Query Optimization Techniques Exploiting Class Hierarchies

Sophie Cluet[1] *Guido Moerkotte*[2]

[1]INRIA
BP 105
Domaine de Voluceau
78153 Le Chesnay Cedex
France
Sophie.Cluet@inria.fr

[2]Lehrstuhl für Informatik III
RWTH-Aachen
Ahornstr. 55
52074 Aachen
Germany
moer@gom.informatik.rwth-aachen.de

Abstract

Since the introduction of object base management systems (OBMS), many query optimization techniques tailored for object query languages have been proposed. They adapt known optimization techniques to the OBMS context, exploit special object-oriented features, or give solutions to problems specific to querying objects. Nonetheless, one of the most prominent features of object models — namely *class hierarchies* — have so far not been exploited for query optimization.

The current paper proposes new optimization techniques for queries referring to classes integrated into a class hierarchy. The techniques are generic in the sense that we do not give a set of algebraic equivalences the optimizer has to apply, but instead try to provide the reader with a general understanding of how to exploit class hierarchies for query optimization purposes. We give general descriptions of the techniques as well as illustrating examples.

Besides yielding considerable savings in terms of execution time, the presented optimization techniques have the additional advantages of (1) being easily implementable and (2) resulting only in a neglectable increase in optimization time.

1 Introduction

Since the introduction of OBMS, much work has been devoted to optimizing object queries. Special index structures for path indexes [2, 12, 18, 20], class indexes [17, 16], and function materialization [11] have been proposed. Logical optimization techniques have been developed. They cover path evaluation [1, 13], factorization [5], unnesting [6, 19], optimization in the presence of expensive methods [9, 15], disjunctions [14] or aggregates [7].

Looking at all the effort, it is surprising, that, in the context of query optimization, class hierarchies did not receive special attention. The only special support developed so far are index structures that are capable to index multiple sets or classes in a hierarchy [17, 16, 20]. So far, no effort has been made to exploit class hierarchies to optimize queries that refer to classes integrated into a class hierarchy. The goal of the paper is to start investigations in this direction.

Either because they are made available to the users or because of preliminary rewritings [10], class extents are usually the collections on which OO queries are issued. We rely on the fact that classes are organized in a hierarchy to introduce several generic optimization techniques for queries referring to class hierarchies. The techniques are generic in the sense that we lay the basis for applying all the traditional techniques for query optimization to queries involving class hierarchies. This is done by pointing out the crucial points that have to be considered. The main ideas are (1) to make the subclasses of a class explicitly visible to the optimizer and (2) to exploit static as well as dynamic type information.

Using these two simple ideas, we show that by exploiting differences in logical and physical properties present in different subclasses can result in much cheaper plans. Further, we give techniques that reduce the number of joins, scans and method evaluations. This gives way to considerable savings that by far outbalance the added optimization cost of having to consider more alternatives.

The paper is organized as follows. The next section presents the preliminaries. Among those, it discusses several possible implementations for class extents. Section 3 introduces the generic optimization principle of *execution plan tuning* and the optimization techniques *input splitting* and *reasoning about symmetry*. Section 4 demonstrates how static and dynamic type information can be exploited to further reduce query evaluation costs. To support this kind of optimization, a new type of easy to maintain aggregate information is introduced and its application to query optimization demonstrated. Effective pretests are introduced which further save access and method evaluation costs. Section 5 concludes the paper.

2 Preliminaries

We assume general familiarity with the basics of object models and OBMSs.

We base our discussion on the simple class hierarchy of Figure 1. The boxes on the right side of the figure contain some attribute definitions or their refinement. A query[1] like

select	e
from	Employee e
where	e.age > 30

not only returns *Employee*s but also *Manager*s and *CEO*s of age greater than 30. If no index is present, a scan over the extents (i.e., the set of all objects which are instances of a class) of *Employee*, *Manager*, and *CEO* is necessary. There exist several possibilities for implementing scans over extents. In this section, we will briefly sketch and classify the different alternatives.

[1] Queries are stated in an OQL-like language [4].

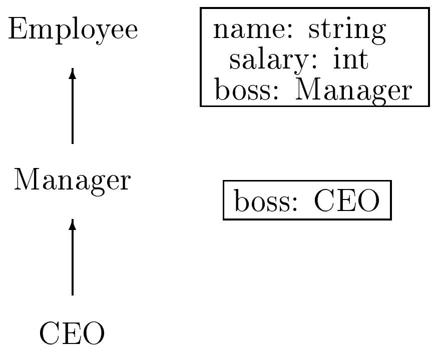

Figure 1: Example Class Hierarchy

The basic preliminaries for implementing extents can be summarized as follows. Most OBMSs organize an object base into areas or volumes. Each area or volume is then further organized into several files. A file is a logical grouping of objects not necessarily consisting of subsequent physical pages on disk. Files don't share pages.

