On Parallel Join Processing in Object-Relational Database Systems

Holger Märtens[*], Erhard Rahm

Universität Leipzig, Institut für Informatik, Postfach 920, D–04009 Leipzig, Germany
{maertens|rahm}@informatik.uni-leipzig.de
http://dbs.uni-leipzig.de

Abstract. So far only few performance studies on parallel object-relational database systems are available. In particular, the relative performance of relational vs. reference-based join processing in a parallel environment has not been investigated sufficiently. We present a performance study based on the *BUCKY* benchmark to compare parallel join processing using reference attributes with relational hash- and merge-join algorithms. In addition, we propose a data allocation scheme especially suited for object hierarchies and set-valued attributes.

1 Introduction

Object-relational database systems (ORDBS) [18] are gaining importance as OR features are covered in the latest SQL standard and more and more vendors include OR capabilities in their DBS products. The latter comprise a number of *parallel database systems* (PDBS) suited for large-scale, enterprise databases. Despite these advances in practical application, only few studies have investigated the performance of parallel query processing in ORDBS. For instance, there are different views to what extent relational data allocation and processing methods can be used and whether reference-based navigation and query processing results in a high communication overhead and disk contention [6, 14].

In this paper, we investigate these trade-offs for parallel join processing by comparing the performance of relational hash and sort-merge joins with object-oriented joins based on *reference-valued attributes.* Broadly speaking, relational joins usually offer efficient, set-oriented access to base relations but tend to read a superset of the required data. In contrast, reference-based *assembly* strategies can minimize tuple access by following only relevant pointers but may yield inefficient I/O (high disk contention) and/or high communication overhead. Note that object references are not limited to representing explicit object relationships (*associations*) but are also useful for implementing detached (often set-valued or just very large) attributes and *is-a* relationships within generalization hierarchies [19, 11]. Aggregating along these references thus corresponds to the processing of *implicit joins.*

[*] Supported by the Deutsche Forschungsgemeinschaft (http://www.dfg.de) under grant Ra 497/10.

Our study focuses on *shared-disk* PDBS which typically provide a good combination of scalability and load balancing [15]. Here, reference-based joins (assembly methods) do not result in extra messages since each processing node can directly access all disks. On the other hand, irregular data access patterns may lead to high disk contention depending on the data allocation and query characteristics. Relational join processing, in turn, may itself require a high communication overhead for data redistribution. Our performance study is based on simulation and uses database characteristics from the *BUCKY* benchmark featuring large sets of objects in comparatively flat and uniform structures. We consider such characteristics typical of business applications for which the OR approach is of increasing interest, owing primarily to its convenience in modeling compared to the established relational model. This is in contrast to previous studies on reference-based join processing assuming complex object structures, e.g., for CAD processing (as exemplified in the *OO7* benchmark).

The remainder of this short paper is organized as follows: Sect. 2 briefly mentions some related work from the literature. Sect. 3 presents the different join methods applied in our study and proposes a data allocation scheme supporting efficient reference navigation and parallel query processing. Preliminary simulation results are detailed in Sect. 4 before we conclude.

2 Related Work

We are not aware of any previous work on join processing specifically addressing OR systems. There are, however, a number of relevant studies on relational DBS as well as object-oriented and other 'complex-object' data models where the basic problem of object assembly has been discussed extensively.

For OODBS, a comparison of pointer-based vs. value-based join techniques showed that pointer-based variants of sort-merge and hash joins are superior to naive object assembly methods that track individual pointers one-at-a-time, often leading to inefficient I/O [16]. It was suggested to augment the latter by collecting and ordering references before following them, thus optimizing disk access [13]. These findings were confirmed by other authors, who also proposed new solutions specially geared towards (nested) sets of references [1, 20]. All these studies, however, were limited to sequential processing, so we cannot simply extrapolate from their results.

For the parallel case, pointer-based hash, sort-merge, and nested-loop joins were modeled analytically, but not compared directly [3]. Also, new algorithms were presented without comparison to existing solutions [4, 7].

A cost model exists to estimate page I/O during parallel navigation in complex objects, but it is restricted to a single partition of a single object type [8]. A related model using abstract 'processing costs' indicates that assembly techniques may be preferable to explicit joins in parallel environments due to high flexibility and better load balancing [9].

