An Experimental Performance Evaluation of Join Algorithms for Parallel Object Databases

Sandra de F. Mendes Sampaio[1], Jim Smith[2], Norman W. Paton[1], and Paul Watson[2]

[1] Department of Computer Science, University of Manchester,
Manchester, M13 9PL, UK
(sampaios, norm)@cs.man.ac.uk
[2] Department of Computing Science, University of Newcastle upon Tyne,
Newcastle, NE1 7RU, UK
(Paul.Watson, Jim.Smith)@newcastle.ac.uk

Abstract. Parallel relational databases have been successful in providing scalable performance for data intensive applications, and much work has been carried out on query processing techniques in such systems. However, although many applications associated with object databases also have stringent performance requirements, there has been much less work investigating parallel object database systems. An important feature for the performance of object databases is the speed at which relationships can be explored. In queries, this depends upon the effectiveness of different join algorithms into which queries that follow relationships can be compiled. This paper presents the results of empirical evaluations of four parallel join algorithms, two value based and two pointer based. The experiments have been run on Polar, a parallel ODMG object database system.

1 Introduction

Applications associated with object databases are demanding in terms of their complexity and performance requirements. However, there has been relatively little work on parallel object databases, and few complete systems have been constructed. There has been still less work on systematic assessment of the performance of query processing in parallel object databases. This paper presents the results of a performance evaluation of different algorithms for exploring relationships in the parallel object database system Polar [9].

The focus in this paper is on the performance of OQL queries over ODMG databases, which are compiled in Polar into a parallel algebra for evaluation. The execution of the algebra on a network of PCs, supports both inter and intra operator parallelism. The evaluation focuses on the performance of three parallel join algorithms, one of which is value based (hash join) and two of which are pointer based (materialize and hash loops). Figures are presented for queries running over the medium OO7 database [2].

The experiments and the resulting performance figures can be seen to serve two purposes: (i) they provide insights on algorithm selection for implementers of parallel object databases; (ii) they provide empirical results against which cost models for parallel query processing can be validated (e.g., [8]).

There is a considerable body of work on pointer based join algorithms. For example, six uni-processor join algorithms are compared in [8], including algorithms based on nested loop, sort merge and hybrid hash. A more recent study [1] both proposes a new pointer based join algorithm and provides a comprehensive comparison with several other approaches. Although [1] contains a few empirical results, the evaluations are based principally on analytical models, and parallelism is not considered.

There have been a few investigations into query evaluation in parallel object database systems. Four hash-based parallel pointer based join algorithms, including hash-loops, are compared in [3]. The comparison, which uses an analytical model, focuses on issues such as the need for extents for both classes participating in a join. Another model-based evaluation is provided by [10], in which their multiwavefront algorithm is compared with a more conventional pointer-based algorithm. There are few empirical evaluations of parallel pointer-based joins, although [4] describes how ParSets can be used to support parallel traversals over object databases for application development rather than query processing.

We see this paper as adding to the results mentioned above by providing comprehensive experimental evaluations over a parallel object database system. The paper is organized as follows: The architecture of the Polar parallel object database is outlined in Section 2. The parallel join algorithms evaluated are described in Section 3. Section 4 describes the experiments in detail, reviewing the OO7 database, and presenting the queries used in the experiments. Section 5 presents results of the experiments and their interpretation. Finally, Section 6 concludes.

2 The Polar System

Polar is a shared-nothing ODMG compliant object database server. For the experiments reported in this paper, it was running on 8 PCs, interconnected by Fast Ethernet. One of these processors serves as a coordinator, running the compiler/optimizer, while the remaining seven serve as object stores, running an object manager and execution engine.

In Polar, OQL queries are compiled into parallel query execution plans (PQEPs) expressed in an object algebra based on that of [5]. All the algebraic operators are implemented as iterators [7]. As such, the operators support three main functions, *open*, *next* and *close*, which define the main interface through which they interact with one another. The implementation of the algebra is essentially sequential, and most of the functionality that relates to parallelism, such as flow control, inter-process communication and data distribution, is encapsulated in the exchange operator, following the operator model of parallelization [6].

