An Evaluation of Alternative Object Reassembly Strategies

Christian Rich, Marc H. Scholl
Faculty of Computer Science, University of Ulm, Germany
e-mail: {rich, scholl}@informatik.uni-ulm.de

Abstract

The performance of object reassembly, notably the most expensive operation and therefore mainly determining the overall performance, is clearly crucial for the success of object-oriented databases. In this paper, we examine alternative processing strategies for object reassembly on a nested relational storage manager (DASDBS) and a flat record storage system (Oracle), as two possible storage subsystems for object database systems. As an example ODBMS we will use COCOON throughout this paper. We show how to exploit the complex record features of DASDBS, which allow innovative processing techniques that can result in performance improvements of up to an order of magnitude. In particular, DASDBS as a research vehicle can even outperform industrial strength, well-tuned strategies performed by Oracle.

1. Introduction

Object database systems (ODBMS) are currently considered as the new generation of DBMSs. The overall goal is to close the gap between advanced applications, e.g. engineering or scientific applications, and the database system. ODBMS are designed based on the observation that today's databases supporting simply structured data are quite successful in commercial areas, but future applications need the support for more complex structures, with complex integrity constraints built into the model and with powerful, set-oriented query interfaces.

It is clearly crucial for the success of ODBMSs to find efficient implementations that improve on the performance of relational systems, rather than being powerful in terms of modeling and features, but just too slow to be used. In a previous paper [RS93] we described the design, architecture and the optimization scheme of COCOON [SLR+92]. This ODBMS project implements an object model, with a powerful query and update language (COOL), on top of a complex record storage manager (DASDBS) [SPSW90]. We believe that the following concepts are crucial for the performance of ODBMSs. First, non-procedural query languages, which offer high level, set-oriented database access. This enables bulk sequen-
tial I/O's, avoiding a one–tuple—at—a–time bottleneck. Further, set—oriented processing can be exploited by parallel processing strategies. Second, efficiency through optimizations performed by the DBMS. This had been one of the fundamental innovations of the relational approach to databases, but is even there still a research issue. However, it is even more important in object database systems, dealing with large amounts of complex data, like shared data objects, connected by complex semantic relationships. We consider algebraically expressible, generic execution strategies as the base concepts to query processing, as a clean and modular approach. Third, object reassembly, that is, retrieval and reconstruction of large and complex object structures, is a very expensive and a heavily used operation, and therefore very crucial to the overall performance.

In this paper, we evaluate alternative processing strategies for object reassembly on a nested relational storage manager (DASDBS) and a flat record storage system (Oracle), as two possible underlying storage systems for COCOON. There are several reasons why reassembly of object structures is heavily used. First, normalization spreads one logical object over many flat records in relational tables to reduce redundancy. Thus reassembly is needed to query them. Second, objects may be partitioned, for example, due to the mapping of inheritance hierarchies. Further, the application of object–valued functions referencing other objects, dynamically constructs complex structures (path queries)\(^1\). Physical support may significantly improve performance for these joins. Index structures tailored for accessing objects along a reference chain are considered e.g. in [KM90a, KM90b, Ber90]. However, it is important to recognize that for efficiency, it is not typically a good idea to first traverse one complex object completely, before considering the next one. Rather, object reassembly can be performed in a set–oriented way by joins, which have several possible execution orders, and several possible implementation strategies. Reordering joins that express path queries, corresponds, to changes of the order of traversal, that is, whether we do forward or backward traversals, or start in the middle of a path query. The key techniques discussed in this paper are (hierarchical) clustering and embedded (sets of) object references, corresponding to the direct storage model. Innovative processing techniques are enabled, which can result in performance improvements of up to an order of magnitude. Since object reassembly is needed in every ODBMS and the examined processing strategies are quite general (based on hierarchical clustering an embedded (sets of) object references) the results of this examination can be applied to many other ODBMS implementations as well.

The paper is structured as follows: Section 2 introduces an experimental object schema, and describes the mapping to the storage systems. In Section 3 we presents COOL object reassembly strategies, which efficiently exploit the nested storage structures used. Section 4 describes the experimental setup, before we present the performance results, and discuss pros and cons of alternative reassembly strategies in Section 5. Finally, we conclude in Section 6.

