Partition Based Path Join Algorithms for XML Data *

Quanzhong Li and Bongki Moon

Department of Computer Science, University of Arizona, Tucson, AZ 85721.
{lqz,bkmoon}@cs.arizona.edu

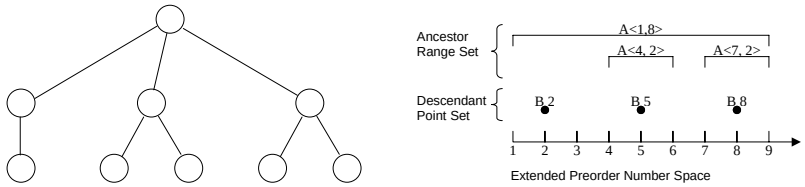
Abstract. Path expression is an important component in querying XML data. The extended preorder numbering scheme enables us to quickly determine the ancestor-descendant relationship between elements in the hierarchy of XML data. Using the numbering scheme, a path expression can be evaluated by join operations to avoid potentially high cost of tree traversals. In this paper, we first formulate XML path queries as range-point join queries. Then we discuss the partition based algorithms that can utilize the *range containment property* to efficiently process the range-point join queries. Under the partition based framework, we propose three algorithms, namely *Descendant partition join*, *Segment-tree partition join* and *Ancestor Link partition join*, which can be chosen by a query optimizer for different input data characteristics. The experimental results show that the partition based algorithms can make better use of the buffer memory than sort-merge algorithms, and the proposed *Ancestor Link join* algorithm yields the best performance by using small in-memory data structures and by taking advantage of unevenly sized inputs.

1 Introduction

With the popularity of XML as a new standard for information representation and exchange on the Internet, the problem of managing and querying XML data is becoming more and more important. Various work has been done to efficiently evaluate queries on XML data. In graph-based data models, a path expression is evaluated through tree traversal according to the shape of the data [8, 3]. With the introduction of the numbering scheme on XML data [6], a path expression can be processed using join algorithms [6, 13, 12]. *Path Join* algorithms [6] (e.g. $\mathcal{EE}$**-Join** and $\mathcal{EA}$**-Join**) are sort-merge based algorithms to process ancestor-descendant type expressions. *Structural Joins* [12] (tree-merge and stack-tree algorithms) optimizes the join performance by introducing in-memory stacks.

Sort-merge based algorithms assume the inputs are in sorted order of assigned numbers, but this order is not always guaranteed. For example, an input may be sorted by data values, or it may be the result from operations using hash indexes. In this paper, we discuss the use of partition based algorithms to process XML join queries. When the descendant input is used as the outer set in the join, the *Descendant Partition Join* algorithm can be used to process join operations. When the ancestor input is used as the outer set, we propose the *Segment-tree Partition Join* and the *Ancestor Link Partition Join* algorithms. The experimental results show that the Ancestor Link algorithm

* This work was sponsored in part by the National Science Foundation CAREER Award (IIS-9876037), NSF Grant No. IIS-0100436, and NSF Research Infrastructure program EIA-0080123. The authors assume all responsibility for the contents of the paper.



(a) Numbering Scheme Example (b) Range-Point Sets Illustration

Fig. 1. Numbering Scheme and Range-Point Sets

can make the best use of memory buffer and take advantage of unevenly sized inputs. We believe that these algorithms are necessary choices for query optimizers to consider during XML query processing for different input characteristics.

The rest of the paper is organized as follows. In Section 2, we will introduce the numbering scheme and three partition based join algorithms. Section 3 presents the performance results. After the brief survey in Section 4, we conclude the work of this paper in Section 5.

2 Partition Based Path Join Algorithms

Since the extended preorder numbering scheme is the basis for path join algorithms, we will briefly introduce it next.