The simplest possible implementation to scan all objects belonging to a certain extent is to perform an area scan and select those objects belonging to the extent in question. Obviously, this is far to expensive. Therefore, some more sophisticated possibilities to realize extents and scans over them are needed. The different possible implementations can be classified along two dimensions. The first dimension distinguishes between logical and physical extents, the second distinguishes between strict and (non-strict) extents.

Logical *vs.* **Physical Extents**

An extent can be realized as a collection of object identifiers. A scan over the extent is then implemented by a scan over all the object identifiers contained in the collection. Subsequently, the object identifiers are dereferenced to yield the objects themselves. This approach leads to logical extents. Another possibility is to implement extent membership by physical containment. The best alternative is to store all objects of an extent in a file. This results in physical extents. A scan over a physical extent is then implemented by a file scan.

Extents *vs.* **Strict Extents**

A strict extent contains the objects (or their OIDs) of a class excluding those of its subclasses. A non-strict extent contains the objects of a class and all objects of its subclasses.

Given a class C, any strict extent of a subclass C' of C is called a subextent of C.

Obviously, the two classifications are orthogonal. Applying them both results in the four possibilities presented graphically in Fig. 2. The underlying class hierarchy is that of Fig. 1. We use the same name to denote a class and its strict extent, and the class name suffixed by a $*$ symbol to denote its non-strict extent.

Note that, while in the logical extent case the redundancy evoked by the membership of an OID in multiple sets seems neglectable, this is most probably not the case when

	Strict Extents	Extents
LOGICAL	Employee: {e1, e2, ...} Manager: {m1. ...} CEO: {c1, ...}	Employee*: {e1, e2, ..., m1, ..., c1} Manager*: {m1. ..., c1, ...} CEO*: {c1, ...}
PHYSICAL	Employee: {e1: [name: Peter, salary:20.000, boss: m1], e2: [name: Mary, salary:21.000, boss: m1], ... } Manager: {m1: [name: Paul, salary:100.000, boss: c1], ... } CEO: {c1: [name: May, salary: 500.000, boss: c1], ... }	Employee*: {e1: [name: Peter, salary:20.000, boss: m1], e2: [name: Mary, salary:21.000, boss: m1], ..., m1: [name: Paul, salary: 100.000, boss: c1], ..., c1: [name: May, salary: 500.000, boss: c1], ... } Manager*: {m1: [name: Paul, salary:100.000, boss: c1], ..., c1: [name: May, salary: 500.000, boss: c1], ... } CEO*: {c1: [name: May, salary: 500.000, boss: c1], ... }

Figure 2: Implementation of Extents

considering physical extents. — Nevertheless, physical containment of objects does not necessitate their duplication if a file is allowed to contain other files. This is possible in e.g. EOS [3].

Note that it is possible to have different implementations of an extent at the same time. For instance, an OBMSs could maintain strict physical extents and, redundantly, non-strict logical extents.

Some of the optimization techniques we propose in the sequel rely on the presence of strict logical or physical extents.

3 Execution Plan Tuning and Input Splitting

This section argues for making the subextents of an extent visible to the optimizer. That is, they can be explicitly found in the query execution plan. The advantage of this approach is demonstrated by applying several optimization techniques to these plans. The disadvantage is the increase in optimization time since more alternatives can be considered now. But since the savings can be considerable, for example more than 50% of the total join cost, the increase in optimization time seems neglectable.

3.1 Execution Plan Tuning

Similar to relations in the relational context, collections and especially extents have certain properties attached to them. In general, one distinguishes between physical and logical properties. By physical, we denote those properties, that concern the storage of the extent whereas logical properties refer to the content of the extent or the values of the contained objects. In this section, we deal with one logical and two physical properties:

1. attribute value distributions
2. order
3. indexes

The main idea is to exploit these properties in case they vary for different extents. We demonstrate possible exploitations of this idea by means of a simple example.

Consider the example class hierarchy consisting of *Employee*, *Manager* and *CEO*. The logical property we will consider is the value of the salary attribute. Obviously, the range of the salary will differ for employees, managers, and CEOs. Further, there could be one index on *salary* for managers and one for CEOs, but none for employees. Under these assumptions let us optimize the following simple example query

select e
from Employee* e
where e.salary > 100.000

The query does not only ask for employees but also for the members of subextents of *Employee*, that is, for *Manager*s and *CEO*s. The explicit $*$ used here is typically not a construct of object query languages but only used for the purposes of the paper in order to emphasize the existence of subextents.