---

1. Notice that in COCOON, as an object–function model, path queries are expressed by nesting function applications.
2. Mapping of COCOON to the Storage Systems

We used DASDBS, our prototype storage system for complex objects, as the target platform for COCOON [SLR+92]. The COCOON architecture and the optimization methodology is described in [RS93]. In addition to the DASDBS based realization of COCOON, we implemented COCOON on top of Oracle, as a commercial platform for competition. In the following, we will focus on comparing the performance for object reassembly. We first show the object schema we used in the experiments. Then we discuss alternative physical designs and their impact on query processing for Oracle and the DASDBS based realization, respectively.

2.1 The Experimental Object Schema

We now present a logical database schema, defining object structures, which will be used later on to describe, analyze and compare different reassembly strategies. We focus on a simple scenario, having just two classes, Projects and Employees. The COCOON type and class definitions, as well as the graphical representation is given in Figure 2.1

```
define database CompanyDB;

define type project=
    title: string,
    budget: integer,
    account: integer,
    description: string,
    staff: set_of employee;

define type employee=
    name: string,
    salary: integer,
    works_in: project inverse staff,
    phone_number: string;

define class Project: project;
define class Employee:employee;
end.
```

![Diagram](image)

Figure 2.1: Our example DB—World

In the definition as given above, there is a one—to—many relationship between Projects and Employees. In addition, we will consider the case of a many—to—many relationship as well, by assuming that Employees can be involved in several Projects at one time. This can be
expressed by changing the Employee type definition, by defining the range of the works_in
function as set_of project (a double arrow head in the graphical notation).

In order to quantitatively compare different reassembly strategies for the most common
tasks, we consider both relationship types, several relation cardinalities, and two types of
queries. The first query results in full reassembly, where all tuples of the first relation are
matched to all tuples of the second relation. The second query reassembles just objects
qualified in a previous selection. Two example queries in the schema as given in Figure 2.1,
are given below. Both queries access Projects and Employees. Query Q1 (or a part of that
query) needs to access all Projects, together with all their Employees. Query Q2 needs to
access one Project, such as the one named 'performance improvement', and all its Emp-

loyees.

\[
Q_1 := \ldots[ \ldots[\ldots] \text{(staff)} ] \text{(Project)}
Q_2 := \ldots[ \ldots[\ldots] \text{(staff)} ]
     \text{(select \[title='performance improvement'\] \text{(Project))}
\]

Obviously, often there is a trade-off between query- and update-performance, e.g. if
clustering is used. In order to evaluate this trade-off, we consider the following update op-
eration U. This update reassigns all employees involved in project 'evaluation', to project
'support' instead.

\[
U := \text{update \[ \text{works}_\text{in} := \text{select \[title = 'support'\] \text{(Project)} \]}
     \text{(select \[\text{title}(\text{works}_\text{in}(e)) = 'evaluation'\] \text{(e: Employee))}}
\]

2.2 Oracle

In the COCOON implementation on top of Oracle, COOL queries are mapped to SQL
queries. However, the Oracle optimization is done only on a logical level, using a number of
heuristics. On can easily observe that Oracle determines query execution plans without con-
sidering execution costs, e.g. based on relation cardinalities and storage sizes. Rather, the
order of table names and predicates used in the query formulation have a significant influ-
ence. The order of the relations in the FROM-clause (right to left), determines the order
of execution, whenever no other heuristic can be applied. This is true, e.g. if no index is
available (what results in sort merge joins), or if indexes exists on both relations and both do
have the same properties (what results in indexed nested loops join). However, as can be
seen in the measurements as well, the execution strategy as well as the execution order sig-
nificantly determines the resulting performance. Therefore, Oracle query plans, often have
worse performance than might be possible. We took these facts into account, and forced
Oracle to perform the desired query processing strategies, by choosing the right query for-
mulation and creation of appropriate indexes. The query processing strategy with the best
observed performance was then used for comparisons. We considered the following physi-
cal design alternatives for the logical database schema given in Figure 2.1:

01 Relations Project and Employee are stored separately, no indexes are created. Note that
a unique object identifier, a stored value to represent the object itself, is added as an
additional attribute to each table. The one—many relationship is modelled in the usual primary to foreign key manner.

\[
\text{Projects}(\text{PID}, \ldots) \\
\text{Employee}(\text{EID}, \ldots, \text{PID})
\]

In order to perform the given queries \(Q_1\) and \(Q_2\), Oracle will use a Sort–Merge join in this case.