2.1 Extended Preorder Numbering Scheme

XML data objects are commonly modeled by a tree structure, where nodes represent elements, attributes and text data, and parent-child node pairs represent nesting between XML data components. The *Extended Preorder Numbering Scheme* captures this tree structure and assigns each node a pair of numbers, $<order, size>$. The *order* is similar to the preorder, and the *size* is the number of descendants. This pair of numbers should satisfy the following conditions:

- For a tree node y and its parent x, $order(x) < order(y)$ and $order(y) + size(y) \leq order(x) + size(x)$. In other words, y's range [$order(y)$, $order(y) + size(y)$] is contained in x's range [$order(x)$, $order(x) + size(x)$].
- For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, $order(x) + size(x) < order(y)$. In other words, x's and y's ranges are disjoint.

For example, Figure 1(a) shows a tree labeled with these pairs of numbers. The ancestor-descendant relationship can be determined in constant time by examining these numbers. That is, for two given nodes x and y of a tree T, x is an ancestor of y if and only if $order(x) < order(y) \leq order(x) + size(x)$. To process path expressions, we can gather two node sets, the ancestor set and the descendant set, and join them together using the above condition.

According to the numbering scheme, we can treat the ancestor set as a range set and the descendant set as a point set. In Figure 1(a), suppose we are going to find all the paths of the pattern "A//B", which is to get all the "B" descendants of "A". We can first gather the ancestor set, which is $\{< 1, 8 >, < 4, 2 >, < 7, 2 >\}$. Then, we obtain the descendant point set, which is $\{2, 5, 8\}$. By the numbering scheme, the ancestor set corresponds to the range set: $\{[1, 9], [4, 6], [7, 9]\}$. Now, the task to find "A//B" is the

same as to find the pairs of range and point, where the point is contained in the range. Figure 1(b) illustrates this range set and point set relationship. After mapping ancestor sets to range sets and mapping descendant sets to point sets, the problem of finding path pattern is reduced to computing the join between a range set and a point set, which will be referred as *range-point join* in this paper.

The ancestor range set is generated from a document tree using the numbering scheme, where each range corresponds to a sub-tree in the document tree. Since there is no partial overlap between any two sub-trees, there is no partial overlap among ranges. We formalize this property as *range containment property*, which is defined as follows:

Range Containment Property: For any two ranges in the range set, either one range is contained in the other or they are disjoint.

It is this property that provides us the opportunity and basis to deal with this special type of join operation. In the following section, we will describe how this property can be exploited to make the join algorithms more efficient.

2.2 Partition Based Algorithms

Data Partitioning The first step in partition join is to partition both input data sets that need to be joined. Each pair of corresponding partitions from both sets need to be joined. In order to avoid data reread, at least one partition of the pair should fit in memory. If no information about the data distribution is available, sampling data can help to determine the partition boundaries. There has been some research work addressing the random sampling problem [1, 7]. We have chosen a similar algorithm to "determinePartIntervals" algorithm [11] to determine the partition intervals. This partition algorithm considers partition cost, sampling cost and join cost to minimize the total I/O. The cost of sampling is computed based on the Kolmogorov test statistic [2]. During sampling, we clustered disk page reads such that sample pages close enough are read sequentially instead of several random reads. If one partition is larger than memory size, further partitioning can be applied recursively. In this paper, we assume that each outer partition can fit in memory, which is also true in our experiments.

Range Partition and Range Cache Unlike the point set, for a range set, there is a possibility that some ranges can overlap with multiple partitions, no matter how the partition boundaries are determined. Which partition should we put those long ranges in? A straightforward solution is to replicate the ranges to all partitions it overlaps. This requires additional disk I/O to handle replicates. Instead, we adopted the solution proposed in [11], where the partitioning is performed using only the start point of each range. A range is put in the partition, where the start point of the range falls in. One requirement of this method is that the partition join should be evaluated in increasing order of partitions. When the current partition is done, the ranges that cross the next partition boundary are kept in memory cache. These ranges in cache will participate in the join of the next partition.