We argue strongly, to make this fact explicitly visible to the query optimizer by translating the query into:

$$\text{select[salary}>100.000](\text{Employee} \cup \text{Manager} \cup \text{CEO})$$

rather than into

$$(1) \quad \text{select[salary}>100.000](\text{Employee}^*).$$

Note that the latter is the predominant way to translate a query. The reason for this is that extents are typically implemented as logical non-strict extents by using e.g. C++ class constructors to add a newly created object to the extent. Since a call to a constructor of a class implies a call to the constructor of the base (or super) classes, the result is a non-strict extent.

By incorporating the extents and their subextents explicitly into the query evaluation plan, the query optimizer is able to use different evaluation plans specifically tailored for the different (sub-) extents. Let us demonstrate this point by means of the example query.

Let us assume that no employee earns more than \$100.000, and that 30% of the managers and 70% of the CEOs earn more than \$100.000. Under these assumptions, an index scan for CEOs is not worth it since the selectivity factor is beyond 50%. Using the index for managers is still quite reasonable although the selectivity factor of 30% comes close to the limit.[2]

Hence, a reasonable plan would be:

$$\begin{array}{c}
\text{select[salary}>100.000](\text{extscan(Employee)}) \\
\cup \\
\text{indexscan[salary}>100.000](\text{Manager}) \\
\cup \\
\text{select[salary}>100.000](\text{extscan(CEO)})
\end{array}$$

Note that this plan could only be developed, since the different subextents of *Employee* are made visible by incorporating them explicitly into the query evaluation plan. By that, different decisions could be made for the different extents *Employee*, *Manager*, and *CEO*.

We will built upon explicit mentioning of the extents in the plan for all subsequent optimization techniques. Since only then, advanced optimization techniques for treating class hierarchies can be applied.

Let us now assume that the OBMS administrator has chosen to materialize the *min/max* values of the *salary* attribute for employees, managers, and CEOs (a reasonable assumption). Then, the optimizer knows beforehand that no employee will qualify. Hence, the plan simplifies to:

$$\begin{array}{c}
\text{indexscan[salary}>100.000](\text{Manager}) \\
\cup \\
\text{select[salary}>100.000](\text{extscan(CEO)})
\end{array}$$

[2] We slightly oversimplify the cost estimation to keep the discussion simple and to be able to emphasize the main points.

The observation here is analogous to the one above. If there would not be distinct min/max values for the salary of each of the extents, there would be no way for the optimizer to perform this optimization.

Let us summarize the main point up to now. The general idea of *execution plan tuning* is to exploit the differences found for the different subextents of an extent. The technical means applied to yield plans which are specifically tailored for each subextent is pushing the algebraic operators down over the union. Recall that there are no problems with duplicates since the strict extents of two different classes are disjoint.

As we will see next, this technique does not always yield a better plan. Sometimes it is better to leave the union at the bottom most level and union the pure extents. Nevertheless, abandoning the possibility of better plans by sticking to some pure extent in the execution plans seems unreasonable.

As already stated, it is not always better to push all the operators inside the unions. To see this, assume that there exists a joined B-tree index on the names of managers and CEOs (i.e., Manager* is indexed on names). Assume further that the query is modified such that the outcome is supposed to be sorted by name.

select	e
from	Employee* e
where	e.salary > 100.000
order by	name

Now, since sorting is expensive, and the 30% selectivity for managers is close to the border where the index scan pays, it might be better to use the already present sort order on the names of all managers and CEOs. Hence, a reasonable plan would be

select[*salary*>100.000](indexscan[*name*](*Manager* union *CEO*))

A last execution plan for our simple example query can be deduced if the B-tree index is on the names of CEOs only. Then,

sort[name](indexscan[salary>100.000](Manager)
mergeunion
select[salary>100.000](indexscan[name](CEO))

could be a perfect plan, if we again require the result to be sorted. Note that here the union — which is modified to a *mergeunion* in order to keep the sort orders — is pushed outside again; or, to put it differently, the other algebraic operators are pushed inside.

Note that all the variety in plans even for a query as simple as the example query would not have been possible without making the subextents visible to the optimizer. Having them visible to the optimizer allows to exploit the different properties of the subextents and to produce an optimal overall plan by chosing individually tailored subplans for each subextent.

3.2 Operator Input Splitting and Reasoning by Symmetry

In this subsection, we give another argument that favors the explicit treatment of subextents by introducing two optimization techniques. The first will be operator *input splitting* whose main advantages are (1) a higher degree of interleaving of I/O and computation and (2) savings in main memory consumption. The second optimization technique is *reasoning by symmetry* which allows to eliminate joins. Input splitting is the prerequisite for this technique.

Despite the fact that input splitting is not only useful for the join operator, but for other algebraic operators as well, we base our discussion on joins only since this is by far the most expensive algebraic operator.

Consider the following simple join query.

select e1, e2
from Employee* e1, Employee* e2
where e1.salary = e2.salary

It retrieves all pairs of employees with the same salary.