02 We consider Project and Employee relations as given above, but create an index on each primary and foreign key attribute. This results in indexed joins for Oracle. Further, we consider storing pointers in addition to the foreign key in the Employee relation, by storing the ROWID of the Project tuple it joins with, within the Employee tuple. Again, we force Oracle to consider alternative plans, e.g. whether to take Projects or Employees as the outer relation.

03 We consider Project and Employee relations clustered on the \(\text{PID}\), having indexes on primary and foreign key attributes as well. This results in indexed joins supported by Oracle clusters.

04 The second logical design, that is the many—many relationship between Employees and Projects, is modelled by decomposing it into two one—many relationships, and an extra relationship—relation (EmpPro). This relationship—relation and relation Projects are clustered on \(\text{PID}\). Further, we provide indexes on primary and foreign key attributes.

\[
\text{Projects}(\text{PID}, \ldots) \\
\text{Employee}(\text{EID}, \ldots) \\
\text{EmpPro}(\text{EID}, \text{PID})
\]

2.3 DASDBS

In the COCOON implementation on top of the DASDBS storage system, a cost based query optimizer is used to map COOL queries down to operations on physical \(\text{NF}_2\) structures [RS93]. The execution plans generated consist of physical \(\text{NF}_2\)—algebra operators, some of which are DASDBS kernel calls, others, such as joins or address dereferencing, are implemented in the COOL query processor. We use alternative query execution plans, in order to evaluate alternative object reassembly strategies. We now show the physical designs considered:

D1 Relations Project and Employee are stored separately, no indexes are created. The one—many relationship is modelled in the primary to foreign key manner, like in Oracle.

\[
\text{Projects}(\text{PID}, \ldots) \\
\text{Employee}(\text{EID}, \ldots, \text{PID})
\]

D2 Relations Project and Employee are stored separately. The one—many relationship is modelled symmetrically, by storing references as foreign keys in both relations. This results in a set of references stored in a subrelation, for the set valued function \(\text{staff}\).
Note that in addition to the logical reference as a foreign key, we store physical object references (denoted by @), that is, (hierarchical) tuple identifiers (HI−TID).

\[
\text{Projects}(\text{PID}, \ldots, \text{staff}(\text{EID}, \text{@EID})) \\
\text{Employee}(\text{EID}, \ldots, \text{PID}, \text{@PID})
\]

D3 Further, we consider hierarchical clustering of the object valued function \textit{staff}. This results in one nested relation, with \textit{Employees} as a subrelation.

\[
\text{Projects}(\text{PID}, \ldots \text{Employee}(\text{EID}, \ldots))
\]

D4 The second logical design, that is the many—many relationship between \textit{Employees} and \textit{Projects}, is mapped to two relations, each with a set of object references in a subrelation. Notice that in contrast to the flat equivalent in design D4 there is no relationship—relation.

\[
\text{Projects}(\text{PID}, \ldots, \text{staff}(\text{EID}, \text{@EID})) \\
\text{Employee}(\text{EID}, \ldots, \text{works_in}(\text{PID}, \text{@PID}))
\]

3. COOL Object Reassembly

Object reassembly exploiting nested relations and particular the aspects of algorithm design and implementation for nested relations has received little attention. There are several interesting new variants of known strategies ([ME92]) for object reassembly in case of deeply nested structures. When joining nested relations, the correlation (join) predicate might refer to attributes on any level, not just the outermost. However, the result of the join of two nested relations always makes the top level of one argument a subrelation (at any level) of the other (see Figure 3.1).

![Figure 3.1: Joining Nested Relations](image)

For example, assuming physical design D2 and the execution plan

\[
\text{Projects join[... \text{Projects.staff.EID=Employee.EID}] Employee}
\]

as executable in the COOL query processor, the result will be a relation with schema

\[
\text{Projects}(\text{PID}, \ldots \text{staff}(\text{EID}, \text{name}, \ldots))
\]

where \textit{staff} is a new subrelation (of each project tuple), that collects all employees working for that project. The second argument relation, \textit{Employee}, was made a (second level) subrelation of \textit{Projects} in the result of this join query.
3.1 Object Reassembly Strategies

The following algorithms can be selected without any requirements on extra physical support.