Finally, there is an enormous body of research on joins in relational PDBS that cannot be detailed here for lack of space. However, we will build on the fact that hash join is usually the most efficient technique [10]. An obvious exception occurs if both inputs are already sorted on the join attribute, in which case a merge join (without a separate

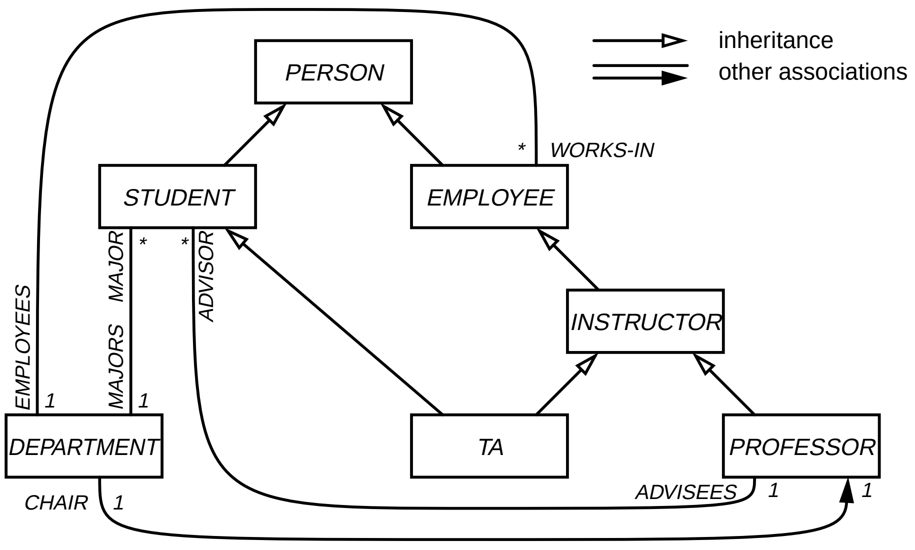

Fig. 1. BUCKY database schema (excerpt)

sorting step) is preferable. This situation has not been considered in any of the complex-object studies cited above.

Note also that most of the work mentioned is concerned with very complex composite-object graphs such as the *OO7* benchmark. This approach differs considerably from the more business-oriented scenarios we have in mind, which may lend themselves more easily to parallel processing. Other aspects not addressed sufficiently in the aforementioned articles include: joins across class hierarchies, disk contention and message overhead in parallel environments, as well as issues of data allocation and load balancing.

3 Data Allocation and Query Processing

This section discusses the allocation and processing approaches used in our simulation study, which is based on the *BUCKY* benchmark proposal [5]. *BUCKY* defines a university scenario including students, professors, courses etc. that has all the characteristics required for our study. In particular, there are class hierarchies expressing *is-a* relationships (e.g., *PERSON-STUDENT*, *PERSON-EMPLOYEE-INSTRUCTOR-PROFESSOR*), set-valued attributes (e.g., *KIDS* of *PERSON*s) and other associations (e.g., *STUDENT*s advised by *PROFESSOR*s and enrolled in *COURSE*s). A subset of the model is shown in Fig. 1.

3.1 Data Allocation

Data allocation in a parallel environment is a very complex issue even in relational DBMSs, and it is compounded by the special properties of the OR model such as non-atomic or reference-typed attributes and class hierarchies. The three most fundamental decisions discussed in the following concern the (de)clustering of object components,

