XQuery Processing in Natix with an Emphasis on Join Ordering

Norman May Sven Helmer Carl-Christian Kanne Guido Moerkotte
Universität Mannheim
Lehrstuhl für Praktische Informatik III
Mannheim, Germany
[norman|helmer|kanne|moer]
@pi3.informatik.uni-mannheim.de

ABSTRACT

We give an overview on how XQuery processing works in our native XML database system Natix. After a brief description of the query compiler we focus on the aspect of join ordering when generating query execution plans. Here we show that better plans can be found when extending the search space of the plan generator.

1. INTRODUCTION

When evaluating queries there are two principle avenues of approach: interpretation and compilation. Due to the fact that query compilation has proved its worth in many commercial relational database management systems (DBMSs), it is the current method of choice. The foundation of this process is an algebra into which a query is compiled. The existence of many alternative query execution plans (QEPs) equivalent to a query is based on algebraic equivalences and is of fundamental importance to find good QEPs. After a query has been compiled into a QEP, the QEP is evaluated by the query execution engine (QEE).

Applying this process to the evaluation of XQuery queries is quite difficult at the moment. There are two reasons for this. First, query compilers and query execution engines in XML database systems are not as sophisticated as those in relational systems yet. Second, a plan generator is severely limited in the number of options (e.g. by having to maintain document order during query evaluation).

We tackle the first problem with an algebra-based approach of query compilation and execution in our native XML database system Natix. We want to be able to transfer the achievements made in traditional DBMSs to the XML world. For the second point we show how the search space of a plan generator can be extended significantly and how this leads to better plans. Since plan generation is too wide a field to be covered in a short paper, we concentrate on the important area of join ordering.

The paper is organized as follows. In Section 2 we give a general overview on XQuery processing in Natix and have a closer look at the query execution engine and query compilation. Following that, we extend the search space of the plan generator for ordering joins step by step in Section 3. Finally, a summary concludes the paper in the last section.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission of the authors.
Informal Proceedings of the *First International Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, June 17-18, 2004, Paris, France.

2. XQUERY PROCESSING IN NATIX

Since the approach of query compilation has been very successful, we also opted for this evaluation technique when implementing XQuery processing in our native XML database system Natix (for details see [4]). The two components responsible for query processing in Natix are the query compiler and the query execution engine. Before going into details on the execution engine and the query compiler, we give a general overview of the architecture of Natix in the following section and in Figure 1. This figure also shows how the compiler and execution engine are embedded into the system.

2.1 General Overview of Natix

Natix consists of three layers. The bottom layer contains the storage engine including buffer management. The middle layer contains the services one typically expects from a database management system (DBMS). Among these services are the query compiler (QC) and the query execution engine (QEE). The top layer focuses on system control and provides the interface to the system via a C++ library. Applications, like the interactive shell included in the Natix distribution, are developed by using this interface.

When formulating queries, we have two alternatives. First, similar to dynamic SQL, an XQuery query can be passed as a string parameter via the C++ API to the Natix core. Second, ad-hoc queries can be evaluated within the interactive shell. In both cases, the query is passed to the QC where a query evaluation plan (QEP) is generated. The QEE then evaluates the QEP and returns the result.

2.2 The Query Execution Engine

A detailed look at QEPs and the QEE can be found in the Natix overview paper [4]. We summarize the relevant aspects below.

In a nutshell, the QEE consists of an iterator-based implementation of algebraic operators. They process ordered sequences of tuples. Tuple attributes either hold base type values such as strings, numbers and tree node references, or again contain ordered sequences.