Making the subextents of *Employee* explicitly visible, the evaluation of the query requires the evaluation of nine joins:

$$
\begin{array}{ccc}
\text{Employee} & \bowtie & \text{Employee} \\
\text{Employee} & \bowtie & \text{Manager} \\
\text{Employee} & \bowtie & \text{CEO} \\
\text{Manager} & \bowtie & \text{Employee} \\
\text{Manager} & \bowtie & \text{Manager} \\
\text{Manager} & \bowtie & \text{CEO} \\
\text{CEO} & \bowtie & \text{Employee} \\
\text{CEO} & \bowtie & \text{Manager} \\
\text{CEO} & \bowtie & \text{CEO}
\end{array}
$$

The result of these joins is then to be unioned to yield the final result. Thus, we split the big join

$$Employee^* \bowtie Employee^*$$

into the above nine smaller joins.

This splitting of the input of an operator exhibits several advantages:

- Instead of producing one big package, nine small packages are produced.
  If pipelining is applied, the interruption of the pipeline due to a sort-merge join or a hash-join is not as bad as it would be otherwise.

  Further, this can be exploited for parallelization.

- Memory consumption is smaller since not all subextents of *Employee** have to be read into main memory at once.
  Again, this is most advantageous in case of pipelining.

- Interleaving of join computation and I/O is possible.
  After the employees have been fetched, the join computation can start. During that, the managers can be fetched from disk, and so on.

Another advantage of input splitting is less obvious and leads us to the next optimization technique we introduce. By *reasoning about symmetry*, we note that some pairs of joins exist, that produce symmetric output. For example, *Manager* $\bowtie$ *Employee* and *Employee* $\bowtie$ *Manager* do not produce the same outcome if we have ordered tuples but it is much cheaper to produce the outcome of the join *Employee* $\bowtie$ *Manager* from the result of *Manager* $\bowtie$ *Employee* than computing the join itself. Further, we could use the result of *Manager* $\bowtie$ *Employee* as an implicit representation of *Manager* $\bowtie$ *Employee* $\cup$ *Employee* $\bowtie$ *Manager*. This would result in cost savings when processing subsequent algebraic operators. Taking this kind of reasoning into account, the joins that have actually to be computed are

$$\begin{array}{ccc}
\text{Employee} & \bowtie & \text{Employee} \\
\text{Employee} & \bowtie & \text{Manager} \\
\text{Employee} & \bowtie & \text{CEO} \\
\text{Manager} & \bowtie & \text{Manager} \\
\text{Manager} & \bowtie & \text{CEO} \\
\text{CEO} & \bowtie & \text{CEO}
\end{array}$$

Thus, instead of computing nine joins, we only have to compute six joins: 1/3 of the joins is saved.

What run-time savings can we expect when constructing a result for $R \bowtie S$ from a given result $S \bowtie R$ instead of computing $R \bowtie S$ explicitly? First, note that this saving heavily depends on the join algorithm used. Obviously, the smallest saving results from using the best join algorithm. Assuming that the runtime of a hash join is (almost) linear in the size of the outer relation, only a constant factor in saving can be expected since the costs of computing the hash values, performing the lookups, and evaluating the predicate is saved. We only have to pay the costs for tuple construction. The best case arizes, if we only need the join in order to compute some aggregate value like a count. Then, $count(R \bowtie S) = count(S \bowtie R)$ and, hence, nothing has to be computed for those joins left out for symmetry reasons. In this case, saving 1/3 of the joins amounts to saving 1/3 of the execution cost.

4 Type-based Scan Reductions and Pretests

In this section we want to exploit type information as a means to reduce the number of scans and evaluations of expensive expressions necessary to evaluate a query. As an introduction consider the following — admittedly stupid but simple — example query.

select p
from Person p
where p.age = 5 and p.age = “five”

By performing the regular type check, the query is detected to be unsafe — not matter what the schema is since the attribute *age* can either be declared to be of type *int* or *string*. Neglecting the type error we will get, another way to look at it is that the result of the query is the empty set; no object qualifies due to disagreeing types for the value of the attribute *age*. Moreover, this can be verified at query optimization time, if we add the type inference to the query optimization.

The idea of this section is to perform (more subtle) reasoning about types in order to reduce the number of qualifying objects. This will allow us to (1) get rid of some scans (no qualifying objects) and (2) avoid evaluating some expensive predicates (some elements of a scan cannot qualify). In the first case we speak of *type-based scan reductions*, in the latter of *type-based pretests*.

For type-based scan reductions, we will not only use static type information as given by the schema, but also dynamic type information which is deduced at query evaluation time. As we will see, this will result in an interleaving of query optimization and query execution.

This section is divided into three parts that illustrate the optimization of (i) simple restrictions, (ii) self joins (i.e. those involving twice the same extent) and (iii) general joins.

