Reihe Informatik
12 / 2003

Constructing Optimal Bushy Trees Possibly Containing Cross Products for Order Preserving Joins is in P

Guido Moerkotte

Constructing Optimal Bushy Trees Possibly Containing Cross Products for Order Preserving Joins is in P

Guido Moerkotte
University of Mannheim, D7, 27, 68131 Mannheim, Germany
moerkotte@uni-mannheim.de, fon(fax):+49 621 181 2582(88)

August 18, 2003

Abstract

One of the main features of XQuery compared to traditional query languages like SQL, is that it preserves the input order—unless specified otherwise. As a consequence, order-preserving algebraic operators are needed to capture the semantics of XQuery correctly. One important algebraic operator is the order-preserving join.

The order-preserving join is associative but, in contrast to the traditional join operator, not commutative. Since join ordering (i.e. finding the optimal execution plan for a given set of join operators) has been an important topic of query optimization for SQL, it is expected that it will also play a major role in optimizing XQuery. The search space for ordering traditional joins is exponential in size. Although the lack of commutativity reduces the search space for ordering order-preserving joins, we show that it is still exponential. This raises the question whether the join ordering problem is also NP-hard, as in the traditional setting. We answer this question by introducing the first polynomial algorithm that produces optimal bushy trees possibly containing cross products.

1 Introduction

XQuery[1] specifies that the result of a query is a sequence. If no `unordered` or `order by` instruction is given, the order of the output sequence is determined by the order of the input sequences given in the `for` clauses of the query. If there are several entries in a `for` clause or several `for` clauses, order-preserving join operators [CKK98] can be a natural component for the evaluation of such a query.

Whenever there are several join operators, whether order-preserving or not, the problem arises to find an optimal (with respect to some given cost function) query evaluation plan. This problem is termed join ordering. It has been treated first in the seminal paper by Selinger et al. [SAC+79]. They proposed to use dynamic programming to solve the problem. Since the search space for plans is rather large, they proposed three heuristics to reduce it:

[1] see http://www.w3.org/XML/Query

1. push selections down as far as possible,

2. avoid cross products, and

3. restrict the evaluation plan to left-deep trees.

Left-deep trees are those operator trees where the right input of every join operator is a base table. Even with these restrictions, the search space is exponential. In fact, for n join operators there are $n!$ possible orderings. Using dynamic programming, the search space is diminished to 2^n.

The use of dynamic programming instead of a polynomial algorithm was justified later by Ibaraki and Kameda [IK84], who proved that constructing optimal left-deep trees for a special cost function for n-way nested-loops joins generally is NP-hard. However, they also gave a polynomial algorithm in case the join graph is acyclic and the cost function has the so called ASI property. This result was made more popular by Krishnamurthy, Boral, and Zaniolo [KBZ86], who also indicated that the complexity of the original algorithm can be improved to $O(n^2)$.

Ono and Lohman looked at restrictions 2) and 3) above and gave examples that sometimes it is favorable to construct bushy trees (i.e. general trees and not just left-deep trees) and to perform cross products even if the join graph is connected. They also gave search space estimates for these cases. Hence, the problem arose whether it is possible to find optimal left-deep trees which might contain cross products for acyclic queries in polynomial time. Cluet and Moerkotte [CM95] showed that this is quite unlikely by proving for a simple cost function which has the ASI property that the problem is NP-hard. Scheufele and Moerkotte investigated the question whether there exists a polynomial algorithm to construct bushy trees for cost functions having the ASI property. Again, the result was negative. In fact, even if we do not have any join predicate, i.e. if we construct an optimal bushy tree for cross products only, the problem is already NP-hard [SM96a].

Restriction 1) above demands that selections are evaluated as early as possible. Hellerstein and Stonebraker showed that this is not always optimal [HS93, Hel94] and provided a good heuristics to avoid this problem. However, their algorithms do not always construct an optimal plan. Yayima, Kitagawa, Yamaguchi, Ohbo and Fujiwara [YKY$^+$91] provided an algorithm that produced optimal results. However, they did not take the cost of join predicates into account. This deficiency was later eliminated by Scheufele and Moerkotte [SM96b]. Both algorithms produce provably optimal results for cost functions with the ASI property and have an exponential runtime. Chaudhuri and Shim [CS96] tried to be better, but their main theorem is wrong. They corrected it later [CS99] by restricting themselves to linear cost functions. However, since every linear cost function has the ASI property, they are back to a special case of [YKY$^+$91]. It still remains an open question whether there exists a polynomial time algorithm that orders joins and selections in an optimal way.