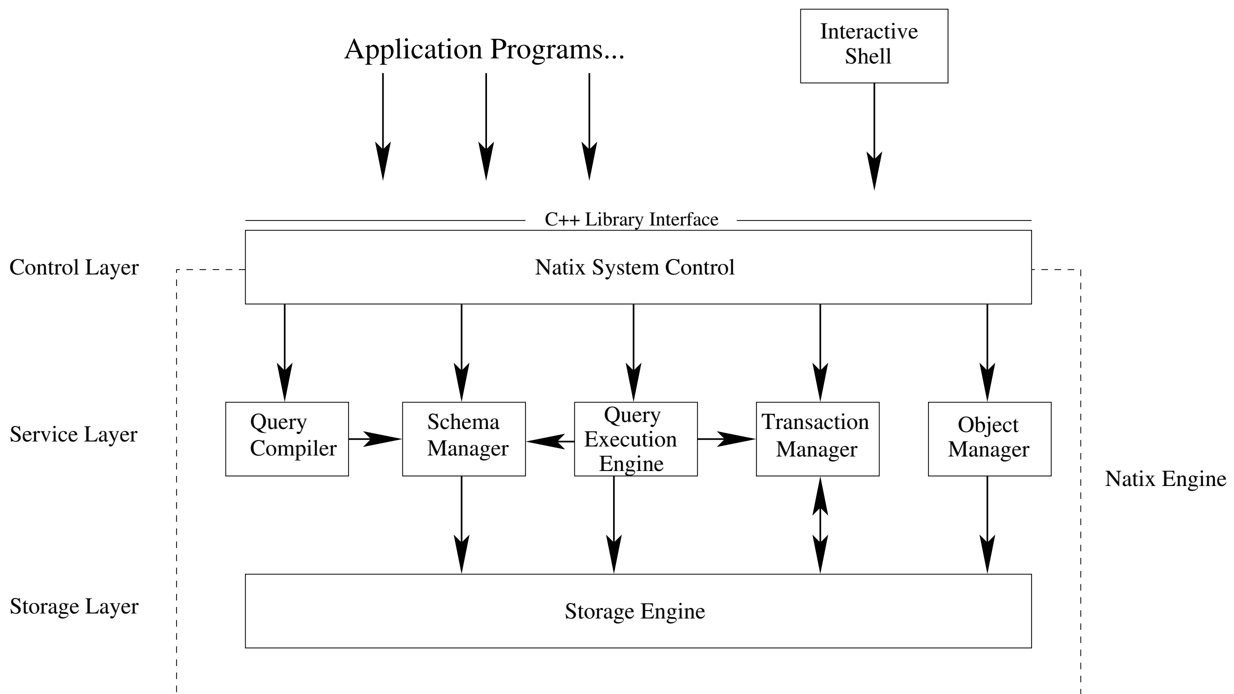

Figure 1: Natix' Architecture

The iterator model has been slightly extended to deal more efficiently with group boundaries and nested queries. It also contains special operators to construct query results [5, 6].

Subscripts of the algebraic operators (such as join or selection predicates) are expressed in an assembly-like language, and are evaluated using the Natix Virtual Machine (NVM). The NVM avoids the overhead associated with interpreted operator trees.

XML data is accessed through special NVM commands which directly access a clustered persistent XML storage format in the page buffer. Hence, expensive representation changes such as pointer swizzling of the data during query execution are not required. Our compact format also results in few page and CPU cache misses, and compares favorably to pure pointer-based main memory representations.

2.3 The Query Compiler

As the query compiler has not been described yet in our previous publications, we focus on this part of Natix here. The architecture of the query compiler follows the rather traditional six phase approach (see Fig. 2).

In the first step, the parser generates an abstract syntax tree.

Following that, the NFST module performs **N**ormalization, **F**actorization of common subexpressions, **S**emantic analysis, and **T**ranslation into an internal representation. This internal representation is a mixture of our algebra and a calculus representation.

After that we can start rewriting the queries. Most importantly, we merge views (which are called functions in XQuery), unnest queries, and rewrite XPath expressions. Since a nested query results in a nested algebraic expression which in turn requires an inefficient nested-loop evaluation, we try to unnest queries whenever possible. This phase is rather involved and details are described in a series of papers [16, 17, 18, 19]. Details on rewriting XPath expressions in Natix can be found in [1, 11, 12].

During the plan generation phase we replace the calculus representation of query blocks with algebraic expressions. Here the plan generator is faced with numerous alternatives because now the concrete execution order as well as an actual implementation of the algebraic operators is defined. Dynamic programming is the prevalent approach when generating execution plans. Over the years, the search space that generators had to deal with was continuously expanded. Starting with left-deep trees for join ordering prohibiting cross products this went on to bushy trees allowing not only cross products, but a plethora of other operators (besides joins) resulting in better and better plans. However, the core of plan generators is still a dynamic programming algorithm[1] with join ordering as its most important task. That is why we take a more detailed look at this subject in Section 3.

In the last but one phase, the generated plan is rewritten. Typically, only small changes are made to the plan. For example, two successive selections are merged. If the plan generator does not consider group operators, then plan rewrite can try to push them down in a way that is beneficial to the costs.

Finally, we generate the code for the QEP.