- **(Index—)** Nested Loops join. The Nested Loops join (NLJ) is, like in the (flat) relational model, the naive iteration over both tuple sets. However, due to the fact that one tuple set is nested within supertuples, the loops are nested to the join level. For each supertuple containing subtuples in the subrelation to be joined, one iteration is performed for this set of subtuples. Assume the join of the subrelation staff of relation Project and the relation Employees, as given above. Each tuple of relation Project is considered, and the iteration is done on the set of subtuples of the subrelation staff and the set of tuples in relation Employees. For large relations, the performance of a Nested Loops join can be significantly improved, if an index on the join attribute of inner relation can be exploited. However, this is only one option. Another improvement is as follows: If the joining tuples of one argument relation are nested within a subrelation, we might force the underlying storage manager to deliver matching tuples for the whole subrelation in one call (this becomes feasible with DASDBS’s object buffers [SPSW90]). Then, NLJ basically becomes a variant of a nested block algorithm, where several join tuples of one argument may be checked against tuples of the other argument without additional I/Os. Further notice that selections can be applied to both relations before performing the join in case of NLJ, but just to the outer relation, in case of INLJ.

- **Sort Merge Join.** Sort Merge join (SMJ) is a well-known strategy for equijoins, and many relational database systems, such as Oracle, use it as the strategy of choice, if INLJ is not applicable. Both relations (say R and S) are sorted on the join attribute. Then these are merged (i.e., joined with linear complexity). For a nested relation, each supertuple (containing subtuples to be joined) is taken one after another, and the subrelation is sorted and merged with the (sorted) second relation, accordingly. Typically, SMJ will be more efficient than NLJ, as well as INLJ using a non-clustered index, in the case of large (sub) relations.

- **(Hierarchical) Cluster Scan.** If the argument relations are already hierarchically clustered (physical design D3, as given above) the join operation will be reduced to a linear hierarchical cluster scan (CS). Thus each object is accessed only once, without any index translation or the like. Obviously, this ”join” strategy is the most efficient one (linear complexity in the size of the join result), unless the join becomes almost a Cartesian Product.

3.2 Improved Object Reassembly Strategies

We now present variants of the join strategies given above that exploit physical object references, thus usually improving the performance. These strategies can be applied, when one
of the relations to be joined includes physical object references (TIDs) in addition to logical
object references, i.e. foreign keys.

- **Link Nested Loops Join.** Link Nested Loops join (LN LJ or @NLJ) is a refinement of
NLJ. Each (sub)tuple of the (sub)relation including the physical references is read, and
the TIDs are used to directly access the partner tuples in the second relation. There is
no scan or index lookup necessary to access tuples of the second relation. Notice that
if physical object references are stored in both relations, both relations could be taken
as the outer relation. Efficiency can be further improved, by collecting large sets of TIDs
and performing set-oriented direct access.

- **Link Sort Merge Join.** Link Sort Merge join (LSMJ or @SMJ) is similar to SMJ. Howev-
er, only one relation, namely the one with physical references, is sorted on these TIDs.
The merging step is to use the sorted TIDs to directly access the tuples in the second rela-
tion. Since the TIDs are sorted, duplicate tuple and/or page fetches from the second relation
are avoided. Further, the inner relation is read in ascending order of addresses, thus
avoiding extraneous disk latencies.

Considering select-project-join (SPJ-) queries, selections can be applied before a link
join, only if these selections apply to the relation with the physical references. However, if
physical object references are stored in both relations, selections on any one of the two rela-
tions can be applied before the join.

### 4. Experimental Setup

All performance experiments were run on a Sun 3/280, with SunOS 4.1.1. We used Oracle
V6.0 with embedded dynamic SQL. This version of Oracle runs as two Unix processes: a
monitor process for interacting with the user and a second process which performs all op-
erations on the database. Both processes ran on the same machine. The DASDBS based
implementation runs within a single Unix process. We used the Unix getusage call in order
to determine response times (real time, no other user processes).