Figure 1 illustrates two possible PQEPs for query Q1 in figure 2. The query is over the OO7 benchmark schema. The query retrieves the *id* attributes of *AtomicParts* and the *CompositeParts* of which they are part, where the *id*s satisfy certain conditions. The plan on the left in figure 1 executes the example query using the value-based hash-join, while the plan on the right executes the same query using the pointer-based hash-loops.

In both plans the *CompositePart* objects are obtained from the stores through a seq_scan operator. Subsequently, the tuples containing the *CompositePart* objects have their multiple valued reference attribute *parts* "unnested", i.e., the unnest operator

transforms each input tuple into a number of tuples which corresponds to the number of values in the attribute. The tuples generated by the unnest contain an additional attribute that represents one of the references in the multiple-valued relationship. The apply operators are responsible for performing projections on the input tuples, discarding attributes that are not relevant to the subsequent steps of the execution, and saving communication costs, as smaller tuples are distributed through the network. The exchange operators are responsible for sending tuples from a producer processor to its consumer(s) and for receiving the tuples on the consumer processors using the network. Thus exchange defines a partitioning of PQEPs into sub-plans, as indicated in the figure. The exchange in the boundary between sub-plans 1 and 2 redistributes the input tuples for the join operator according to the join attribute, i.e., the attribute added by unnest. The policy used for tuple distribution is chosen at query compile time, e.g. *select_by_oid* or *round_robin*. The exchange in the boundary between sub-plans 2 and 3 is responsible for directing the intermediate results to a single processor (the coordinator) for building the final result and sending it to the user.

Object identifiers in Polar contain a volume identifier, and a logical identifier within the volume. Tables are maintained that allow the node of a volume to be identified from the volume id, and to allow the page of an object to be identified from its logical identifier.

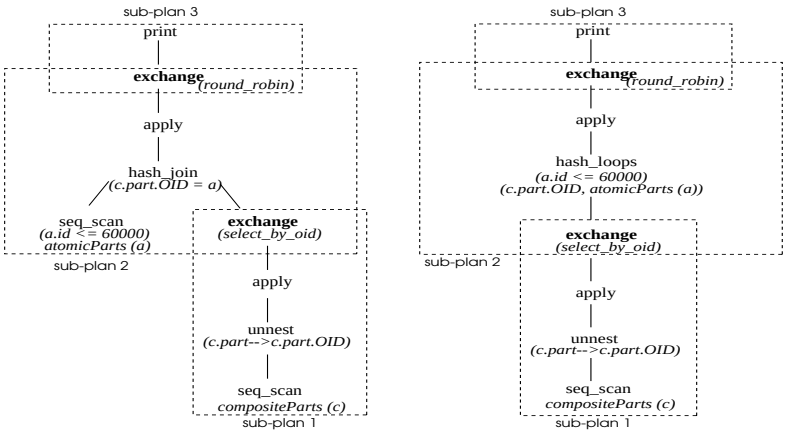

Fig. 1. On the left hand side: Possible PQEP, based on hash-join, for example query Q1. On the right hand side: Possible PQEP, for hash-loops, for example query Q2.

3 Algorithms

This section describes the design of the three join algorithms evaluated in Polar.

Hash-join: The Polar version of hash-join is a one-pass implementation of the relational hash-join implemented as an iterator. This algorithm hashes the tuples of the smallest

input on their join attribute(s), and places each tuple into a main memory hash table. Subsequently, it uses the tuples of the largest input to probe the hash table using the same hash function, and tests whether the tuple and the tuples that have the same result for the hash function satisfy the join condition.