One nice property of the XML data range set is that the number of ranges crossing any partition boundary is bounded by the height of the document tree. Thus, we can pre-allocate the in-memory cache to hold those crossing boundary ranges according to the tree height. A possible improvement is to use the maximum level difference of the ranges plus one as the cache upper bound. This upper bound can be obtained before

Algorithm 1: Descendant Partition Join

```
   Input: (ancestor range set, descendant point set)
 1 Set range cache to be empty;
 2 Determine partition boundaries according to the descendant point set;
 3 Partition both range and point sets;
 4 for Each partition pair in increasing order do
 5     Load descendant partition in memory;
 6     for Each range in range cache do
 7         Join the range with descendant partition;
 8         Remove the range if it doesn't cross the next partition boundary;
       end
 9     for Each page of range partition do
10         Load the range set page in memory;
11         for Each range in the page do
12             Join the range with descendant partition;
13             Put the range in the range cache if it crosses the next partition boundary;
           end
       end
   end
```

sampling and partition. We can take the size of this in-memory range cache out from the total memory buffer size before partitioning. So, the partitioning can be done as normal point data partitioning without considering the boundary crossing problem.

Descendant Partition Join In the *descendant partition join* algorithm, which is shown in Algorithm 1, the descendant point set is the outer set. The sampling and partitioning is based on the point set. During the join phase, each point partition is loaded in memory to be joined with the corresponding range partition and the cached ranges from previous range partitions.

In line 7 and line 12, the join operation is to join one range with all the descendant points in a partition. Since it is an in-memory operation, at first, we directly used the nested loop join. From our preliminary experimental results, we found that the performance of the nested loop join was very bad due to the high CPU and memory access cost. So, we changed the nested loop to binary search. The descendant point partition is sorted after it is loaded in memory. When a range is to be joined with the points partition, a binary search is used to locate the first point in the range. Then, we scan the sorted point set until we reach the last point in this range.

In line 13, after the join of each range, we put the range in the range cache if it crosses the next partition boundary. At the beginning of the join, the ranges in the range cache are joined with points first (see line 7). At the same time, we try to eliminate the ranges that do not cross the next partition boundary, which is shown in line 8. Thus, the range cache is maintained in such a way that only the ranges crossing the next partition boundary are kept in memory, and are used in the join of the next partition.

In a partition join, either join input data set can be chosen as the outer set to determine the partition boundaries. However, partitioning based on the larger input produces more partitions than partitioning based on the smaller input. With the smaller input set as the outer set, we can have smaller number of partitions and more sequential I/O. If the smaller input can totally fit in memory, there is no partitioning needed at all. Usually, in XML data, the size of the descendant point set is larger than that of the ancestor range set. So, partitioning based on the ancestor range set could produce better performance. We next introduce two partition join algorithms using the ancestor range set as

Algorithm 2: Segment Tree Partition Join

```
Input: (ancestor range set, descendant point set)
1  Set range cache to be empty;
2  Determine partition boundaries according to the ancestor range set;
3  Partition both range and point sets;
4  for Each partition pair in increasing order do
5      Load ancestor range partition in memory;
6      Build a segment tree for ranges in the partition and the range cache;
7      for Each page of point partition do
8          Load the point set page in memory;
9          for Each point in the page do
10             Join the point using the segment tree;
           end
       end
11     Dispose the segment tree;
12     Remove from range cache the ranges not crossing the next boundary;
13     Put ranges crossing the next boundary from range partition to range cache;
   end
```

the outer set. They are named *Segment Tree Partition Join* and *Ancestor Link Partition Join* based on their in-memory join algorithms.