4.1 Simple Restrictions

We consider the optimization of equality and membership predicates. In both cases, we are interested in predicates over the attribute of a variable. Note that these are the most common ones.

Equality Predicate Let us consider the following example. We have an object or a persistent variable with name *peter*, and we are interested in all the employees that have *peter* as a boss. We use the name *peter* to simplify the query. However, any constant subquery would serve our purposes.

> **select** e
> **from** Employee* e
> **where** e.boss = peter

Having no indices to rely on, the best evaluation plan one can come up with is

$$\text{select}[boss{=}peter](Employee \cup Manager \cup CEO)$$

But obviously,

> if *peter* points to an object which belongs to the class *Employee* (and nothing more), then the answer to the query is the empty set.

This is because the attribute *boss* was defined to be of type *Manager*. Hence, no instance of *Employee* can be a boss. This kind of inference still seems a little rough. It will be refined later on.

The query optimizer can decide the emptiness of the query result immediately without accessing any of the extents *Employee*, *Manager*, and *CEO*. The only thing to be done is to dereference *peter* at query optimization time and then perform the type inference using the dynamic type *Employee* of *peter*. Note that at the schema level, there is no way to tell by using the static types only, that the answer to the query will be the empty set.

This type of query optimization is easiest implemented within an interpreter. In the case of a compiler, additional code has to be generated for the type check and the decision on what plan to execute depending on the outcome of the type check. The interpreter has to evaluate queries partially and use the resulting type information to simplify the remaining expression. A compiler could be implemented such that it generates different alternative plans depending on the possible outcome of the derived type restrictions, and then generates an according *chose-plan* operator [8] tying the different plans together.

Many of the subsequent examples are of the same flavor in that dynamic type information is used to simplify the query. Hence, we subsequently assume that the query optimizer is modified such that this information can be taken care of.

Let us now resume the discussion of the above restriction. What can be do if *peter* is a *CEO*? In general or without any more information, nothing can be done. But consider the following case. In object schemas, attributes can be refined. Most systems restrict refinement to either covariance or contravariance. Let us assume that the system at hand supports covariance and that the attribute *boss* is refined to be of type *CEO* for *Manager*.

Again, static type inference does not help much. Even though, the bosses of the employees are restricted to be managers, they are only restricted to be at least managers. There is nothing that forbids them to be CEOs. Nonetheless, in practice we would expect the (direct) bosses of employees to be managers and not CEOs. Assume for a moment, that this information is available. Then, for all objects e of class *Employee*, we would have that $e.boss$ is of class *Manager* which is a superclass of *CEO*. Hence, $e.boss = peter$ cannot become true for $e \in Employee$. That is, we only have to scan the extents of *Manager* and *CEO*, saving the scan over *Employee*. The resulting query evaluation plan is

$$\text{select}[boss{=}peter](Manager \cup CEO)$$

Note that, in this example, it is likely that the scan over *Employee* (that we eliminate) is by far the most expensive, since there are typically more employees than managers and CEOs.

Clearly, the information we used — all bosses of employees are simple managers and not CEOs — is not readily available. There exist two possibilities to get hold on this information. First, similar to materializing the maximum and minimum value of a scalar attribute, one materializes the highest and the lowest class of the attribute values of a class valued attribute. That is, redundant information is introduced.

One might object that the problem with this approach is the incurred update penalty. But this is not necessarily true. First, if this kind of information is hold only for attributes that do not change frequently, like for the *boss* attribute, then the query speed up by far outweighs the additional update costs. Second, note that not every update leads to an update of the aggregated type information. Only if the type of the attribute value changes, an update of the aggregated type information is necessary.

Furthermore, it is our opinion, that this is a very good argument to introduce indexes solely based on the type of the attributes instead of their values. This would even increase the possibilities of the query optimizer to optimize queries. In case of the example, there might exist some exceptions such that an employee is directly managed by a CEO but these would then easily be detected by using the index. Although this kind of index is very useful for query optimization, we do not elaborate on their possible realizations in the current paper but leave it for future research.

Membership Predicate Let us consider the following query that selects all employees having poor bosses.

```
select  e1
from    Employee* e1
where   e1.boss in select  e2
                   from    Employee* e2
                   where   e2.salary < 100.000
```

The traditional evaluation, without type inference, would have to consider the union of all the strict extents *Employee*, *Manager*, and *CEO* as a range for both, $e1$ and $e2$. Applying the techniques of this section, we can do better:

- By using static type inference, we can first eliminate the extent of *Employee* from the range of $e2$, since a *boss* must be at least a manager.

Let us assume that the evaluation of the subquery returns only managers, since CEOs are all rich. Then,

- if we have the type information that managers have only CEOs as bosses (see above), we eliminate the *Manager* and *CEO* from the range of $e1$.