3. JOIN ORDERING

Before going into the details of join ordering, let us point out some fundamental differences between join ordering in the relational case and join ordering in the context of XQuery.

[1]or its top-down counterpart memoization

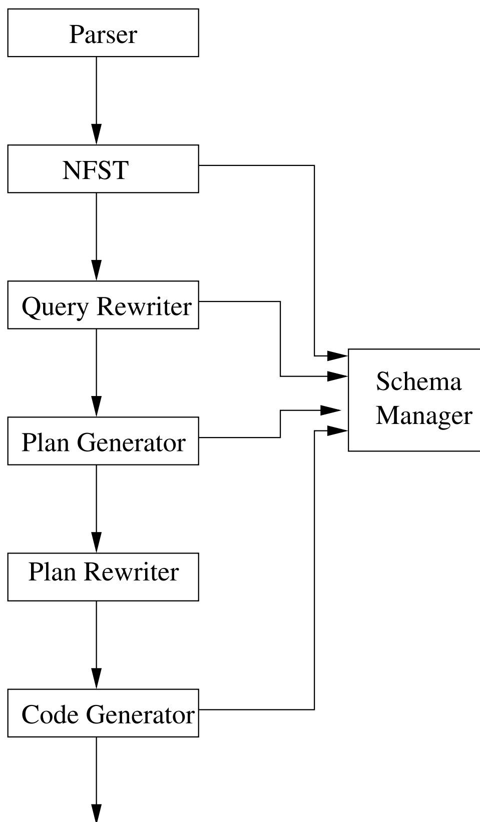

Figure 2: Query Compiler

In general, XQuery demands that joins are order-preserving, except if the user explicitly disregards order[2]. An order preserving join is associative but not commutative. This results in a much smaller search space for join ordering. Let us denote by $C(n)$ the Catalan numbers. Then, for ordering n order-preserving joins, there are "only" $C(n) = 1/(n+1)\binom{2n}{n}$ different execution plans. Moreover, the join ordering problem can be solved in polynomial time ($O(n^3)$) independently of the query graph and the cost function [21]. This is in stark contrast with the traditional join ordering problem. In general it is NP-hard and the search space size for generating bushy trees is $(n+1)!C(n) = (2n)!/n!$ for n joins [23]. These numbers become more disparate if we consider left-deep trees only. There exist exactly two left-deep order-preserving plans[3] whereas there are $(n+1)!$ left-deep plans in the classical context [23]. (Note that all formulas above refer to plans allowing cross products.) Although a larger search space is not always a guarantee to find better plans, we show that for XQuery this is true.

3.1 Example Query

With the help of the following example query we will show how extending the search space will influence the quality of the QEP. We have chosen a simple SPJ-query that is well-suited to explain the tasks performed by the plan generator during query optimization.

```
for $x1 in document("d1.xml")//K,
    $x2 in document("d2.xml")//A,
    $x3 in document("d3.xml")//A
where
    $x1/*/@a eq $x2/*/@a
and $x2/*/@d eq $x3/*/@d
and $x1/*/@c eq $x3/*/@c
return
  <result>
    <x1>{ $x1/@id }</x1>
    <x2>{ $x2/@id }</x2>
    <x3>{ $x3/@id }</x3>
  </result>
```

Before delving into the details of optimizing this query we will explain some properties of the used example document collection.

We used a tool to generate the documents, which basically all have the same structure. Each document contains 5000 elements of the types A, B, C, D, and so on. The frequency with which the elements appear is halved for each subsequent letter. So A and K appear in the ratio 1024:1. Also, every element contains the attributes a, b, c, d and an id. The range of these attributes values doubles for each subsequent letter. That means attribute a only takes on the values 0 or 1, while attribute c may have the values from 0 to 7, the values of attribute d range from 0 to 15, while id takes on a unique value for each element. The values for each attribute a, b, c, and d are uniformly distributed.

This has the following implications on the query evaluation. The variable $x1$ is bound to a sequence that is much smaller than the sequences bound to $x2$ and $x3$. (Note also that the query graph contains a cycle $x1 \leftrightarrow x2 \leftrightarrow x3 \leftrightarrow x1$.) The selectivity of the predicates also varies from the first to the last because in the first predicate only two different attribute values occur as opposed to sixteen in the second predicate and eight in the last predicate.