The latter gives rise to the restriction on the problem discussed in this paper. We give a polynomial algorithm that produces bushy trees for a sequence of order-preserving joins and selections. These trees may contain cross products even if the join graph is connected. Hence, we do not apply heuristics 2) and 3) above. However, we apply selections as early as possible. The algorithm then produces the optimal plan among those who push selections down. The cost function is a parameter of our

algorithm, and we do not need to restrict ourselves to those having the ASI property. Further, we need no restriction on the join graph, i.e. our algorithm produces the optimal plan even if the join graph is cyclic.

The rest of the paper is organized as follows. For self-containedness, we define the order-preserving join in Section 2. The algorithm is given in Section 3. An example illustrating the algorithm is given in Section 4. Section 5 concludes the paper and mentions some open problems for further research.

2 Order Preserving Join

The order-preserving join operator is used in several algebras in the context of

- semi-structured data and XML (e.g. SAL [BT99], XAL [FHP02]),
- OLAP [SJS02], and
- time series data [LS03].

Before defining the order-preserving join, we need some preliminaries. The above algebras work on sequences of sets of variable bindings, i.e. sequences of unordered tuples where every attribute corresponds to a variable. Single tuples are constructed using the standard $[\cdot]$ brackets. Concatenation of tuples and functions is denoted by $\circ$. The set of attributes defined for an expression e is defined as $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$. For sequences e, we use $\alpha(e)$ to denote the first element of a sequence. We identify single element sequences with elements. The function τ retrieves the tail of a sequence, and $\oplus$ concatenates two sequences. We denote the empty sequence by ϵ.

We define the algebraic operators recursively on their input sequences. We define the order-preserving join operator as the concatenation of an order-preserving selection and an order-preserving cross product. For unary operators, if the input sequence is empty, the output sequence is also empty. For binary operators, the output sequence is empty whenever the left operand represents an empty sequence.

The order-preserving join operator is based on the definition of an order-preserving cross product operator defined as

$$e_1 \times e_2 := (\alpha(e_1) \overline{\times} e_2) \oplus (\tau(e_1) \times e_2)$$

where

$$e_1 \overline{\times} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \overline{\times} \tau(e_2)) & \text{else} \end{cases}$$

We are now prepared to define the join operation on ordered sequences:

$$e_1 \bowtie_p e_2 := \sigma_p(e_1 \times e_2)$$

where the order-preserving selection is defined as

$$\sigma_p(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else} \end{cases}$$

As usual, selections can be reordered and pushed inside order-preserving joins. Besides, the latter are associative. The following equivalences formalize this.

$$
\begin{array}{lcll}
\sigma_{p_1}(\sigma_{p_2}(e)) & = & \sigma_{p_2}(\sigma_{p_1}(e)) & \\
\sigma_{p_1}(e_1 \bowtie_{p_2} e_2) & = & \sigma_{p_1}(e_1) \bowtie_{p_2} e_2 & \text{if} \quad \mathcal{F}(p_2) \subseteq \mathcal{A}(e_1) \\
\sigma_{p_1}(e_1 \bowtie_{p_2} e_2) & = & e_1 \bowtie_{p_2} \sigma_{p_1}(e_2) & \text{if} \quad \mathcal{F}(p_2) \subseteq \mathcal{A}(e_2) \\
e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) & = & (e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3 & \text{if} \quad \mathcal{F}(p_i) \subseteq \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1})
\end{array}
$$

While being associative, the order-preserving join is not commutative, as the following example illustrates. Given two tuple sequences $R_1 = \langle [a : 1], [a : 2] \rangle$ and $R_2 = \langle [b : 1], [b : 2] \rangle$, we have

$$
\begin{array}{lcl}
R_1 \bowtie_{true} R_2 & = & \langle [a : 1, b : 1], [a : 1, b : 2], [a : 2, b : 1], [a : 2, b : 2] \rangle \\
R_2 \bowtie_{true} R_1 & = & \langle [a : 1, b : 1], [a : 2, b : 1], [a : 1, b : 2], [a : 2, b : 2] \rangle
\end{array}
$$

3 Algorithm

Before introducing the algorithm, let us have a look at the size of the search space. Since the order-preserving join is associative but not commutative, the input to the algorithm must be a sequence of join operators or, likewise, a sequence of relations to be joined. The output is then a fully parenthesized expression. Given a sequence of n binary associative but not commutative operators, the number of fully parenthesized expressions is (see [CLR90])

$$
P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n > 1 \end{cases}
$$

We have that $P(n) = C(n-1)$, where $C(n)$ are the Catalan numbers defined as $C(n) = \frac{1}{n+1}\binom{2n}{n}$. Since $C(n) = \Omega(\frac{4^n}{n^{3/2}})$, the search space is exponential in size.