Materialize: The materialize operator is the simplest pointer-based join, which performs naive pointer chasing. It iterates over its input tuples, and for each tuple reads an object, the OID of which is an attribute of the tuple. Dereferencing the OID has the effect of following the relationship represented by the OID-valued attribute. Unlike the hash-join described previously, materialize does not retain (potentially large) intermediate data structures in memory, since the only input to materialize does not need to be held onto by the operator after the related object has been retrieved from the store. The pages of the related objects retrieved from the store may be cached for some time, but the overall space overhead of materialize is small.

Hash-loops: The hash-loops operator is an adaptation for the iterator model of the pointer-based hash-loops join proposed in [3]. The main idea behind hash-loops is to minimize the number of repeated accesses to disk pages without retaining large amounts of data in memory. The first of these conflicting goals is addressed by collecting together repeated references to the same disk pages, so that all such references can be satisfied by a single access. The second goal is addressed by allowing the algorithm to consume its input in chunks, rather than all at once. Thus, hash-loops may fill and empty a main memory hash table multiple times to avoid keeping all of the input tuples in memory at the same time. Once the hash-table is filled with *window_size* tuples, each bucket in the hash table is scanned in turn, and its contents matched with objects retrieved from the store. Since the tuples in the hash table are hashed on the page number of the objects specified in the inter-object relationship, each disk page is retrieved from the store only once within each window. Once all the tuples that reference objects on a particular page have been processed, the corresponding bucket is removed from the hash table, and the next page, which corresponds to the next bucket to be probed, is retrieved from the store. Thus, hash-loops seeks to improve on materialize by coordinating accesses to persistent objects, which are likely to suffer from poor locality of reference in materialize.

4 The Experiments

OO7 Database The OO7 Benchmark [2] provides three different sizes for its database: small, medium and large. The differences in size are reflected in the cardinalities of extents and inter-object relationships. The following table shows the cardinalities of the extents and relationships used in the experiments for the medium OO7 database, which is used here.

Extent	Cardinality	Cardinality of Relationships
AtomicParts	100,000	partOf: 1
CompositeParts	500	parts: 200, documentation 1
Documents	500	
BaseAssemblies	729	componentsPriv: 3

To give an indication of the sizes of the persistent representations of the objects involved in OO7, the following are the sizes of individual objects obtained by measuring the collections stored for the medium database: AtomicPart – 190 bytes; CompositePart – 2,761 bytes; BaseAssembly – 190 bytes; Document – 24,776 bytes.

Experiment Queries The benchmark queries, Q1 to Q4, are given in Figure 2. Q1 and Q2 explore single-valued relationships, whereas Q3 and Q4 explore multiple-valued relationships. The predicate in the `where` clauses in Q1 and Q3 are used to vary the selectivity of the queries over the objects of the input extents, which may affect the join operators in different ways. The selectivities are varied to retain 100%, 10%, 1%, and 0.1% of the input extents.

```
Q1:
select struct(A:a.id,
       B:a.partOf.id)
from a in AtomicParts
where a.id <= v1
 and a.partOf.id <= v2;

Q2:
select struct(A:a.id,B:a.docId,
 C:a.partOf.documentation.id)
from a in AtomicParts
where a.docId !=
  a.partOf.documentation.id;

Q3:
select struct(A:c.id,B:a.id)
from c in Compos-
iteParts,
 a in c.parts
where c.id <= v1
  and a.id <= v2;

Q4:
select struct(A:b.id,B:c.id)
from b in BaseAssem-
blies,
  a in b.componentsPriv,
  c in a.parts
where b.buildDate<c.buildDate;
```

Fig. 2. OQL expressions for the experiment queries.

4.1 Experiment Context

The environment used in the experiments is a cluster of 233MHz Pentium II PCs running RedHat Linux version 6.2, each with 64MB main memory and a number of local disks, connected via a 100Mbps Fast ethernet hub. For each experiment, data is partitioned in "round robin" style over some number of disks, of which there is one at each node, each being a MAXTOR MXT-540SL. All timings are based on cold runs of the queries, with the server shut down and the operating system cache flushed between runs. In each case, the experiments were run three times, and the fastest time obtained is reported.