3.2 Naive Translation

The semantics of XQuery demand that the result of a query is computed in document order. That means, when an element A is visited before element B (in the traversal of an XML document), then A is located before B in the resulting sequence of elements. When sequences of items from multiple sources are combined using `for` clauses the order of the `for` clauses in the query determines the order of the combined sequence of items.

The canonic translation of a query results in a sequence of (order-preserving) Cartesian products connecting the sequences in the `for` clauses. When we have join dependencies between those sequences, a translation into an order-preserving join is possible [2]. Thus, a naive query translator will turn the example query into a sequence of nested loop joins (see query plan QP1 in Figure 3). We employ nested loop joins due to their order-preserving property. Typically, hash join algorithms do not have this property, because they partition the input on secondary storage. Sort-merge joins resort their input on the join attribute (which does not necessarily match document order).

Evaluating the join predicates is done via a nested query.

[2]for example via the **unordered** function, or in constructs such as aggregation, quantifiers or explicit sorting.

[3]In addition to $(((R_1 \bowtie R_2) \bowtie R_3) \ldots \bowtie R_n)$, which preserves the order of R_1, $R_1 \bowtie R_2, \ldots$ in each join, there is the alternative $(((R_n \bowtie R_{n-1}) \bowtie R_{n-2}) \ldots \bowtie R_1)$, preserving the order of R_i in each join. The second left-deep plan is actually the "mirrored" right-deep plan.

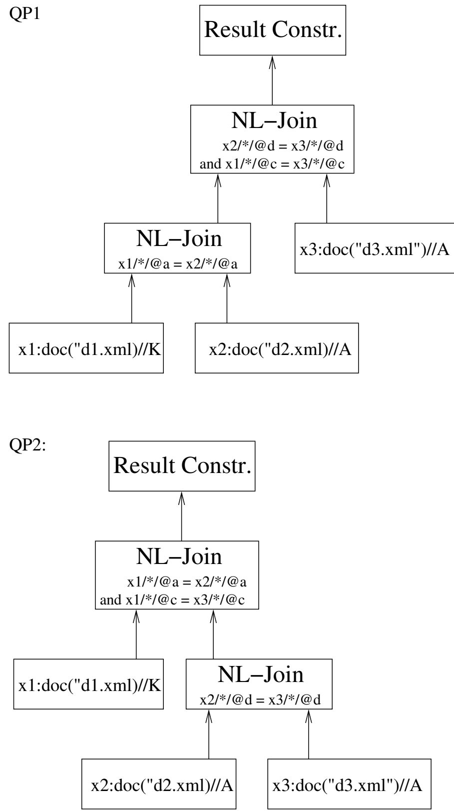

Figure 3: Naive Plans

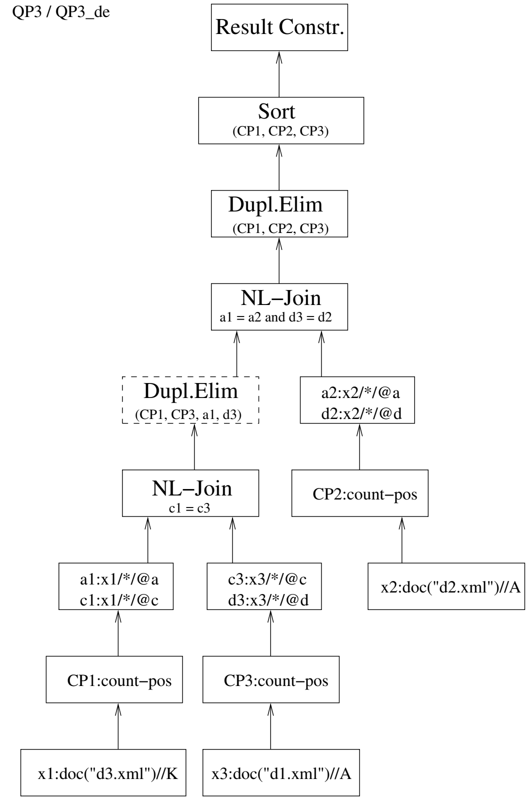

Figure 4: Optimized Join Order

For example, for the first join predicate all the attribute values of `$x1/*/@a` are generated and compared to those generated for `$x2/*/@a`. In case we find a value that both sequences have in common, the predicate is true. (Another variant to evaluate the join predicates would be to attach the sequence of attribute values to each tuple of x1, x2, and x3 and employ an adapted version of a set-valued join algorithm [13, 20, 24].)