We realized all physical designs given in Section 2. The cardinality of class Project was fixed
to 100 tuples, so as to model a selection that has already been evaluated. The number of
Employees involved in a Project was drawn from a uniform distribution over ranges of a)
1–5, b) 5–10, c) 10–50, and d) 50–100. This results in a number of 300 to 7500 Employee
tuples in total. These cardinalities had been used for the one–many relationship, as well as
for the many–many relationship. The cardinality of relation EmpPro therefore ranged
from 300 to 7500 tuples. The data stored in both systems were exactly the same, and when a
selection was performed on Projects, we took care to select projects with the average num-
ber of employees.
5. Experimental Results

In this section, we show the results of evaluating object reassembly on the DASDBS based implementation and Oracle, respectively.

First, object reassembly on flat relations is compared. This results in similar execution strategies for Oracle and DASDBS, respectively. Then we consider hierarchical clustering in DASDBS, as described in Section 2.3, and object reassembly strategies on these nested structures, as described in Section 3. We compare object reassembly for one—many relationships in Section 5.2 and for many—many relationships in Section 5.3.

5.1 Flat Tables

This first experiment using flat tables only was used so as to “calibrate” the DASDBS prototype and the Oracle product. We consider the standard flat relational physical designs (O1 and D1), with the join query Q1 and the select—project—join query Q2. Both are implemented by a SMJ on Oracle and DASDBS. We evaluated both possible orders for inner/outer relations (P indicates Project is outer, E means Employee is outer).

As can be seen in Figure 5.1, SMJ is faster, if the relation with the smaller number of matching tuples is used as the outer relation as this reduces the cost for merging. Oracle processes relations in the FROM clause from right to left, i.e., the smaller relation should be the last one in the FROM clause! The response time can be improved roughly by 7% in this example, if the Project relation is chosen as the outer relation, instead of the Employee relation. This result is similar for DASDBS and Oracle.

Comparing response times of Oracle and DASDBS, we find Oracle to be up to 70% faster for query Q1, and up to 6 times faster for query Q2. This is not surprising, as Oracle is a commercial, well-tuned system, whereas DASDBS as a university prototype was mostly implemented by students, with the main focus on functionality rather than on fine-grained code tuning.

Note that for simplicity, we will use the terms DASDBS and Oracle in the following, instead of the term COCOON implementation on top of DASDBS, and Oracle, respectively.
5.2 One–To–Many Relationship

Next we compare object reassembly, when nested storage structures are used in DASDBS, and physical object relationships are supported by 1) logical references, 2) physical references and 3) (hierarchical) clustering.

5.2.1 Logical References

When objects are stored in several records, storing logical references as foreign keys is the simplest way to store relationships. We now compare reassembly strategies in DASDBS and Oracle, if only these logical references can be exploited.

![Figure 5.3: Join Query Q1](image1)

![Figure 5.4: SPJ Query Q2](image2)

We consider query Q1 and physical designs O1 and D2. Obviously SMJ will be the strategy of choice. As expected, SMJ performs much better than NLJ, for medium to large relation sizes. NLJ is therefore not considered for query Q1.

Considering design D2, another option is to first unnest the subrelation containing the join attribute, before joining (the third DASDBS alternative marked with ”μ” in Figure 5.3). This is advantageous, if the number of subtuples is small (300–3000 Employees), such that buffer space gets better exploited, more tuples are sorted at once, and therefore less merging runs on the second relation are needed. However, it is favorable if unnesting and sorting, which requires substantial work, stays within buffer space.

For query Q2 performed on DASDBS, SMJ is not inferior to NLJ. This can be explained as follows. Because the implementation of NLJ retrieves tuples in units of blocks, rather than individually (see Section 3.1), and the result of the selection performed in the SPJ query Q2 is one block, this results in one retrieval on the inner relation. Performing a SMJ requires sorting, what is additional work in this case.

Comparing the best strategies found for Oracle and the DASDBS based COOL query processor, we find Oracle to perform 2 to 3 times faster, for both queries, Q1 and Q2.

5.2.2 Physical References

We now show the performance results of object reassembly strategies, if physical object references can be exploited. In DASDBS we used (hierarchical) tuple identifiers (HI–TIDs)
stored within the referencing tuples. In Oracle ROWIDs stored within the referencing tuples, as well as (separate stored) indexes are considered.