Several of the algorithms have tuning parameters that can have a significant impact on the way they perform (e.g., the hash table sizes for hash-join and hash-loops). In all cases, we have tried to select values for these parameters that will allow the algorithms to perform at their best. In hash-join, the hash table size is set differently for each join, to the value of the first prime number after the number of buckets to be stored in the hash table by the join. This means that there should be few clashes during hash table

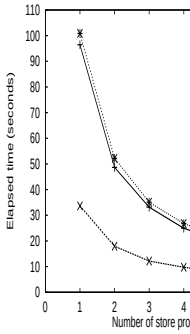

Fig. 3. Elapsed time for Q1.

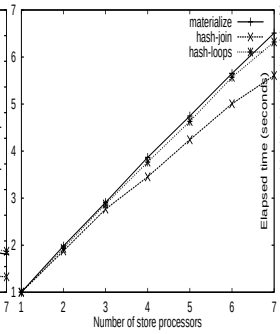

Fig. 4. Speedup graph for Q1.

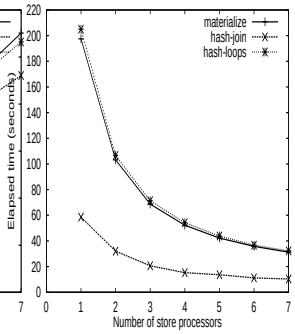

Fig. 5. Elapsed time for Q2.

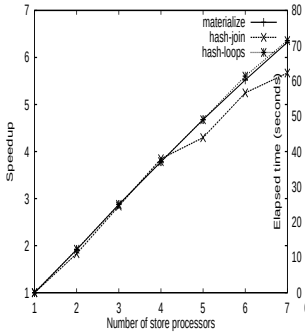

Fig. 6. Speedup graph for Q2.

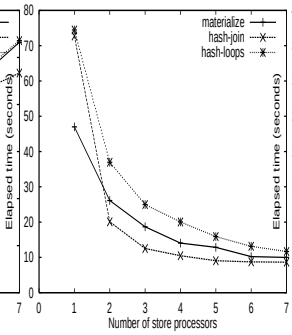

Fig. 7. Elapsed time for Q3.

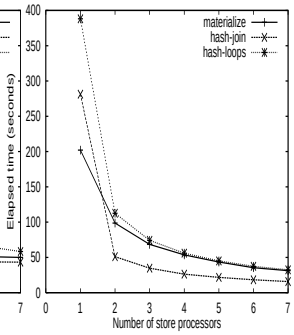

Fig. 8. Elapsed time for Q4.

construction, but that the hash table does not occupy excessive amounts of memory. In hash-loops, the hash table size is also set differently for each join, to the value of the first prime number after the number of pages occupied by the extent that is being navigated to. This means that there should be few clashes during hash table construction, but that the hash table does not occupy an excessive amount of memory. The other parameter for hash-loops is the window size, which is set to the size of the input collection, except where otherwise stated. This decision minimizes the number of page accesses carried out by hash-loops, at the expense of some additional hash table size. None of the experiments use indexes, although the use of explicit relationships with stored OIDs can be seen as analogous to indexes on join attributes in relational databases.

5 Results

5.1 Following Path Expressions

Path expressions, whereby one or several single-valued relationships are followed in turn, are common in OQL queries. Test queries Q1 and Q2 contain path expressions,

of lengths one and two, respectively[1]. Response times and speedup results for these queries using 100% selectivity are given in figures 3 to 6.