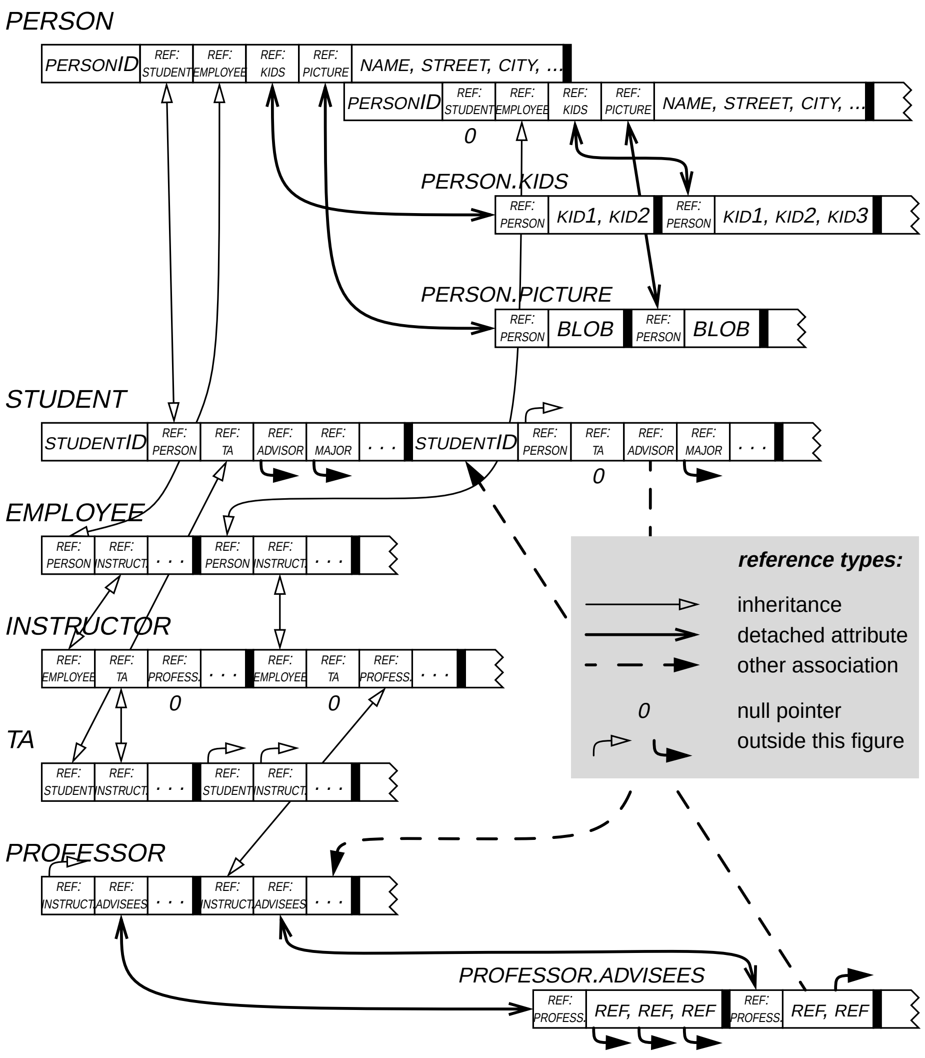

Fig. 2. Storage model of the *BUCKY* database (excerpt)

the fragmentation of relations, and the allocation of fragments to disks. While these parameters can be optimized for any given query type, mixed workloads will usually also contain some poorly supported queries, which will be considered in our study.

To implement the *BUCKY* schema for our study, we chose a declustered storage strategy based on references as illustrated in Fig. 2. Class hierarchies are decomposed such that all attributes materialize in the relation where they first occur. For instance, all *PERSON* attributes are stored in the *PERSON* relation but not in *EMPLOYEE*, *STUDENT* etc. Consequently, each object of a derived class is represented by a tuple in the associated relation but also has tuples in all its base relations. (For simplicity, we treat object classes and the relations implementing them as synonymous.) All these tuples are connected using bidirectional references, e. g., between associated *INSTRUCTOR* and *PROFESSOR* entries. Similarly, large or set-valued attributes such as *PERSON.PICTURE* and *PERSON.KIDS* are detached from the relations they belong to and stored separately; this also applies to sets of references as in *PROFESSOR.ADVISEES*. Such storage strategies have been found superior for queries addressing only a subset of a class's properties [19, 12]. In contrast, we will consider other query types that necessitate implicit join operations across the required class components.