Our algorithm is inspired by the dynamic programming algorithm for finding optimal parenthesized expressions for matrix-chain multiplication [CLR90]. The differences are that we have to encapsulate the cost function and deal with selections. We give a detailed example application of the algorithm in the next section. This example illustrates (1) the optimization potential, (2) that cross products can be favorable, (3) how to plug in a cost function into the algorithm, and (4) the algorithm itself. For space reasons, we do not give its formal derivation (and hence its correctness proof) from recurrences. It can be found in the report.

The algorithm itself is broken up into several subroutines. The first is `applicable-predicates` (see Fig. 1). Given a sequence of relations $R_i, \ldots, R_j$ and a set of predicates, it retrieves those predicates applicable to the result of the join of the relations. Since joins and selections can be reordered freely, the only condition for a predicate to be applicable is that all its free variables are bound by the given relations.

The second subroutine is the most important and intrigued. It fills several arrays with values in a bottom-up manner. The third subroutine then builds the query evaluation plan using the data in the arrays.

The subroutine `construct-bushy-tree` takes as input a sequence $R_1, \ldots, R_n$ of relations to be joined and a set $\mathcal{P}$ of predicates to be applied. For every possible subsequence $R_i, \ldots, R_j$, the algorithm finds the best plan to join these relations. Therefor,

```
applicable-predicates(R, P)

01  B = ∅
02  foreach p ∈ P
03    if (F(p) ⊆ A(R))
04      B+ = p
05  return B
```

Figure 1: Subroutine `applicable-predicates`

```
construct-bushy-tree(R, P)

01  n = |R|
02  for i = 1 to n
03    B =applicable-predicates(R_i, P)
04    P = P \ B
05    p[i,i] = B
06    s[i,i] = S_0(R_i, B)
07    c[i,i] = C_0(R_i, B)
08  for l = 2 to n
09    for i = 1 to n − l + 1
10      j = i + l − 1
11      B = applicable-predicates(R_{i..j}, P)
12      P = P \ B
13      p[i,j] = B
14      s[i,j] = S_1(s[i,j−1], s[j,j], B)
15      c[i,j] = ∞
16      for k = i to j − 1
17        q = c[i,k] + c[k+1,j] + C_1(s[i,k], s[k+1,j], B)
18        if (q < c[i,j])
19          c[i,j] = q
20          t[i,j] = k
```

Figure 2: Subroutine `construct-bushy-tree`

it determines some k such that the cheapest plan joins the intermediate results for $R_i, \ldots, R_k$ and $R_{k+1}, \ldots, R_j$ by its topmost join. For this it is assumed that for all k the best plans for joining $R_i, \ldots, R_k$ and $R_{k+1}, \ldots, R_j$ are known. Instead of directly storing the best plan, we remember (1) the costs of the best plan for $R_i, \ldots, R_j$ for all $1 \leq i \leq j \leq n$ and (2) the k where the split takes place. More specifi cally, the array $c[i, j]$ contains the costs of the best plan for joining $R_i, \ldots, R_j$ and the array $t[i, j]$ contains the k such that this best plan joins $R_i, \ldots, R_k$ and $R_{k+1}, \ldots, R_j$ with its topmost join. For every sequence $R_i, \ldots, R_j$, we also remember the set of predicates that can be applied to it, excluding those that have been applied earlier. These applicable predicates are contained in $p[i, j]$. Still, we are not done. All cost functions we know, use some kind of statistics on the argument relation(s) in order to compute the costs of some operation. Since we want to be generic with respect to the cost function, we

```
extract-plan(R, t, p)

01   return extract-subplan(R, t, p, 1, |R|)

extract-subplan(R, t, p, i, j)

01   if (j > i)
02     X = extract-subplan(R, t, p, i, t[i, j])
03     Y = extract-subplan(R, t, p, t[i, j] + 1, j)
04     return X ⋈_{p[i,j]} Y
05   else
06     return σ_{p[i,i]}(R_i)
```

Figure 3: Subroutine `extract-plan` and its subroutine

encapsulate the computation of statistics and costs within functions S_0, C_0, S_1, and C_1. The function S_0 retrieves statistics for base relations. The function C_0 computes the costs of retrieving (part of) a base relation. Both functions take a set of applicable predicates as an additional argument. The function S_1 computes the statistics for intermediate relations. Since the result of joining some relations $R_i, \ldots, R_j$ may occur in many different plans, we compute it only once and store it in the array s. C_1 computes the costs of joining two relations and applying a set of predicates. The next section shows how concrete (simple) cost and statistics functions can look like.