The graphs illustrate that all three algorithms show near linear speedup, but that hash-join is significantly quicker throughout. The difference in response times between hash-join and the pointer based joins is explained with reference to Q1. There are three possible explanations for the better performance of hash-join:

1. Hash join retrieves the instances of the two extents (*AtomicParts* and *CompositeParts*) by scanning. By contrast, the pointer based joins scan the *AtomicParts* extent, and then retrieve the related instances of *CompositeParts* as a result of dereferencing the *partOf* attribute on each of the *AtomicPart*s. This leads to essentially random reads from the extent of *CompositePart* (until such time as the entire extent is stored in the cache), and thus to potentially greater IO costs for the pointer based joins. However, based on the published seek times for the disks on the machine (an average of around 8.5ms and a maximum of 20ms), the additional time spent on seeks into the *CompositePart* extent should not be significantly more than 1s on a single processor.
2. The navigational joins all perform mappings from a logical OID to the physical disk page where an object is stored. If this logical to physical mapping is slow, that would further increase the cost of obtaining access to a specific object within an extent, a feature that is not required by the value-based joins.
3. When an object has been read in from disk, it undergoes a mapping from its disk based format into the nested tuple structure used for intermediate data by the evaluator. Because each *CompositePart* is associated with many *AtomicPart*s, the pointer based joins perform the *CompositePart* $\rightarrow$ *tuple* mapping once for every *AtomicPart* (i.e., 100,000 times), whereas this mapping is carried out only once for each *CompositePart* when the extent is scanned for the hash-join (i.e., 500 times).

An additional experiment was carried out to identify the cost of the *CompositePart* $\rightarrow$ *tuple* mapping, which shows that around 50% of the time taken to evaluate Q1 for materialize (and thus also hash-loops) on one processor is spent on the mapping. A further test showed that the essentially random fetching of the pages from the *CompositeParts* extent occupied around 20% of the time taken to evaluate Q1 for materialize. We will seek to improve the implementation of the mapping function in future versions of Polar. We note that if reducing the mapping cost significantly proves difficult, it will be easier to modify hash-loops than materialize to reduce the numbers of mappings performed, as hash-loops performs more coordinated access to objects within pages.

We note the very similar performance of materialize and hash-loops. Hash-loops seeks to improve on materialize by reducing the number of page reads, through coordinating accesses to related pages. However, this shows no benefit in the example queries, because the complete *CompositeParts* and *Documents* extents can be cached, and thus the potential costs associated with uncoordinated accessing of the disk by materialize have been mitigated.

[1] We define the length of a path expression to be the number of joins required to evaluate it.

5.2 Following Multiple-Valued Relationships

Multiple valued relationships are expressed in OQL by iterating over collections that represent relationships in the from clause. Polar uses the same join algorithms for evaluating such relationships as for following path expressions. Test queries Q3 and Q4 follow one and two multiple valued relationships respectively. Response times for these queries using 100% selectivity are given in figures 7 and 8.

A striking feature of the figures for both Q3 and Q4 is the superlinear speedup for both hash-join and hash-loops, especially in moving from 1 to 2 processors. Both hash-join and hash-loops have significant space overheads associated with their hash tables, which causes swapping during evaluation of these algorithms in the configurations with smaller numbers of nodes. Monitoring swapping on the different nodes shows that by the time the hash tables are split over 3 nodes they fit in memory, and thus the effect of swapping on performance is removed for the larger configurations.

Another noteworthy feature is the fact that the relative performance of materialize and hash loops compared with hash join is better in Q3 and Q4 than in Q1 and Q2. This can be explained with reference to Q3. In Q3, the total number of *CompositePart* $\rightarrow$ *tuple* and *AtomicPart* $\rightarrow$ *tuple* mappings is the same for all the join algorithms (in contrast with the situation in Q1 and Q2). Thus the overhead that resulted from repeatedly mapping the same object does not affect materialize or hash loops in Q3.

As mentioned in Section 3, it is possible to reduce the space overhead associated with the hash table in hash loops by constructing several smaller hash tables over parts of the input extent, rather than one hash table over the whole extent. To test the effect of this on the performance of hash loops, figures 9 and 10 show results for different values of the window size parameter of hash-loops, for Q2 and Q4. Note from figure 9 that for Q2 the variation in the window size did not affect the performance. However, for Q4 (figure 10) there is a reduction in the elapsed times for smaller numbers of processors, as a consequence of the reduction of the swapping activity in those configurations.