Segment Tree Partition Join In partition join algorithms using the ancestor range set as the outer set, the partitioning is based on the ancestor range set. As in the *descendant partition join* algorithm, a range cache is also used. During the join phase, each range partition is first loaded into memory. These ranges in the partition and the ranges in the range cache are together to be joined with each descendant point from the inner partition. Because the nested loop join is inefficient, in the segment tree partition join algorithm, which is shown in Algorithm 2, the segment tree [10] is used as the in-memory algorithm to quickly find a set of ranges containing a point. The worst case space complexity of the segment tree is $O(N \cdot logN)$, where N is the number of ranges [10]. In our implementation, we used several techniques to minimize the size of the segment tree data structure. In this in-memory join context, the ranges and their end-points are already known. No insertion and deletion of end-points is required after the tree is built. We can utilize this property to build a static segment tree, which is more compact than a dynamic one.

The first step of building a segment tree is to sort the end points of the ranges in memory (duplicate points should be eliminated). Let us suppose the number of ranges is N. Because the start point of each range is a unique preorder ID number, there are at least N points in the segment tree. After sorting, an *virtual* empty segment tree is there, because the parent-child relationship in the segment tree can be determined by the array index calculation. There is no additional space needed for the structure pointers of the tree. However, the number of segment tree nodes is twice the size of the point array. When a range is inserted to the segment tree, a pointer is needed to store the linked list for each segment tree node. So, at least $2N$ pointers are needed. For each range, at least one linked list record is associated with it. Each linked list record contains the range ID and the next link pointer. In total, we need at least $3N$ pointers, and N range ID's. If pointers and ID's are represented by integers, then at least $4N$ integers are required for the segment tree. This is a large memory requirement, since in our implementation, each XML element node record only uses four integers. In this case, half of the memory is used for the segment tree.

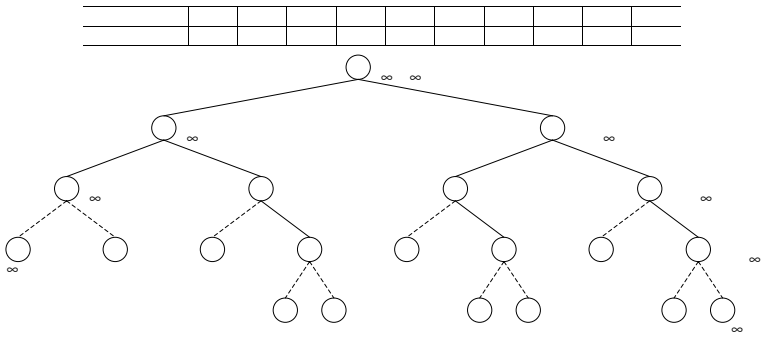


Fig. 2. End Point Array and Its Segment Tree

As an example, the table in Figure 2 shows a set of point values along with their index in an array. In the segment tree illustrated below, the number in a node circle is the index value. It is empty for leaf nodes, since they are virtual nodes. The point value and the node range are labeled near each node. The root index value of each sub-tree is the middle index value of the index range of the sub-tree. That is $tree\ root = \lfloor (start\ index + end\ index)/2 \rfloor$. For example, index value 4 is the root of the index range [0,9], which is the whole tree.

Since memory size is limited, the more memory used for in-memory index like segment tree, the smaller memory is left for partitions. To solve the large memory occupation problem of the segment tree, we propose to use *Ancestor Link* algorithm for in-memory join, which is described in the next section.

Ancestor Link Partition Join The control flow of the *Ancestor Link Partition Join* algorithm is the same as the segment tree partition join algorithm (Algorithm 2), except that the in-memory join is now the *ancestor link join*. Instead of building a segment tree, an ancestor link data structure is used. Each descendant point is joined with all the in-memory ranges using this ancestor link data structure.

According to the *range containment property*, for any two ranges, either one range contains the other range, or they are disjoint. We can build a *range tree* according to the containment relationship. In the range tree, a child range is directly contained in the parent range, which is the smallest range containing the child. If a descendant point p is contained in a range R in the range tree, then this point p is also contained in all the ancestor ranges of range R in the range tree. Using this range tree, we can efficiently perform the join between a point and a set of ranges.