Given the above, the algorithm (see Fig. 2) fills the arrays in a bottom up manner by first computing for every base relation the applicable predicates, the statistics of the result of applying the predicates to the base relation and the costs for computing this intermediate results, i.e. for retrieving the relevant part of the base relation and applying the predicates (lines 02-07). Note that this is not really trivial if there are several index structures that can be applied. Then computing C_0 involves considering different access paths. Since this is an issue orthogonal to join ordering, we do not detail on it.

After we have the costs and statistics for sequences of length one, we compute the same information for sequences of length two, three, and so on until n (loop starting at line 08). For every length, we iterate over all subsequences of that length (loop starting at line 09). We compute the applicable predicates and the statistics. In order to determine the minimal costs, we have to consider every possible split point. This is done by iterating the split point k from i to $j - 1$ (line 16). For every k, we compute the cost and remember the k that resulted in the lowest costs (lines 17-20).

The last subroutine takes the relations, the split points (t), and the applicable predicates (p) as its input and extracts the plan. The whole plan is extracted by calling `extract-plan`. This is done by instructing `extract-subplan` to retrieve the plan for all relations. This subroutine first determines whether the plan for a base relation or that of in intermediate result is to be constructed. In both cases, we did a little cheating here to keep things simple. The plan we construct for base relations does not take the above-mentioned index structures into account but simply applies a selection to a base relation instead. Obviously, this can easily be corrected. We also give the join operator the whole set of predicates that can be applied. That is, we do not distin-

guish between join predicates and other predicates that are better suited for a selection subsequently applied to a join. Again, this can easily be corrected.

Let us have a quick look at the complexity of the algorithm. Given n relations with m attributes in total and p predicates, we can implement `applicable-predicates` in $O(pm)$ by using a bit vector representation for attributes and free variables and computing the attributes for each sequence $R_i, \ldots, R_j$ once upfront. The latter takes $O(n^2 m)$.

The complexity of the routine `construct-bushy-tree` is determined by the three nested loops. We assume that S_1 and C_1 can be computed in $O(p)$, which is quite reasonable. Then, we have $O(n^3 p)$ for the innermost loop, $O(n^2)$ calls to `applicable-predicates`, which amounts to $O(n^2 pm)$, and $O(n^2 p)$ for calls of S_1. Extracting the plan is linear in n. Hence, the total runtime of the algorithm is $O(n^2(n+m)p)$

4 Example

In order to illustrate the algorithm we need to fix the functions S_0, S_1, C_0 and C_1. We use the simple cost function C_{out} [CM95, SM96a]. It calculates the total cost of evaluating an expression by summing up the cardinalities of the intermediate results. As a consequence, the array s simply stores cardinalities, and S_0 has to extract the cardinality of a given base relation and multiply it by the selectivities of the applicable predicates. S_1 multiplies the input cardinalities with the selectivities of the applicable predicates. We set C_0 to zero and C_1 to S_1. The former is justified by the fact that every relation must be accessed exactly once and hence, the access costs are equal in all plans. Summarizing, we define

$$\begin{aligned} S_0(R, \mathcal{B}) &:= |R| \prod_{p \in \mathcal{B}} f(p) \\ S_1(x, y, \mathcal{B}) &:= xy \prod_{p \in \mathcal{B}} f(p) \\ C_0(R, \mathcal{B}) &:= 0 \\ C_1(x, y, \mathcal{B}) &:= S_1(x, y, \mathcal{B}) \end{aligned}$$

where $f(p)$ gives us the selectivity of a predicate p.

We illustrate the algorithm by an example consisting of four relations $R_1, \ldots, R_4$ with cardinalities $|R_1| = 200$, $|R_2| = 1$, $|R_3| = 1$, $|R_4| = 20$. Besides, we have three predicates $p_{i,j}$ with $\mathcal{F}(p_{i,j}) \subseteq \mathcal{A}(R_i) \cup \mathcal{A}(R_j)$. They are $p_{1,2}$, $p_{3,4}$, and $p_{1,4}$ with selectivities 1/2, 1/10, 1/5.

Let us first consider an example plan and its costs. The plan

$$((R_1 \bowtie_{p_{1,2}} R_2) \bowtie_{true} R_3) \bowtie_{p_{1,4} \wedge p_{3,4}} R_4$$

has the costs $240 = 100 + 100 + 40$.

For our simple cost function, the algorithm `construct-bushy-tree` will fill the array s with the initial values:

s			
200			
	1		
		1	
			20

After initilization, the array c has 0 everywhere in its diagonal and the array p empty sets.