Furthermore, we use a fragmentation scheme in which derived relations are stored analogously to their base relations. For instance, assuming that *PERSON* is range-partitioned based on the *PERSONID* attribute, *STUDENT*, *EMPLOYEE* etc. are all fragmented on the same (inherited) attribute, even though it is not materialized in the derived relations themselves. Thus, pairs of fragments can be identified for implicit joins across hierarchies that can be joined without involving other fragments. Moreover, we propose to keep all relations in the hierarchy sorted on the fragmentation attribute to enable efficient, ordered access within fragments especially for merge joins (cf. Sect. 3.2). Alternatively, clustered indices on *PERSONID* might be used for all relations to avoid physical sorting.

Similarly, detached attributes are fragmented in the same fashion as the tuples they belong to. All *datasets* (i. e., relations or detached attributes) are allocated to all disks (full declustering). For each dataset, we create far more fragments than disks to provide a smaller granule for load balancing (cf. next section) and allocate several of them on each device. We alleviate potential skew in the distribution of references by changing the allocation pattern between datasets. In addition, we make sure that corresponding fragments from different datasets never reside on the same device, enabling parallel I/O, for instance, between a *PERSON* fragment and its associated *STUDENT*s.

Orthogonal to the allocation, arbitrary indices may be defined on the datasets. We merely assume a clustered index on *PERSON.PERSONID* for some of our experiments.

Note that, while developed for a shared-disk architecture, the same allocation can be used in shared-everything environments. For shared-nothing, we advise to first distribute the data across the processors such that related tuples reside on the same node, then apply our method locally if a node has multiple disks.

3.2 Query Processing

The basis of processing is a *scan* operator that is parallelized based on the fragmentation of the data. Thus, a *local scan* (executed by a single processor) works exclusively on a single fragment of a dataset. On top of this scan method, we consider the following three join algorithms; the first two are well known from relational DBMSs whereas the third is borrowed from the object-oriented model:

- The *hash join* first scans the inner dataset to build hash tables on all processors, then probes the outer dataset against the hash tables. This may imply full redistribution of one or both inputs unless their fragmentation attributes are identical to the join attribute.
- The *merge join* scans both datasets simultaneously such that associated fragments are scanned by the same processor and joins are executed locally. This algorithm

depends on its input being sorted on the join attribute and cannot be used for unordered datasets. It can join more than two datasets at a time.

- The *deref(erencing) join* scans one dataset and follows its references to the other dataset one-at-a-time. If more than two datasets are involved, this process is nested in a depth-first manner. An explicit scan is performed only on the first dataset; access to other datasets is limited to relevant tuples referenced from the first.

4 Simulation Setup and Results

We investigate the different join methods within the *SIMPAD* simulation system that has been used successfully in previous studies [17]. *SIMPAD* (*Sim*ulation of *Pa*rallel *D*atabases) is based on the *CSIM/C++* simulation library and provides a full hardware environment with processors, disks, and networks (modeled as server-type facilities) as well as lock and buffer management services. We have developed an object-relational extension on top of the existing relational model, adding an OR database schema as well as query generation and processing modules [2].

For our experiments, we use a simulated shared-disk environment of 5 processing nodes and 20 disks. Each node has 2000 pages of main memory. Scaling the original *BUCKY* schema by a factor of 10, the sizes of the relevant datasets are as follows:

PERSON	1000000 tuples	17860 pages
PERSON.KIDS	3000000 tuples	15960 pages
STUDENT	500000 tuples	3420 pages
PROFESSOR	250000 tuples	1900 pages

For each query, a *coordinator* node is selected that controls its execution and collects the final result. All processing is fully parallelized if a sufficient number of fragments is involved. An adjustable number of *jobs* (local scans or joins) can run concurrently on the same processor. Disk contention is minimized by estimating the I/O required per job and disk, then scheduling together those jobs that show minimal overlap. More details on the implementation and its parameters can be found in [2].

We have evaluated the performance of the three join methods for three common cases of implicit, reference-based joins: hierarchies, detached attributes, and arbitrary associations. We expect the following factors to become manifest in the simulation results:

- The merge join will access both datasets simultaneously, whereas the hash join must do so sequentially, which is probably less efficient. The deref join follows an intermediate strategy by interleaving access to both inputs.
- The merge- and hash-join methods read full data fragments from start to end. Using prefetching, this is very efficient and produces little disk contention but may include tuples irrelevant to the query. The deref join will – by definition – access only required tuples, at least on the ‘target’ side of a reference, but may be inefficient and contention-prone especially for unordered access.