Since all range records to be joined are already in the buffer memory, no additional range information needs to be duplicated again in the range tree. In our implementation, the range tree is only a pointer array, which has the same size as the number of the ranges in memory. The positions of pointers correspond to the positions of the ranges in the range array. The content of a pointer is the array index of the parent range. So, an edge in the range tree is pointing from a child node to a parent node. To build the range tree, we first sort all the ranges by the start points. After the sorting, the ranges are also in sorted preorder with respect to the range tree. Then we scan the range array once with a pointer stack to find the parent of each range and update the ancestor pointer array. This algorithm is shown in Algorithm 3.

Algorithm 3: Build Ancestor Link

```
   Input: (sorted range array, range tree array)
1  Initialized pointer stack to be empty;
2  for Each range in the range array do
3      while pointer stack is not empty and the top of pointer stack is not the parent of the
       current range do
4          Pop the pointer stack;
       end
5      if pointer stack is empty then
6          Set the current range tree pointer to be null;
       else
7          Set the current range tree pointer to be the top of the pointer stack;
       end
8      Push the current range pointer to the pointer stack;
   end
```

Next, we show how to find the first (smallest) range containing a given point p in the range tree. Since the range array is sorted by the start point, we can use binary search to find the range R_p that may contain the point p. That is R_p is the first range whose start point is smaller than p. If R_p contains p, then all the ancestors of R_p in the range tree also contain p. If R_p does not contain p, we can follow the ancestor link of R_p to find the ranges that contain p. The main advantage of the ancestor link is its small memory requirement. The whole structure is only an array of pointers. The size is at least less than one fourth of the segment tree. So, more memory can be used for partitions.

3 Experiment

We implemented these three partition join algorithms discussed in the paper. They are referred to as *partition-d*, *partition-s* and *partition-a* for Descendant partition join, Segment-tree partition join and Ancestor-link partition join algorithms respectively. For performance study, we also implemented the sort-merge based algorithms similar to the Stack-Tree join algorithm (Stack-Tree-Desc) [12]. Two variations of sort-merge join algorithms, *sortmerge-s* and *sortmerge-m*, were implemented. Sortmerge-s sorts each input into a single run before the join phase, while sortmerge-m combines the last merge phase of sorting with the join phase.

The above algorithms were all implemented on top of a paged file layer, which also provides buffer management functionality. In our experiments, the page size was 4K bytes. The input files were paged files, and were directly generated from the data-set in random order. We have chosen two data sets, Shakespeare and DBLP, to demonstrate the performance. The Shakespeare data set is the Shakespeare's plays in XML format. The DBLP data set is a computer science bibliography [9]. In our experiment, we used the conference portion with a raw data size 58MB.

In the experiments, we measured the elapsed time (both CPU and IO) of query processing. For fair comparison, the sorting cost for sort-merge based algorithms, and the sampling cost for partition based algorithms were both counted in the performance measure. Since the cost of output generation is the same regardless of algorithms applied, the output cost was not included in the measurements. Experiments were performed on an Intel workstation with a Pentium 4 1.6GHz CPU running Solaris 8 for Intel platform. This workstation has 512M bytes of memory and a 40GB EIDE disk drive (with 7200

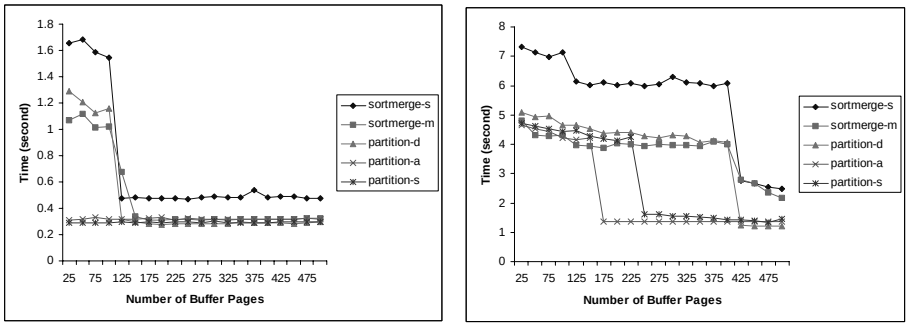
Data Set	Query	Ancestor		Descendant	
		Record#	Page#	Record#	Page#
Shakespeare	ACT//SPEECH	185	1	31028	122
	SPEECH// LINE	31028	122	107833	423
DBLP	dblp//author	783	4	320300	1257
	inproceedings//author	140936	553	320300	1257

