Pipelining in Query Execution *

Annita N. Wilschut Peter M. G. Apers

University of Twente

P.O.Box 217, 7500 AE Enschede, the Netherlands

1 Introduction

Recently, attention has been paid to using *parallelism* to improve the response time of relational database systems (DBMS). Two forms of parallelism are used [4]: *taskspreading* is parallelism between independent subtasks; *pipelining* is parallelism between two subtasks that are in a producer-consumer relation. The use of *taskspreading*, both within individual operations [3] and between operations belonging to a nested query [2] has been studied extensively. This paper investigates whether the use of pipelining between operations can improve the response time of nested query execution in a *main memory* DBMS running on a shared-nothing multi-processor. It is shown that an adjustment of the standard algorithms is needed to efficiently use pipelining. When adjusted algorithms are used, the overall response time of nested queries can be improved. *Equi-join* algorithms are used as an example to illustrate important issues. Similar results can be derived for other relational operations, as well.

Execution model. A dataflow model for query execution [1] is assumed. A nested query can be represented as a tree: the root represents the result, the leaves are operations on base data and other nodes are operations that get their input from child operations and send their output to their parent. The processors of the multi-processor system host *query processors* that execute operations belonging to a query. To execute an operation they first read their input locally or from the network, then they perform some operation on the input and the output can be sent via the network to another query processor. The nodes of a query tree are allocated to query processors and the arcs represent the flow of intermediate results from one operation to another. Pipelining is possible between nodes that are in a parent-child relationship. This pipelining effectively yields parallelism, if some result of the child operation is available in an early stage of its operation, so that the parent operation can start executing.

2 Pipelining algorithms

Hash-join algorithms [3] are used here as example of pipelining algorithms. An adjusted version of the hash-join algorithm (pipelining hash-join) is proposed that is especially designed to gain performance from pipelining in nested queries. A main memory version of the common hash-join algorithm is compared to our pipelining hash-join algorithm. **Common hash-join.** This algorithm first reads one entire operand and builds an in-memory hash-table for it. In the second phase, the tuples of the other operand are read, hashed and compared to the tuples in the corresponding bucket of the other operand. If a match is found an output tuple is produced.

Pipelining hash-join. This algorithm consists of only one phase in which a hash-table for both operands is built. When a tuple arrives, it is hashed and compared to the tuples in the corresponding bucket of the other operand that have already arrived. If a match is found an output-tuple is formed. Finally, the input-tuple is inserted in its own hash-table.

Characteristics. The common hash-join algorithm only starts producing result-tuples in its second phase. The pipelining hash-join on the other hand, has a high probability of finding result tuples in an early stage of the join process, because it does not need an entire operand before it can start producing any output tuples. Therefore the pipelining

hash-join is expected to produce more parallelism via pipelining.

Costs. The pipelining hash-join is slightly more expensive than the common hash-join due to the fact that an extra hash-table has to be maintained.

Example. The figure shows the execution characteristics of both algorithms for an example query: $((\sigma A \bowtie \sigma B) \bowtie (\sigma C \bowtie \sigma D))$. The operands are equal in size, the selections have a selectivity of 10%, and the joins operations match one tuple of one operand to exactly one of the other. The operations are executed on a private processor. Each operation has its own diagram, plotting the percentage of the cpu used by the operation against the time. The figure is produced by a simulation program. The figure shows that the pipelining hash-join has two advantages over the common hash-join. Firstly, the pipelining hash-join does more work in an early stage of the join-process, so that its processor is used more effectively. Consequently, the first join operations can be executed fully in parallel with the selections. Secondly, the join operations do indeed produce output in an early stage, so that the second join operation can start operating shortly after the first joins have started.

3 Conclusion

In this abstract, we have shown that the use of an adjusted algorithm can improve the performance of a nested query execution containing join operations. Apart from the equi-join, the *union*, *cartesian product*, Θ -join, and *intersection* operations can be implemented in a pipelining way. The *difference* cannot be implemented in this way as well. The proposed algorithms are implemented in PRISMA/DB [4]; performance analysis is done. Furthermore, we work on query optimization techniques that can exploit the possibilities of pipelining.

References

- [1] W. Alexander and G. Copeland, *Process and Dataflow Control in Distributed Data-Intensive Systems*, Proc. 1988 SIGMOD, USA, June 1988.
- [2] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.
- [3] D.A. Schneider and D.J. DeWitt, *A performance Evaluation of Four Join Algorithms in a Shared-Nothing Multiprocessor Environment*, Proc. 1989 SIGMOD, USA, June 1989.
- [4] A.N. Wilschut e.a. *Implementing PRISMA/DB in an OOL*, Proc. 6th IWDM, France, June 1989.

*The work reported in this document was conducted as part of the PRISMA project, a joint effort with Philips Research Eindhoven, partially supported by the Dutch "Stimuleringsprojectteam Informaticaonderzoek (SPIN)".