- The hash join requires a certain amount of main memory for its hash tables, whereas its competitors can allocate all storage to their I/O buffers. The deref join may suffer from low hit rates in a small buffer due to its potentially random access pattern, causing multiple I/O on the same data. This is no problem for hash and merge joins reading sequentially.
- The hash join incurs an additional communication overhead when data redistribution is required.

4.1 Hierarchy References

Our first series of experiments targets hierarchy references between the *PERSON* and *STUDENT* relations. Due to our storage model, queries referring to both personal and student-specific information of a student will have to access both relations and use the references between the two tuples in order to assemble the original object.

Since both relations are sorted and partitioned on the same key (*PERSONID*), we can employ the merge-join technique from the relational model, using the references as a join attribute. The hash join can be ruled out as inferior (cf. Sect. 2). As a comparison, we use the object-oriented deref join. Since references exist for both *PERSON* to *STUDENT* and *STUDENT* to *PERSON*, this requires us to define the 'direction' in which to process the query. We make this decision depending on the selection predicate so as to start evaluation with the relation that undergoes the stricter selection. No such decision is necessary for the merge-join operator, which reads both relations (and applies all given selections) simultaneously.

***PERSON* to *STUDENT*.** Consider the query

Find all students with PERSONID in range R.

Fig. 3a shows the comparison of merge join to deref join for this query where the selection range *R* was varied to achieve different selectivities. Since the selection takes place on a *PERSON* attribute, a deref join has to read *PERSON* tuples and follow all non-zero references to *STUDENT* (i.e., for all persons that are students). Using the clustered index on *PERSON.PERSONID*, both algorithms can restrict access to *PERSON* to the absolute minimum. For the deref operator, it follows that only relevant *STUDENT* tuples are accessed as well. The merge join, by contrast, must always process full fragments of *STUDENT* but can exclude those for which the associated *PERSON* fragment is found to be irrelevant.

The graph shows for both algorithms the same basic development of response times in relation to selectivity, owing to their similar access patterns. But the merge join is generally faster due to its efficiency in sequential disks access. Deref join cannot compete here as it must interleave access to both relations and can initiate I/O to the next page only after evaluating a reference. At best, it can be on par with merge join for low selectivity values, where it avoids some unnecessary I/O. Note that optimizing the deref join's order of disk access as discussed in Sect. 2 is useless since references are already sorted.

***STUDENT* to *PERSON*.** We now vary the query to

Find all students with STUDENTID in range R.

Selection now takes place on a *STUDENT* attribute that can no longer be supported by a clustered index (because both relations are fragmented by *PERSONID*, which is unrelated