For $l = 2$, the algorithm produces the following values:

l	i	j	k	s[i,j]	q	current c[i,j]	current t[i,j]
2	1	2	1	100	100	100	1
2	2	3	2	1	1	1	2
2	3	4	3	2	2	2	3

For $l = 3$, the algorithm produces the following values:

l	i	j	k	s[i,j]	q	current c[i,j]	current t[i,j]
3	1	3	1	200	101	101	1
3	1	3	2	200	200	101	1
3	2	4	2	2	4	4	2
3	2	4	3	2	3	3	3

For $l = 4$, the algorithm produces the following values:

l	i	j	k	s[1,4]	q	current c[1,4]	current t[1,4]
4	1	4	1	40	43	43	1
4	1	4	2	40	142	43	1
4	1	4	3	40	141	43	1

where for each k the value of q (in the following table denoted by q_k) is determined as follows:

$$\begin{array}{rcrcrcrcrcr} q_1 & = & c[1,1] & + & c[2,4] & + & 40 & = & 0+3+40 & = & 43 \\ q_2 & = & c[1,2] & + & c[3,4] & + & 40 & = & 100+2+40 & = & 142 \\ q_3 & = & c[1,3] & + & c[4,4] & + & 40 & = & 101+0+40 & = & 141 \end{array}$$

Collecting all the above $t[i,j]$ values leaves us with the following array as input for `extract-plan`:

$i \setminus j$	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

The function `extract-plan` merely calls `extract-subplan`. For the latter, we give the call hierarchy and the result produced:

```
000 extract-plan(..., 1, 4)
100   extract-plan(..., 1, 1)
```

```
200   extract-plan(..., 2, 4)
210    extract-plan(..., 2, 3)
211     extract-plan(..., 2, 2)
212     extract-plan(..., 3, 3)
210    return (R2 ⋈true R3)
220    extract-plan(..., 4, 4)
200   return ((R2 ⋈true R3) ⋈p3,4 R4)
000  return (R1 ⋈p1,2∧p1,4 ((R2 ⋈true R3) ⋈p3,4 R4))
```

The total cost of this plan is $c[1,4] = 43$.

5 Conclusion

We presented the first polynomial algorithm that computes a plan with minimal costs for a given sequence of order-preserving join operators and a set of selections. The plan has minimal costs among those plans that push selections. Hence, ordering joins and selections is not integrated. On the positive side, cross products are considered and plans are not restricted to left-deep trees. Further, no assumptions on the cost function are made. Specifically, we do not need the ASI property, as in the traditional unordered context. Also, the algorithm does not depend on any assumption about the join graph.

If pushing selections as far down as possible does not yield the best plan, our algorithm fails. It remains an open question whether there exists a polynomial algorithm that orders joins and selections at the same time. Note that this question is not only open for order-preserving operations, but also for the traditional case when plan generation is restricted to left-deep trees with no cross products—the other cases have been proven to be NP-hard even if we have no selections.

Acknowledgement. I thank Simone Seeger for her help in preparing the manuscript.

References

[BT99] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 1999.

[CKK98] J. Claussen, A. Kemper, and D. Kossmann. Order-preserving hash joins: Sorting (almost) for free. Technical Report MIP-9810, University of Passau, 1998.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CM95] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 54–67, 1995.

[CS96] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 87–98, 1996.

[CS99] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. on Database Systems*, 24(2):177–228, 1999.

[FHP02] F. Frasincar, G.-J. Houben, and C. Pau. XAL: An algebra for XML query optimization. In *Australasian Database Conference (ADC)*, 2002.

[Hel94] J. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 325–335, 1994.

[HS93] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–277, Washington, DC, 1993.

[IK84] T. Ibaraki and T. Kameda. Optimal nesting for computing n-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 128–137, 1986.

[LS03] A. Lerner and D. Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2003. to appear.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, 1979.

[SJS02] G. Slivinskas, C. Jensen, and R. Snodgrass. Bringing order to query optimization. *SIGMOD Record*, 13(2):5–14, 2002.

[SM96a] W. Scheufele and G. Moerkotte. Constructing optimal bushy trees for join queries is NP-hard. Technical Report 96-11, Universität Mannheim, 1996.

[SM96b] W. Scheufele and G. Moerkotte. Optimal ordering of selections and joins in acyclic queries with expensive predicates. Technical Report 96-3, RWTH-Aachen, 1996.

[YKY+91] K. Yajima, H. Kitagawa, K. Yamaguchi, N. Ohbo, and Y. Fujiwara. Optimization of queries including adt functions. In *International Symposium on Database Systems for Advanced Applications*, pages 366–376, Tokyo, Japan, 1991.