5.3 Varying Selectivity

The selectivity experiments involve applying predicates to the inputs of Q1 and Q3, each of which carries out a single join. Response times for these queries over the medium OO7 database running on 6 nodes, varying the values for *v1* and *v2* in the queries, are given in figures 11 and 12, and figures 13 and 14, respectively. Note that what is being varied here is the selectivities of the scans of the collections being joined, not the *join selectivity* itself, which is ratio of the number of tuples returned by a join to the size of the cartesian product.

The experiments measure the effect of varying the selectivity of the scans on the inputs to the join thus:

1. *Varying the selectivity of the outer collection (v1):* The outer collection is used to probe the hash table in the hash-join, and is navigated from in the pointer-based joins. The effects of reducing selectivities are as follows:
   hash-join: The number of times the hash table is probed and the amount of network traffic is reduced, although the number of objects read from disk and the size of the

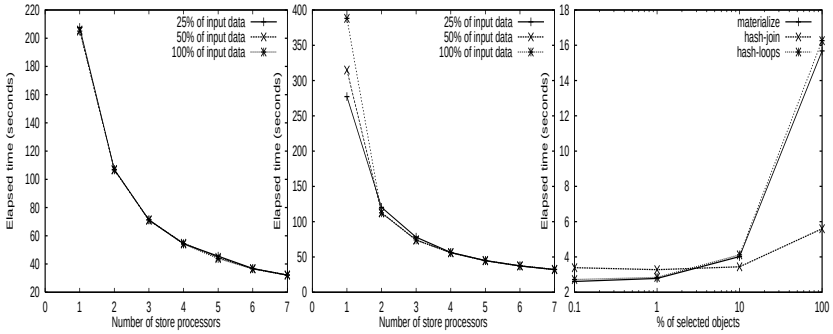

Fig. 9. Elapsed times for Q2, varying the window size of hash-loops.

Fig. 10. Elapsed times for Q4, varying the window size of hash-loops.

Fig. 11. Elapsed time for Q1, varying predicate selectivity using *v1*.

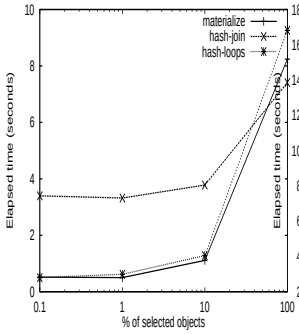

Fig. 12. Elapsed time for Q3, varying predicate selectivity using *v1*.

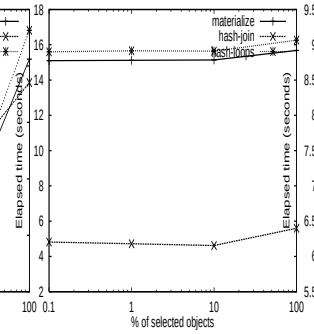

Fig. 13. Elapsed times for Q1, varying predicate selectivity using *v2*.

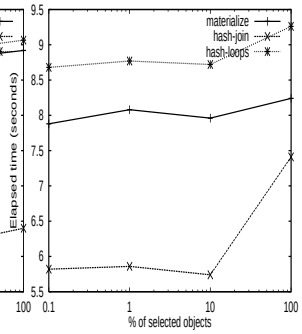

Fig. 14. Elapsed times for Q3, varying predicate selectivity using *v2*.

hash table remain constant. In Q1, the times reduce to some extent with drops in selectivity, but not all that much, so we can conclude that neither network delays nor hash table probing make substantial contributions to the time taken to evaluate the hash-join version of Q1. As the reduction in network traffic and in hash table probes is similar for Q1 and Q3, it seems unlikely that these factors can explain the somewhat more substantial change in the performance of Q3. The only significant feature of Q3 that does not have a counterpart in Q1 is the unnesting of the *parts* attribute of *CompositeParts*. The unnest operator creates a large number of new intermediate tuples in Q3 (100,000 in the case of 100% selectivity), so we postulate that much of the benefit observed from reduced selectivity in Q3 results from the smaller number of collections to be unnested.

pointer-based joins: The number of objects from which navigation takes place reduces in line with the selectivity, so reducing the selectivity of the outer collection significantly reduces the amount of work being done. As a result, changing the se-

lectivity of the scan on the outer collection has a substantial impact on the response times for the pointer-based joins in the experiments. As many real queries will have low selectivities, there should be many cases in which pointer based joins will give better performance than value based joins in which full extents are scanned.