3.3 Ordering Order-preserving Joins

Since an order-preserving join is associative, the plan generator may choose between different plans. This choice depends primarily on the input cardinality and the selectivity of the joins. (Further equivalences for optimizing plans that preserve document order and an algebra on lists were presented in [27]. A plan generator may use them as a foundation for transformations in an order-preserving optimizer.) For our example query the plan generator may choose between two different queries, QP1 and QP2 (see Figure 3). When running the queries on Natix, QP1 takes 587.75 seconds, while QP2 takes 395.18 seconds.

As we can see there is a better alternative to the naive translation of our query. This resembles results obtained by Wu et al. in [29] in the context of structural joins for evaluation of XPath. However, this is just a first step in extending the options of the plan generator. In the following sections we present some approaches to increase the search space even further.

3.4 Disregarding Order Preservation

The cardinality of the input and the selectivity of the join predicate are also important parameters for generating the cost-optimal order of joins that do not preserve order. A rule of thumb for plan generators is, e.g., joining the input with the smallest cardinality and highest selectivity first. However, the naive order-preserving evaluation limits our choices significantly. In order to lift these restrictions, we now disregard document order during query processing. As a consequence, for n join operators we now have $(n+1)!C(n)$ possible orderings (due to the commutativity of a join operator that does not preserve order). Thus we get twelve different plans for our example query increasing our search space considerably. Another consequence is that we have to sort the final result to make sure that document order is obeyed, since it may have been corrupted by using non-order-preserving join operators or by reordering the joins exploiting the commutativity.

Let us have a look at some implementation details. First of all we use a map operator to add an attribute containing the position to each tuple in a sequence (see `count-pos` in Figure 4). In this way we are able to reconstruct document order later on. We also unnest the join attributes. For example, for the first join predicate this means that we generate a tuple for each attribute value in `doc("d1.xml")//K/*/@a` and `doc("d2.xml")//A/*/@a` and then join the two sequences on these attribute values. As this results in duplicates, we also have to insert a duplicate elimination step after joining all sequences.

The best QEP we found among all twelve plans, called QP3, is depicted in Figure 4. (Note that the dashed box is not included in QP3 yet, it is introduced in the next section

with the plan QP3_de.) In this case the sequences for x1 and x3 are joined first. In our case this is better than joining the smallest sequence with the least selective predicate or joining the largest sequences with the most selective predicate first. Comparing the execution time of this plan (125.48 seconds) to QP2 from the previous section, we see that we improved the performance approximately by a factor of three.

3.5 Pushing Duplicate Elimination

As noted in the previous section, the joins might contain duplicates in their results. We now extend the search space of the plan generator to consider introducing an additional duplicate elimination after each join. In a query with n joins we may introduce at most $n-1$ additional duplicate elimination operators. The final duplicate elimination is mandatory in our case. Note that introducing the duplicate elimination does not harm the document order.

Duplicate elimination does not come for free. It is not trivial to decide if introducing duplicate elimination is beneficial, and this issue has been addressed in the context of *early aggregation* in the literature [15]. Deciding on the effectiveness of doing so is the task of the cost-based query optimizer [14].

In our example the search space is extended only by one additional alternative for QP3. The resulting plan QP3_de contains an additional duplicate elimination after the join of $x1$ and $x3$ and yields an execution time of 103.21 seconds. As the example query shows, it is worth to extend the search space of the plan generator to consider introducing duplicate elimination. However, it is not always worth doing this extra work, but it gives more freedom to the plan generator.

3.6 Choice of Join Algorithm

When giving up the order-preserving property of the join operators, we can go one step further and employ different non-order-preserving implementations of join operators. Consequently, we extend the search space even more by allowing the plan generator to choose among several different join evaluation techniques [3, 7, 8, 9, 10, 26].

In our case we implemented two other join algorithms: a hash-based and a sort-based one. The hash-based join is a block-wise nested loop algorithm that pipes main-memory sized blocks of the outer producer into a hash table and probes each block with all the tuples of the inner producer. The sort-based join is a standard n:m sort-merge join.

For our measurements, we restrict ourselves to the most efficient plan from the previous section. We replace the nested loop joins either by hash joins or sort-merge joins. For the hash join we have a query evaluation time of 7.34 seconds and for the sort-merge join an evaluation time of 8.00 seconds. Comparing this to the original optimal order-preserving plan, we improved the execution time by almost 50 times even on this small example query with just two join operators.