Suppose that the system does not maintain this information. Then, we cannot avoid the scans on *Manager* and *CEO*. However, we may further optimize the query by using dynamic type information at evaluation time. The idea is to evaluate *e1.boss* and perform a type check before the actual predicate is to be evaluated. In the case of equality for two simple values, this would of course not pay: instead of one equality test for the value only, we have now two equality tests, one for the type and one for the value. However, in the present case this can very well pay. Indeed, by performing a very simple type-based test, we may avoid evaluating an expensive membership predicate. This is what we call *type-based pretests*. Note that type-based pretests can be added very easily to the implementations of restriction or join operations.

Last, note that if we would have the *max* aggregate type information available on the salary of each subextent of *Employee*, then CEOs could be excluded from the range of $e2$ immediately.

Summarizing, using type informations we only have to consider *Employee* as the range for $e1$, and *Manager* as the range for $e2$. Thus, we save

- scanning the extent *CEO*,
- the evaluation of *e1.boss* for *Manager*,
- the evaluation of *e1.boss* for *CEO*,
- the evaluation of *e2.salary* for *Employee*, and

- the evaluation of *e2.salary* for *CEO*.

Note in the case of logical extents, the latter evaluations of the attribute values of e_1 and e_2 result in dereferencing the according object identifiers and, hence, might result in additional page faults — which we save.

4.2 Joining an Extent with Itself

The following query retrieves those employees that have the same boss.

select	e1, e2
from	Employee* e1, Employee* e2
where	e1.boss = e2.boss

Applying input splitting results in the nine joins

Employee	$\bowtie$	Employee
Employee	$\bowtie$	Manager
Employee	$\bowtie$	CEO
Manager	$\bowtie$	Employee
Manager	$\bowtie$	Manager
Manager	$\bowtie$	CEO
CEO	$\bowtie$	Employee
CEO	$\bowtie$	Manager
CEO	$\bowtie$	CEO

By dynamic type inference for *e1.boss* and *e2.boss* for each of the (sub-)extents of *Employee*, four joins can be eliminated immediately. For example, an employee and a manager can never have the same boss (due to the refinement). Hence, the joins *Employee* $\bowtie$ *Manager* and *Manager* $\bowtie$ *Employee* can be eliminated. The same argument applies to employees and CEOs. Note that even if we have to get this information by scanning the extents (as we have to do anyway), the elimination of these four joins will considerably reduce the costs.

Hence, only the following five joins are left to be computed:

Employee	$\bowtie$	Employee
Manager	$\bowtie$	Manager
Manager	$\bowtie$	CEO
CEO	$\bowtie$	CEO
CEO	$\bowtie$	Manager

Further, if we apply reasoning by symmetry, we can further reduce the number of joins to be computed. The remaining ones are

Employee	$\bowtie$	Employee
Manager	$\bowtie$	Manager
Manager	$\bowtie$	CEO
CEO	$\bowtie$	CEO

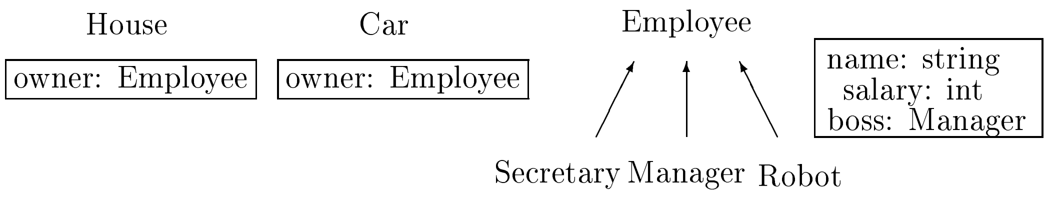

Figure 3: New Simple Schema

Hence, only four instead of nine joins have to be computed. That is, we save more than 50% of the join execution cost. Since the join operation is the most expensive operation, this is considerable.

4.3 General Joins

Let us change our example to that of Figure 3. Again, boxes contain attribute definitions. The two classes *Car* and *House* reference the class *Employee* which, this time, has three subclasses and no attribute redefinitions.

The following query selects pairs of houses and cars having the same owner:

select h, c
from House h, Car c
where h.owner = c.owner **and**
h.owner.salary > 100.000

Let us define

$$C|_{a \in C'} := \{c \in C | c.a \in C'\}$$

to be the objects of class C whose value for attribute a is of class C'. Hence, we can partition an extent into several subsets depending on the type of the attribute. If this partition of an extent is not available by a special index on the type of a, it has to be determined dynamically. The sequel shows that even if we have to get the information dynamically, partitioning may be interesting. Note that (attribute-type-based) partitioning is similar in spirit to hash-based partitioning. But the former not only exhibits the same advantages as the latter but also some more (see below).