In Figure 5.5 we show the performance results for object reassembly, using physical designs O2 and D2. In Oracle, we can exploit either the index on relation Employee, or the index on relation Project, or exploit the pointers, stored inside the Employee tuples. The most efficient way to process this kind of join, is to use the smaller relation as the outer relation (relation Project), and an index on the larger, inner relation (Employees). Using the larger relation, i.e. Employees, as the outer relation, can be improved by exploiting pointers, instead of using the index. However, since the pointers are in the direction which is opposite to the most efficient direction for processing, the performance is still inferior to the index nested loops join, using Project as the outer relation. The COOL query processor on top of DASDBS is able to exploit pointers, stored in the direction of the most efficient way for processing. For query Q1, the resulting performance comes up to the performance of Oracle.

As can be seen in Figure 5.6, performing query Q2 in Oracle by INL using Projects as the outer relation, is far better than exploiting pointers stored within Employees. The COOL/DASDBS query processor can exploit pointers efficiently. However, the number of tuples to be joined after the selection is small. This results in small performance improvements compared to the overall cost, which therefore stays inferior to the performance of the best strategy found for Oracle.

5.2.3 Clustering

We now give performance results of object reassembly strategies, that exploit clustering. Obviously this is the strongest way to support object reassembly.

In Figure 5.7, we show the performance results for object reassembly, using physical designs O3 and D3. Considering a cluster supported join in Oracle, we observe again that it is advantageous to take the smaller relation as the outer relation. In the COOL/DASDBS query processor, however, even no index access and address translation has to be performed, and the resulting performance is superior to Oracle by a factor ranging from 5 to 7 for query Q1.
When considering query Q₂, this advantage melts away, with the number of tuples to be joined after performing the select operation. This results in similar performance for Oracle and DASDBS.

5.2.4 Overall Comparison

We will now use the results given above, and compare the impact of physical support for relationships (i.e. logical references, physical references and (hierarchical) clustering) on the performance.

In Figure 5.9 we compare alternative reassembly strategies in Oracle, for each physical design. We observe, that Oracle’s optimizer does not determine the best join ordering, that is, which of the relations to take as the outer relation. This results in substantial performance differences, for alternative relation orderings in the FROM—clause. Choosing the outer relation correctly is already necessary for SMJ. However, when indexes or clusters are created in order to support object reassembly, choosing the right ordering is even more important. When the order is chosen incorrect, performance will even decrease. Further, remember, that storing physical references in Oracle, does not improve performance, because the most efficient way to perform one—many relationship joins is typically opposite to
the direction of these pointers. Comparing the best strategies found for each physical support, we get an improvement up to 8% by indexes and up to 14% by clustering.

For query Q₂, physical support through indexes and clustering can result in high performance improvements. However, if Oracle chooses the wrong operations ordering, performance will decrease as well.

![Figure 5.11: Join Query Q₁](image1)

![Figure 5.12: SPJ Query Q₂](image2)

In Figure 5.11 we compare object reassembly in the COOL/DASDBS query processor, for either of the three alternatives of physical support (plain value based, with pointers, materialization). We see that storing (sets of) physical object references within the referencing objects, substantially speeds up object reassembly (a factor of 2–3 was observed). This is because the referenced tuples of the second relation can be fetched directly, without scans, index lookups, or sorting. Finally, materializing object functions, as expected, performs best. The cost function is a linear function of the relation cardinality, compared to e.g. a logarithmic function for Sort Merge join. The improvement increases with the relation cardinality, and ranged in our experiment from a factor 5 to 7 over pointers and 10 to 20 over value–based joins. For SPJ query Q₂, similar results are observed, and shown in Figure 5.12.

![Figure 5.13: Join Query Q₁](image3)

![Figure 5.14: SPJ Q₂](image4)
In Figure 5.13 we compare the Oracle and the DASDBS based implementation, by using the best reassembly strategies found in both systems, for each physical support. We see that exploiting physical references and clustering results in substantial performance improvements in the DASDBS based implementation. This results in catching up the performance of the Oracle based implementation, if physical references can be exploited, and even being superior, if object reassembly is supported by clustering.

