledgments.** The authors would like to thank Val Tannen and Derick Wood for insightful discussions. The first author is supported under a Research Grants Council Earmarked Research Grant. Some work of the first author was done while he is visiting the Hong Kong University of Science and Technology.

## References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB Conference*, 2001.

[2] D. S. Batory. On searching transposed files. *TODS*, 4(4):531–544, 1979.

[3] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *ICDE*, 2002.

[4] P. Boncz, A. N. Wilschut, and M. L. Kersten. Flattening an Object Algebra to Provide Performance. In *ICDE*, 1998.

[5] J. Bosak. The Plays of Shakespeare in XML. Available at `http://www.oasis-open.org/cover/bosakShakespeare200.html`., July 1999.

[6] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. Xml query use cases. Available at `http://www.w3.org/TR/xmlquery-use-cases/`, 2002.

[7] Y. Chen, S. Davidson, and Y. Zheng. 3XNF: Redundancy Eliminating XML Storage in Relations. Technical Report, 2003.

[8] G. Copeland and S. Khosafian. A Decomposition Storage Model. In *SIGMOD Conference*, 1985.

[9] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *KRDB Conference*, 2001.

[10] D. Florescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report, 1999.

[11] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *ICDT*, 2003.

[12] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the Truth About ad hoc Join Costs. *VLDB Journal*, 6(3):241–256, 1997.

[13] IBM Corporation. IBM Deskstar 60GXP Hard Disk Drives. Available at `http://www.storage.ibm.com/hdd/prod/ds60gxpdata.htm`.

[14] E. B. Institute. SWISS-PROT. Available at `http://www.ebi.ac.uk/swissprot/Documents/documents.html`, 2002.

[15] M. Ley. DBLP Bibliography. Available at `http://www.informatik.uni-trier.de/~ley/db/`, May 2003.

[16] Z. Li and K. A. Ross. Fast Joins Using Join Indices. *VLDB Journal*, 8(1):1–24, 1999.

[17] H. Liefke and D. Suciu. XMill: an Efficient Compressor for XML Data. In *SIGMOD Conference*, Jun 2000.

[18] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J.Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB Conference*, 1999.

[19] E. Shriver. Performance Modeling for Realistic Storage Devices, 1997.

[20] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD Conference*, 2002.

[21] P. Valduriez. Join Indices. *TODS*, 12(2):218–246, 1987.

[22] S. B. Yao. Approximating the Number of Accesses in Database Organizations. In *Communications of the ACM*, 1977.

# A Appendix – The Pseudo-Code of the HHJJ Algorithm

**Input**: $P_x$, $P_y$, $Q_x$, $Q_y$ refers to the paths shown in figure 11, and $skel$ is a skeleton
**Output**: A binary relation of offsets
scan_hash($P_x$, $P_y$ $Q_x$, $Q_y$, $skel$) {
//This part corresponds closely to the hash join in
//relational setting
//Partition the two vectors into $K$ partitions
  Scan the skeleton once. During the scan,
    find the node by following $P_x$ and
    denote the offset of this node as $o_x$
    find the data entries by following $Q_x$
    and denotes it as $V_x$
    where $V_x = \{d_{x_1}, d_{x_2}, ... d_{x_m}\}$
    construct $V_{o_x} = \{(o_x, d_{x_i}) \mid d_{x_i} \in D_x\}$
    partition the $V_{o_x}$ by the hash function $h$
  Scan the skeleton once. During the scan,
    find the node by following $P_y$ and
    denote the offset of this node as $o_y$
    find the data entries by following $Q_y$
    and denotes it as $V_y$
    where $V_y = d_{y_1}, d_{y_2}, ... d_{y_n}$
    partition the $V_{o_y}$ by the hash function $h$
//Probing Phase
  `for` $l$ = 1, ..., $k$ `do` {
  `foreach` offset-data pair the partition $\in x$
    read an offset-data pair and insert into
    hash table using $h2$
  `foreach` offset-data pair the partition $\in y$
    read an offset-data pair $s$ and probe the table
    using $h2(s)$
    if it matches, put $<o_x,o_y>$ into $Res$
  clear hash table to prepare for next partition
  }
  `return` $Res$;
}

Figure 28: Algorithm 2. The Hash Join Phrase for the HHJJ Algorithm for the Vectorized XML. (Based on this algorithm, we yield the hybrid version of the scan_hash procedure.)

**Input**: $JoinIndex$ is a binary relation of offsets and it is sorted by the offsets in the first column, $R_x$, $R_y$ refers to the paths shown in Figure 11, and $skel$ is a skeleton
scan_jive($JoinIndex$, $R_x$, $R_y$ $skel$) {
//Getting \$$x$/$R_x$
  Scan the skeleton and in parellel, perform the below
  `foreach` $<o_x, o_y>$ in $JoinIndex$
    $t_{o_x}$ = `reconstructSubtree`($skel$, $o_x$, $R_x$)
    $pid$ = `getPartitionID`($o_y$)
    add $t_{o_x}$ to $Res_{x,pid}$
    add $o_y$ to $Temp_{pid}$
`flush_memory`()
//Getting \$$y$/$R_y$
  Scan the skeleton and in parellel, perform the below
  `foreach` $pid \in AllPartitionID$
    read the entire $Temp_{pid}$
    in-memory sort the $y$ offsets in $Temp_{pid}$
    `foreach` $o_y$ in $Temp_{pid}$
      $t_{o_y}$ = `reconstructSubtree`($skel$, $o_y$, $R_y$)
      add $t_{o_y}$ to $Res_{y,pid}$
    look up the original order of $o_y$ in $Temp_{pid}$
    write the $t_{o_y}$ accordingly.

  combine the result ($Res_{x,pid}$, $Res_{y,pid}$)
}

Figure 29: Algorithm 3. The Jive Join Phrase of the HHJJ Algorithm for the Vectorized XML

**Input**: $P_x$, $P_y$, $Q_x$, $Q_y$, $R_x$, $R_y$ refers to the paths shown in Figure 11, and $skel$ is a skeleton
`HHJJ`($P_x$, $P_y$ $Q_x$, $Q_y$, $R_x$, $R_y$ $skel$) {
//Compute the pair of offsets of $<x, y>$
//participate in the join
  $OffsetPair$ = scan_hash($P_x$, $P_y$ $Q_x$, $Q_y$, $skel$);
  $JoinIndex$ = sort $OffsetPair$ by the first column;
//Construction of Result
  scan_jive($JoinIndex$, $R_x$, $R_y$ $skel$);
}

Figure 30: Algorithm 4. The HHJJ Algorithm for the Vectorized XML