Table 1. Queries and Description

RPM and 8ms average seek time). The disk is locally attached to the workstation and used to store XML data. We used the direct I/O feature of Solaris for all experiments to avoid operating system's cache effects.

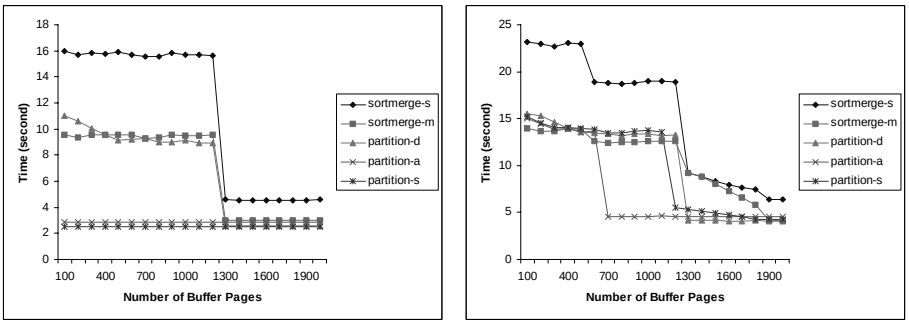
3.1 Query Performance and Analysis

Table 1 describes the queries we used in the experiments. It also provides the ancestor range set size and the descendant point set size information. All queries are of the form "$E_A//E_B$", which is to find the ancestor-descendant element pairs.



(a) ACT//SPEECH (b) SPEECH//LINE

Fig. 3. Shakespeare Data Queries



(a) dblp//author (b) inproceedings//author

Fig. 4. DBLP Data Queries

Figure 3 shows the performance for two queries on the Shakespeare data and Figure 4 shows the performance for the DBLP data. The performance measure is elapsed

time in seconds with different number of memory buffer pages. Although the size of the DBLP data is much larger than that of the Shakespeare data, we observed similar performance trends.

In Figure 3(a) and Figure 4(a), the ancestor set size is small. Since the partition-a and the partition-s algorithms use the ancestor set as the outer set, there is no need to do partitioning at all if the memory can hold the outer set. So, their performances are better than others when the total buffer size is small.

For both sort-merge join and partition join algorithms, if memory buffer can hold both sets, there is only one scan of the input files needed to load data in memory. That is the minimum cost for the query processing. So, when the memory buffer size is large enough, there is a sharp performance increase. After this, the performance remains almost constant. This trend can be seen from Figure 3(a) and Figure 4(a).

In the sortmerge-m algorithm, the last merge phase of sorting is combined with the join phase, so the last merge scan is saved. On the other hand, for the sortmerge-s algorithm, the last sorting merge scan of input files is always needed. It increases both the I/O and the CPU cost. Among these algorithms, the sortmerge-s algorithm is the worst, which is clearly shown in Figure 3 and Figure 4.

In Figure 3(b) and Figure 4(b), the ancestor set is large. When neither join set can fit in memory, all algorithms have to scan the input files at least twice. In this case, the partition join algorithms (partition-d, partition-a and partition-s) and the sortmerge-m algorithm have similar performance. From this result, we can also see that the sampling cost for the partition based algorithms is low. The reason is that the number of samples is small compared to the whole set, and we also optimized the sampling process using clustered read technique as described in Section 2.2.