5.3 Many—To—Many Relationship

We now give performance results of object reassembly for a many—to—many relationship. For simplicity, we consider just one physical design in each system, i.e. design O4 and D4, which is the one giving the best physical support (without introducing redundancy). Notice, that for many—many relationships, the normalized physical design in flat relations, results in three relations, and at most two of them can be clustered in one Oracle cluster. However, in DASDBS, the ability of storing sets of foreign keys within tuples, results in the possibility to represent the many—many relationship more naturally, within two relations.

As can be seen in Figure 5.15, for query Q1, the performance of the DASDBS based implementation is much better than the one of Oracle. This is the result from the fact, that there has to be performed only one join operation in the COOL/DASDBS query processor, compared to two join operations in Oracle.

For SPJ—Queries, again, this advantage melts away with the number of result tuples after the select operation. For query Q2, this results to almost the same performance, when comparing the best strategy found for Oracle and the DASDBS based implementation, respectively. However, when the execution order is chosen, such that the selection is performed after joining, a lot of unnecessary work is performed. As already mentioned, the execution order in Oracle is determined by the order of relations in the FROM—clause. For query Q2, this results in a possible performance decrease by a factor of 55.
5.4 Updates

In order to evaluate, how expensive update operations are in respect to the physical design, we ran the following experiment: We consider physical designs O2, O3, and D3, and perform the update operation U, given in Section 2.1. In Oracle, as well as in the DASDBS based implementation, first, two selections on relation Projects have to be performed, in order to determine the key values and the physical addresses, respectively. The second step in Oracle, is to update the affected Employee tuples, by changing the foreign keys accordingly. Further, updating the index on the changed attribute is necessary as well. Considering design O3, that is Employees and Projects are clustered, additional overhead is needed. This is because the placement of data depends on the cluster key, so changing cluster keys usually causes physical relocation of the tuples. In the DASDBS based implementation, the update requires to delete and insert subtuples (containing employees) anyway.

Comparing the performance (Figure 5.17), we find what we expected. In Oracle, updating the cluster key becomes more expensive, than updating the foreign key attribute in non-clustered relations. The additional overhead is about 70%. Using the DASDBS based implementation, it is not possible to improve update performance by exploiting hierarchical clustering. The performance of DASDBS stays inferior, than the performance of Oracle. However, the performance difference of Oracle and DASDBS stays within the general performance difference found for equivalent operations.

6. Conclusion

In this paper, we presented and evaluated alternative processing strategies for object reassembly on a nested relational storage manager (DASDBS), and a flat record storage system (Oracle), as two possible underlying storage subsystems for object database systems. As an example ODBMS we used COCOON throughout this paper. We showed how to exploit the complex record features of DASDBS, which allow innovative processing techniques that can result in performance improvements of up to an order of magnitude. Since object reassembly is needed in every ODBMS and the examined processing strategies are quite general (based on a few key concepts) the results of this examination can be applied to many other ODBMS implementations as well.

The key techniques discussed in this paper are (hierarchical) clustering and embedded (sets of) object references. We found pointers to improve performance significantly, if used correctly. In general they perform best, if used in the smaller relation, in order to find matching tuples in a larger relation. The use of pointers in SPJ queries is superior, only if the pointers
are not in the opposite direction to the most efficient way for processing. However, storing (sets of) pointers in both directions, overcomes this problem.

Comparing Oracle and DASDBS, we found the Oracle implementation to perform in general much faster than the DASDBS based implementation, for the same query processing strategy. However, exploiting complex record features of DASDBS, and special processing techniques, changes this result. Object reassembly on one—to—many relationships supported by hierarchical clustering in DASDBS, outperforms Oracle clusters. Further, many—to—many relationships supported by hierarchical clustering of sets of object references in DASDBS, outperforms Oracle (exploiting a cluster) as well. The performance of updates showed to be as expected. Updating (hierarchically) clustered data is more expensive than updating non clustered data, however, the performance is still reasonable.

This results in the following overall conclusion: Oracle as a commercial, well tuned DBS leads to better performance than DASDBS, a university prototype, for equivalent operations (factor 2 to 6 in our examples). Conversely, if efficient reassembly strategies are exploited in DASDBS, based on complex storage system features, such as hierarchical clustering, these results are twisted. Therefore, the goal should definitely be, to enrich commercial DBSs so as to support complex storage structures (like hierarchical clustering), to include efficient reassembly strategies exploiting these storage structures, and to extend the query optimizers correspondingly.

References