Applying the above definition, we can partition the original join $House \bowtie Car$ into the following 16 joins

$$
\begin{array}{lcl}
\text{House}|_{owner \in Employee} & \bowtie & \text{Car}|_{owner \in Employee} \\
\text{House}|_{owner \in Employee} & \bowtie & \text{Car}|_{owner \in Manager} \\
\text{House}|_{owner \in Employee} & \bowtie & \text{Car}|_{owner \in Secretary} \\
\text{House}|_{owner \in Employee} & \bowtie & \text{Car}|_{owner \in Robot} \\
\text{House}|_{owner \in Manager} & \bowtie & \text{Car}|_{owner \in Employee} \\
\text{House}|_{owner \in Manager} & \bowtie & \text{Car}|_{owner \in Manager} \\
\text{House}|_{owner \in Manager} & \bowtie & \text{Car}|_{owner \in Secretary} \\
\text{House}|_{owner \in Manager} & \bowtie & \text{Car}|_{owner \in Robot} \\
\text{House}|_{owner \in Secretary} & \bowtie & \text{Car}|_{owner \in Employee} \\
\text{House}|_{owner \in Secretary} & \bowtie & \text{Car}|_{owner \in Manager} \\
\text{House}|_{owner \in Secretary} & \bowtie & \text{Car}|_{owner \in Secretary} \\
\text{House}|_{owner \in Secretary} & \bowtie & \text{Car}|_{owner \in Robot} \\
\text{House}|_{owner \in Robot} & \bowtie & \text{Car}|_{owner \in Employee} \\
\text{House}|_{owner \in Robot} & \bowtie & \text{Car}|_{owner \in Manager} \\
\text{House}|_{owner \in Robot} & \bowtie & \text{Car}|_{owner \in Secretary} \\
\text{House}|_{owner \in Robot} & \bowtie & \text{Car}|_{owner \in Robot}
\end{array}
$$

out of which only the following 4 survive:

$$
\begin{array}{lcl}
\text{House}|_{owner \in Employee} & \bowtie & \text{Car}|_{owner \in Employee} \\
\text{House}|_{owner \in Manager} & \bowtie & \text{Car}|_{owner \in Manager} \\
\text{House}|_{owner \in Secretary} & \bowtie & \text{Car}|_{owner \in Secretary} \\
\text{House}|_{owner \in Robot} & \bowtie & \text{Car}|_{owner \in Robot}
\end{array}
$$

We just eliminated those joins which for sure result in the empty set. For example, we know that

$$House|_{owner=Employee} \bowtie Car|_{owner=Manager}$$

results in the empty set since the equality of two owners trivially implies the equality of their respective types. In general, if we have c possible classes for an attribute value, only c joins have to be performed. The rest of the c^2 joins results in empty sets.

The question now is in how far does this kind of saving in the number of joins to be executed result in savings of actual execution time. More specifically: Are the execution costs for the remaining four joins, including the partitioning costs of the input into the different classes faster to execute than the single join $Employee* \bowtie Employee*$?

Let us first consider a theoretical argument. Further let us assume, that the class hierarchy rooted at C of the join attribute's class C consists of c classes containing exactly n/c elements each. Then, the join costs depend on the actual join algorithm but the in-memory costs can be assumed to be proportional to the following expressions:

join algorithm	cost for C^*	cost for c small joins
nested-loop	n^2	$c(n/c)^2$
sort-merge	$n \lg n$	$c(n/c \lg n/c)$
hash	n	$c(n/c)$

if we assume that the result size is smaller than n. Now, the factors we save can be estimated as follows:

join algorithm	expected saving factor
nested-loop	$c(n/c)^2/n^2 = 1/c$
sort-merge	$c(n/c \lg n/c)/(n \lg n) = 1 - \lg c/\lg n$
hash	$c(n/c)/n = 1$

Hence, the largest saving is for the nested-loop algorithm, some small savings are still available for the sort-merge join, and no saving exists for the hash join.

Nevertheless, in practice the results will differ for two reasons:

1. Overhead
   The higher the number of joins, the higher the overhead of their initialization and unioning of the result (although this union does not involve duplicate elimination).

   For example, we would expect c small hash joins to be slower than the one big hash join.

2. Paging
   The arguments of the smaller joins will fit better into the available buffer than the big arguments for the big join. We would expect less paging and, hence, less execution time spent for the smaller joins, if the size of the arguments exceeds a certain threshold. This becomes even more true for the hash join since its access to pages is — opposed to the sort-merge join — rather random.

Since the proportion of the overhead becomes smaller for larger joins, we would expect a break-even point between the two execution plans for hash join. This is illustrated in Fig. 4 where the execution times for different join algorithms was measured for $c = 4$ classes where each class contains exactly $n/4$ elements.