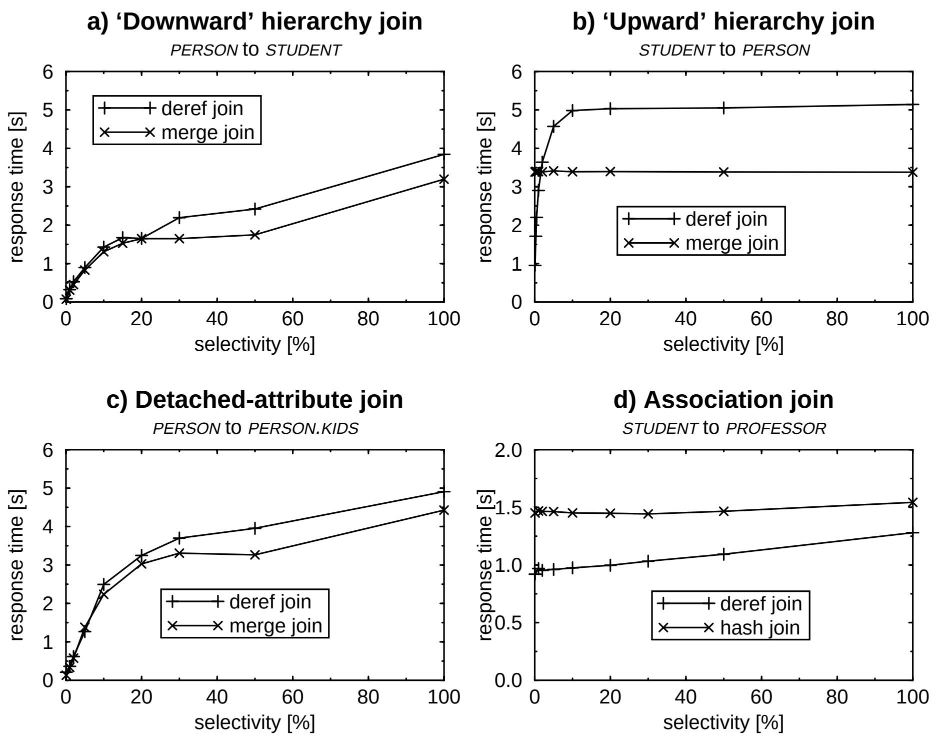

Fig. 3. Simulation results.

to *STUDENTID*). In fact, we assume no index at all although a non-clustered one might be of use for very small selectivity factors. As Fig. 3b shows, execution times are now almost constant for the merge-join operator, which has to perform full scans on both relations in all cases. The deref algorithm, on the other hand, performs a full scan only on *STUDENT* (which is smaller than *PERSON* by a factor of 6) and must access only those pages of *PERSON* that contain relevant tuples. As a consequence, response times – which are otherwise higher than for merge joins as in the previous case – drop sharply for low percentages of selectivity, with the break-even point around 2 %. Below this point, deref join is more efficient than merge join.

4.2 Detached Attributes

Next, we examine the implicit join between a relation and one of its detached attributes, using the query

Find all persons with PERSONID in range R and their children,

which involves relation *PERSON* and its set-valued, detached attribute *KIDS*. Since we assume detached attributes to be ordered and partitioned in the same way as the relations they belong to, we compare only the merge-join and deref-join methods as in Sect. 4.1. Basic circumstances being the same, we expect similar results in the simulations.

Fig. 3c confirms this assumption, showing only one direction of execution due to space limitations. The clustered index on *PERSON* is used as for the *PERSON*-to-*STUDENT* query above. The response time difference is about the same but total response time is higher than in the previous case because the *KIDS* dataset is larger than *STUDENT*.

4.3 Other Associations

The final series of simulations presented here considers reference-based joins on arbitrary associations. Our sample query is

Find all professors advising students with STUDENTID in range R.

This query must access the *STUDENT* and *PROFESSOR* relations. Both are part of the *PERSON* hierarchy and fragmented accordingly, and there is no correlation between person IDs of professors and those of the students they advise. Hence, we cannot assume the order and partitioning of *STUDENT* and *PROFESSOR* to match, making the merge-join algorithm unusable. We will use a hash join instead and compare it to the deref join. The latter will scan the *STUDENT* relation first to perform the selection on *STUDENTID* early, and additionally read the *PROFESSOR* tuples referenced by selected students. The hash join will redistribute both relations based on the join key, also beginning with *STUDENT*. As both inputs are rather small, we have reduced the buffers to 500 pages per node.

Fig. 3d shows the results of our experiments. We find that hash join is inferior for all selectivities due to its message passing overhead and its need to scan both relations in full. (Note that a sort-merge join would face the same problems.) In addition, it can read only one dataset at a time whereas deref join can interleave access to both and also save some I/O on *PROFESSOR*. Disk contention appears to be no problem for the deref join even though its access pattern on *PROFESSOR* is highly irregular. We ascribe this to our allocation scheme that enables effective load balancing on the disks of the system. This aspect, however, must be validated for a wider range of parameters as the unordered access to *PROFESSOR* raises concerns of buffer thrashing that have yet to be studied thoroughly. Note, though, that in contrast to the previous cases, an optimization of disk access is likely to increase the deref join's advantage over relational methods.

5 Conclusions and Future Work

Although our study is still in an early stage, preliminary results are quite interesting. We found that, as in relational PDBS, merge join is the preferred method when both inputs are fragmented and ordered on the join attribute. In this case, the deref join can compete only for low selectivity factors on non-indexed attributes, where it requires less total I/O than the merge operator. When inputs are ordered differently, however, the deref operator beats relational joins by saving some communication overhead while avoiding disk contention owing to our data allocation. Due to its efficacy, we propose our fragmentation and allocation scheme as a basis of future work.