4. CONCLUSION AND OUTLOOK

In XQuery processing there is a lot of potential in optimizing the evaluation that is waiting to be tapped. Our contribution is twofold. On one hand, we give an overview on how an algebra-based query compiler and query execution engine can look like for a native XML database system. On the other hand, we show how to help the plan generator in producing better plans by extending its search space.

Join Algorithm	Preserve Order?	Dupl. Elim.	Time (s)
Nested Loop	Yes	No	395.18
Nested Loop	No	Final	125.48
Nested Loop	No	Pushed	103.21
Hash Join	No	Pushed	7.34

Table 1: Performance summary

The resulting execution times are summarized in Table 1. Our main goal was not so much the presentation of new techniques, but to make people aware of the existing possibilities.

There are areas of optimization we have not touched in this paper. For example, it may pay off to keep a close watch over the order of tuples during query evaluation. Order may only be partially lost when using non-order-preserving operators. Optimization of order has been investigated in [26]. These ideas were extended by [28] to incorporate grouping and secondary sorting. They extend the concept of interesting orders proposed by [25] by including functional dependencies, key properties and predicates in their optimization procedure. For a general framework, see [22]. So, these techniques could be applied considering document order as one interesting order.

5. REFERENCES

[1] M. Brantner. Algebraic XPath evaluation in Natix. Master's thesis, University Mannheim, 2004.

[2] S. Cluet and G. Moerkotte. Nested queries in object bases. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Proc. of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 226–242. Springer, 1993.

[3] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. ACM SIGMOD Conference, Boston, Massachusetts, USA*, pages 1–8. ACM Press, June 1984.

[4] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. *VLDB Journal*, 11(4):292–314, 2002.

[5] T. Fiebig and G. Moerkotte. Algebraic XML construction in Natix. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, pages 212–221, 2001.

[6] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2002.

[7] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. 10th ICDE Conference, Houston, Texas, USA*, pages 406–417, February 1994.

[8] G. Graefe, A. Linville, and L. D. Shapiro. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):934–944, December 1994.

[9] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[10] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about *ad hoc* join

costs. *VLDB Journal*, 6(3):241–256, May 1997.

[11] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath expressions into algebraic expressions parameterized by programs containing navigational primitives. Technical Report 11, University of Mannheim, 2002.

[12] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of xpath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, 2002. 215-224.

[13] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 386–395, 1997.

[14] S. Helmer, T. Neumann, and G. Moerkotte. Estimating the output cardinality of partial preaggregation with a measure of clusteredness. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 656–667, 2003.

[15] Per-Åke Larson. Data reduction by partial preaggregation. In *18th Int. Conf. on Data Engineering*, San Jose, California, 2002.

[16] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. Technical report, University of Mannheim, 2003.

[17] N. May, S. Helmer, and G. Moerkotte. Quantifiers in XQuery. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, pages 313–316, 2003.

[18] N. May, S. Helmer, and G. Moerkotte. Three Cases for Query Decorrelation in XQuery. In *XML Database Symposium*, pages 70–84, 2003.

[19] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *Proc. IEEE Conference on Data Engineering*, pages 239–250, 2004.

[20] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. on Database Systems*, 28(1):56–99, 2003.

[21] G. Moerkotte. Constructing Optimal Bushy Trees Possibly Containing Cross Products for Order Preserving Joins is in P. Technical Report 12, University of Mannheim, 2003.

[22] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, page to appear, 2004.

[23] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 314–325, 1990.

[24] K. Ramasamy, J.M. Patel, J.F. Naughton, and R.Kaushik. Set containment joins: The good, the bad, and the ugly. In *Proc. of the 26th VLDB Conference*, Cairo, Egypt, August 2000.

[25] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conference, Boston, Massachusetts, USA*. ACM press, May 1979.

[26] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proc. of the ACM SIGMOD Conference, Montreal, Quebec, Canada*, pages 57–67. ACM Press, June 1996.

[27] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Bringing order to query optimization. *SIGMOD Record*, 31(2):5–14, 2002.

[28] Xiaoyu Wang and Mitch Cherniack. Avoiding sorting and grouping in processing queries. In *29th VLDB, Berlin, Germany*, pages 826–837, September 2003.

[29] Y. Wu, J.M. Patel, and H.V. Jagadish. Structural join order selection for xml query optimization. In *19th Inf. Conf. on Data Engineering (ICDE'03)*, pages 443–454, Bangalore, March 2003.