With the memory size getting larger, the performances of the partition join algorithms improve faster than the sortmerge-m algorithm, since the partition join algorithms can avoid partitioning when one join set can fit in memory. For the sortmerge-m algorithm, it can keep all runs in memory only if the memory is large enough to hold both join sets. Otherwise, additional I/O for runs is unavoidable. Among the partition join algorithms, the performance of the partition-a algorithm is the best. The in-memory join data structure *ancestor link* requires less memory than *segment tree*. So, more memory can be used to hold partitions.

The above experimental results provide useful information for query optimization. If one of the two join sets is much smaller that the other, using the partition join algorithms require less memory. The query optimizer can choose the smaller set as outer set and the corresponding partition join algorithm.

4 Previous Work

Sort-merge join has been widely used in relational database. For equality queries, hash-based join can be used as an alternative to sort-merge join. Comparisons of sort-based and hash-based algorithms show that many dualities exist between the two types of algorithms and both should be available in a query-processing system [4]. There are a large amount of work has been done in the temporal database area to process temporal intersection joins [5], in which join predicates over time attributes are mostly of the inequality type. To process valid-time joins, a partition-based evaluation algorithm has been proposed [11]. This algorithm utilizes in-memory cache to store “long-lived” tuple and avoids the replication of tuples in multiple partitions. For XML data, with the

introduction of *the extended preorder numbering scheme* [6], XML path queries can be processed using traditional relational database techniques. For example, sort-merge based algorithms like $\mathcal{EE}$**-Join**, $\mathcal{EA}$**-Join** and *Structure join* [12] have been proposed. *Stack-Tree Join* [12] algorithms utilizing in-memory stack to hold ancestor nodes have been proposed to deal with structural join [12].

5 Conclusion

In this paper, we have proposed partition based algorithms, which can be chosen by query optimizer according to the characteristics of the inputs. An in-memory range cache is used to hold ranges crossing partition boundaries. In the Ancestor Link partition join algorithm, we propose to use the *Ancestor Link* data structure, which is much smaller in size compared to the segment-tree. So, more memory can be used for holding partitions. The experimental results show that the Ancestor Link algorithm can make the best use of memory buffer and take advantage of the uneven sized inputs. We believe that those algorithms are necessary choices for query optimizer to consider during XML query processing.

References

[1] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 436–447, 1998.

[2] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Scheneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th VLDB Conference*, Barcelona, Spain, September 1991.

[3] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd VLDB Conference*, pages 436–445, Athens, Greece, September 1997.

[4] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. Sort versus hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):934–944, 1994.

[5] Himawan Gunadhi and Arie Segev. Query processing algorithms for temporal intersection joins. In *Proceedings of the 7th Inter. Conference on Data Engineering*, Kobe, Japan, April 1991.

[6] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *Proceedings of the 27th VLDB Conference*, Rome, Italy, September 2001.

[7] Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116:195–226, 1993.

[8] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proceedings of the 25th VLDB Conference*, pages 315–326, Edinburgh, Scotland, September 1999.

[9] Michael Ley. DBLP Computer Science Biblography. http://www.informatik.uni-trier.de/-ley/db/index.ht ml, November 2001.

[10] Franco P. Preparata and Michael Ian Shamos. Computational geometry - an introduction. Springer-Verlag, Berlin/Heidelbrg, Germany, 1985.

[11] Michael D. Soo, Richard T. Snodgrass, and Christian S. Jensen. Efficient evaluation of the valid-time natural join. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 282–292. IEEE Computer Society, 1994.

[12] Divesh Srivastava, Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, and Yuqing Wu. Structural joins: A primitive for efficient xml query pattern matching. In *Proceedings of the 18th Inter. Conference on Data Engineering*, San Jose, California, February 2002.

[13] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM-SIGMOD Conference*, Santa Barbara, CA, May 2001.