Further research on parallel OR query processing will have to include the following aspects: First, our workload must be extended with additional query types. Second, our allocation needs to be compared to alternative schemes. Finally, scale-up and speed-up as well as multi-user experiments should be performed to validate previous conclusions.

References

1. Braumandl, R., Claußen, J., Kemper, A.: Evaluating Functional Joins Along Nested Reference Sets in Object-Relational and Object-Oriented Databases. Proc. VLDB Conf., New York, 1998.
2. Bessonow, L.: Simulation objektrelationaler Join-Verfahren in parallelen Datenbanksystemen. Diplomarbeit, Universität Leipzig, 2000.
3. Buhr, P.A., Goel, A.K., Nishimura, N., Ragde, P.: Parallel Pointer-Based Join Algorithms in Memory-mapped Environments. Proc. ICDE Conf., New Orleans, 1996.
4. Bassiliades, N., Vlahavas, I.: Hierarchical Query Execution in a Parallel Object-Oriented Database System. Parallel Computing 22(7), 1996.
5. Carey, M.J., DeWitt, D.J., Naughton, J.F., Asgarian, M., Brown, P., Gehrke, J., Shah, D.: The BUCKY Object-Relational Benchmark (Experience Paper). Proc. ACM SIGMOD Conf., Tucson, 1997.
6. DeWitt, D.J.: Combining Object Relational & Parallel: Like Trying to Mix Oil & Water? Presentation on Object-Relational Summit, 1996.
   Available at: http://www.cs.wisc.edu/~dewitt/.
7. DeWitt, D.J., Naughton, J.F., Shafer, J.C., Venkataraman, S.: Parallelizing OODBMS traversals: a performance evaluation. VLDB Journal 5(1), 1996.
8. Gesmann, M.: A Cost Model for Parallel Navigational Access in Complex-Object DBMSs. Proc. DASFAA Conf., Melbourne, 1997.
9. Gesmann, M., Härder, T.: Supporting Parallel Navigation in Object-Relational DBMSs. Manuscript available at http://wwwdbis.informatik.uni-kl.de/.
10. Graefe, G.: Sort-Merge-Join: An Idea Whose Time Has(h) Passed? Proc. ICDE Conf., Houston, 1999.
11. Härder, T., Rahm, E.: Datenbanksysteme: Konzepte und Techniken der Implementierung. Springer-Verlag, Berlin, 1999.
12. Keßler, U., Dadam, P.: Benutzergesteuerte, flexible Speicherungsstrukturen für komplexe Objekte. Proc. BTW Conf., Braunschweig, 1993.
13. Keller, T., Graefe, G., Maier, D.: Efficient Assembly of Complex Objects. Proc. ACM SIGMOD Conf., Denver, 1991.
14. Olson, M.A., Hong, W.M., Ubell, M., Stonebraker, M.: Query Processing in a Parallel Object-Relational Database System. Data Engineering Bulletin 19(4), 1996.
15. Rahm, E.: Dynamic Load Balancing in Parallel Database Systems. Proc. Euro-Par Conf., Lyon, 1996.
16. Shekita, E.J., Carey, M.J.: A Performance evaluation of Pointer-Based Joins. Proc. ACM SIGMOD Conf., Atlantic City, 1990.
17. Stöhr, T., Märtens, H., Rahm, E.: Multi-Dimensional Database Allocation for Parallel Data Warehouses. Proc. 26th VLDB Conf., Cairo, 2000.
18. Stonebraker, M.: Object-relational DBMSs: the next great wave. Morgan Kaufman Publishers, San Francisco, 1996.
19. Teeuw, W.B., Blanken, H.M., Complex Object Joins in a Distributed Database. Proc. CSN Conf., Amsterdam, 1991.
20. Wang, Q., Maier, D., Shapiro, L.: Revisiting Reference Materialization Techniques for Object Query Processing. Technical report CSE-99-004, Oregon Graduate Institute, 1999.