Let us know resume the optimization of the example. So far, we have only applied type-based partitioning which is similar in spirit to hash partitioning. Nevertheless, applying type-based partitioning allows us to incorporate more knowledge into the optimization process of the single partitions. If we assume that only employees and managers earn more than 100.000 and that this knowledge is available through min/max statistics on the extents, we can eliminate two further joins resulting in

$$\begin{array}{lcl} \text{House}|_{owner \in Employee} & \bowtie & \text{Car}|_{owner \in Employee} \\ \text{House}|_{owner \in Manager} & \bowtie & \text{Car}|_{owner \in Manager} \end{array}$$

In case *Employee* is a virtual class, we are done with

$$\text{House}|_{owner \in Manager} \quad \bowtie \quad \text{Car}|_{owner \in Manager}$$

Last, we have to introduce the selection. Assuming that all managers earn more than 100.000, no selection is needed. Assume we have logical extents. Then, saving this selection saves accessing the *Manager* objects.

Remarks The above discussion was based on attribute values. If we replace these attributes and their type by methods and their result type, the above optimization techniques still remain valid. That is, the argument types of a method can just be neglected.

n	nl	nl (P)	sm	sm (P)	hl	hl (P)
256	0.05	0.02	0.02	0.02	0.02	0.02
512	0.19	0.06	0.03	0.03	0.02	0.03
1024	0.95	0.34	0.07	0.06	0.03	0.04
2048	3.78	1.52	0.16	0.15	0.06	0.07
4096	15.21	6.03	0.34	0.33	0.11	0.13
8192	60.66	24.39	0.75	0.73	0.23	0.25
16384	242.61	97.38	1.66	1.62	0.49	0.63
32768	969.50	382.43	3.83	3.48	1.16	1.37
65536	-	-	8.38	7.82	2.83	3.89
131072	-	-	20.13	17.88	9.17	8.27
262144	-	-	49.86	39.38	25.95	17.16

Figure 4: Sample experimental result to illustrate the effect of partitioning

The execution times for nested-loop (nl), sort-merge (sm), and hash-loop (hl) join algorithms are given in seconds. The variant where partitioning was applied is marked with an additional "P". Due to its tremendous run-time, the nested loop algorithm was not measured for all extent sizes n.

Furthermore, since the cost of evaluating a method can be far higher than the evaluation of a simple comparison, the saving is higher. Additionally to saving scan and join operations, one is able to save method applications.

As a last remark in this section let us note that although we dealt with refinement under covariance the analogous optimization techniques exist for contravariance.

5 Conclusion

We presented several techniques to optimize queries that span a hierarchy of class extents. The main idea is to make the (sub-) extents visible to the optimizer and to apply traditional optimization techniques, reasoning about symmetry, and type inference to yield highly optimized execution plans.

In order to apply some of our optimization techniques more successfully, we relied on the possible types of the values of an attribute. One way to yield this information was to dynamically check types. Another was to rely on type aggregate information or type based indices. We did not elaborate on how to maintain this kind of information efficiently. Hence, some future research in this direction is necessary.

References

[1] J. Banerjee, W. Kim, and K.-C. Kim. Queries in object-oriented databases. MCC Technical Report DB-188-87, MCC, Austin, TX 78759, June 1987.

[2] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledge and Data Engineering*, 1(2):196–214, Jun 1989.

[3] A. Biliris and E. Panagos. EOS user's guide. Technical Report Release 2.2, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1994.

[4] R. Cattell, editor. *The Object Database Standard: ODMB-93*. Morgan Kaufmann, 1994. Release 1.1.

[5] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392, 1992.

[6] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, 1993.

[7] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. Technical Report 95-5, RWTH-Aachen, 1995.

[8] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 358–366, 1989.

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

[10] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proc. Int. Conf. on Extended Database Technology (EDBT)*, pages 169–187, Venice, 1990.

[11] A. Kemper, C. Kilger, and G. Moerkotte. Materialization in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1991.

[12] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 364–374, 1990.

[13] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. on Very Large Data Bases*, pages 294–305, 1990.

[14] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Disjunctive queries in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1994.

[15] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimization of boolean expressions in object bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 79–90, 1992.

[16] C. Kilger and G. Moerkotte. Indexing multiple sets. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 180–191, Santiago, Chile, Sept. 1994.

[17] W. Kim, K. C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394, Massachusetts, 1989. Addison Wesley.

[18] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proc. IEEE Intl. Workshop on Object-Oriented Database Systems, Asilomar, Pacific Grove, CA*, pages 171–182. IEEE Computer Society Press, 1986.

[19] H. Steenhaben, P. Apers, H. Blanken, and R. de By. From nested-loop to join queries in oodb. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 618–629, 1994.

[20] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigation in object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 522–533, 1994.