2. *Varying the selectivity of the inner collection (v2):* The inner collection is used to populate the hash table in hash-join and to filter the results obtained after navigation in the pointer-based joins. The effects of reducing selectivities are as follows:
   hash-join: The number of entries inserted into the hash table reduces, as does the size of the hash table, although the number of objects read from disk and the number of times the hash table is probed remains the same. In Q1 there are at most 500 entries in the hash table (one for each *CompositePart*), which is reduced to 100 for the lowest selectivity. However, in Q3 the corresponding figures are 100,000 (one for each *AtomicPart*) and 20,000, so the reductions in hash table inserts and in hash table size are much more substantial for Q3 than for Q1, which could explain the pattern observed in figures 13 and 14.
   pointer-based joins: The amount of work done by the navigational joins is unaffected by the addition of the filter on the result of the join. As a result, changing the selectivity of the scan on the inner collection has a modest impact on the response times for the pointer-based joins in the experiments.

6 Conclusions

The paper has presented results on the evaluation of queries over a parallel object database server. To the best of our knowledge, the results reported are the most comprehensive to date on the experimental evaluation of navigational joins, and on the performance of parallel object databases in a shared-nothing environment.

To draw very general conclusions from specific experiments would be injudicious, but the results reported certainly demonstrate that the join algorithm of choice depends on many factors, including the sizes of the collections participating in the join, the cardinalities of the relationships explored, the selectivities and locations of predicates on inputs to nodes, and the amount of memory available.

What use are these experiments to others? We hope that the results could be particularly useful to three groups of people: developers of object database systems, who must select appropriate algorithms for inclusion in their systems; researchers working on analytical models of database performance, who stand to benefit from the availability of experimental results against which models can be validated; and the developers of query optimizers, who have to design cost models or heuristics that select physical operators based on evidence of how such operators perform in practice.

References

1. R. Braumandl, J. Claussen, A. Kemper, and D. Kossmann. Functional-join processing. *VLDB Journal*, 8(3+4):156–177, 2000.
2. M. Carey, D.J. DeWitt, and J.F. Naughton. The OO7 benchmark. In *ACM SIGMOD*, pages 12–21, 1993.

3. D. J. DeWitt, D. F. Lieuwen, and M. Mehta. Pointer-based join techniques for object-oriented databases. In *Proc. of the 2nd Int. Conference on Parallel and Distributed Information Systems (PDIS)*, pages 172–181. IEEE-CS, 1993.
4. D. J. DeWitt, J. F. Naughton, J. C. Shafer, and S. Venkataraman. Parallelising OODBMS traversals: A performance evaluation. *VLDB Journal*, 5(1):3–18, January 1996.
5. L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, December 2000.
6. G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *ACM SIGMOD*, pages 102–111, 1990.
7. G. Graefe. Iterators, schedulers, and distributed-memory parallelism. *Software-Practice and Experience*, 26(4):427–452, April 1996.
8. E. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *ACM SIGMOD*, pages 300–311, Atlantic City, NJ, May 1990.
9. J. Smith, S. F. M. Sampaio, P. Watson, and N. W. Paton. Polar: An architecture for a parallel ODMG compliant object database. In *Conference on Information and Knowledge Management (CIKM)*, pages 352–359. ACM press, 2000.
10. S. Y. W. Su, S. Ranka, and X. He. Performance analysis of parallel query processing algorithms for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(6):979–997, December 2000.