Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases*

J. Claussen[1] *A. Kemper*[1] *G. Moerkotte*[2] *K. Peithner*[1]

[1]Universität Passau
Lehrstuhl für Informatik
94030 Passau, Germany
⟨lastname⟩@*db.fmi.uni-passau.de*
http://www.db.fmi.uni-passau.de/

[2]Universität Mannheim
Lehrstuhl für Praktische Informatik III
68131 Mannheim, Germany
moer@pi3.informatik.uni-mannheim.de
http://pi3.informatik.uni-mannheim.de/

Abstract

We investigate the optimization and evaluation of queries with universal quantification in the context of the object-oriented and object-relational data models. The queries are classified into 16 categories depending on the variables referenced in the so-called *range* and *quantifier predicates*. For the three most important classes we enumerate the known query evaluation plans and devise some new ones. These alternative plans are primarily based on anti-semijoin, division, generalized grouping with count aggregation, and set difference. In order to evaluate the quality of the many different evaluation plans a thorough performance analysis on some sample database configurations was carried out. The quantitative analysis reveals that—if applicable—the anti-semijoin-based plans are superior to all the other alternatives, even if we employ the most sophisticated division algorithms. Furthermore, exploiting object-oriented features, anti-semijoin plans can be derived even when this is not possible in the relational context.

Universität Passau
Technical Report MIP-9706

*An excerpt of this report appears in *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, Athens, Greece, August 1997.

1 Introduction

There exist only few research papers on optimizing and evaluating queries with universal quantification (see the discussion of related work below). This lack of attention is largely due to the absence of an explicit language construct for universal quantification in SQL-92. In SQL, the database users are forced to "work around" universal quantification by nesting **not exists**-clauses or by formulating the universal quantification as a **count**ing problem. Therefore, most optimizers of commercial DBMS products cannot properly detect the hidden universal quantifications and, as a consequence, generate query evaluation plans that are far from optimal.

We predict that the interest in universal quantification will drastically increase in the near future—basically for three reasons: (1) It is obvious that universal quantification is a very important concept in decision support queries (e.g., finding the suppliers that offer all parts needed for a particular assembly or finding the employees that have all the skills required for a particular project). (2) Language constructs for explicit universal quantification were included in the ODMG standard object query language OQL [Cat96] and are being considered in the SQL-3 standardization [Mel94, <universal clause> in 8.13]. (3) As we will show in this paper, queries with universal quantification can be evaluated very efficiently in "modern" data models that support set/multi-valued references such as the object-oriented model of ODMG [Cat96] or the object-relational models [Sto96].

Existing work on universal quantification is mostly focused on a single facet of the problem: Integration into the query language, equivalences for rewriting or special implementations for operators supporting universal quantification have been discussed. Almost all of the previous work on universal quantification was performed in the context of the pure/flat relational data model. Some work has been done in the object-oriented/object-relational context, e.g. [Ste95], however, only algebraic equivalences were discussed. This paper is—to our knowledge—the first comprehensive treatment of universal quantification from the query language level to the evaluation, including correct treatment of null values.

Graefe and Cole [GC95] give a very thorough account of evaluating relational division. Unfortunately, query evaluation plans based on division are only reasonable for a special class of universally quantified queries, i.e., those for which the quantifier's range constitutes a closed formula. Furthermore, the division is a relational algebra operator tailored for the flat relational model; in a data model supporting multi-valued relationships via set attributes one can usually do much better.

[HP95, RBG96, Car86, WMSB90] propose generalized universal quantifiers in different variations for relational languages, e.g., as SQL extensions. These works are at the conceptual (i.e., language) level except for [RBG96] which includes work on evaluating such generalized quantifiers using special data structures (bit matrices).

Jarke and Koch [JK83] and Bry [Bry89b, Bry89a] devised rules to move selections into the quantifier range definition in order to reduce the number of tuples that have to be evaluated. Steenhagen [Ste95] lists several alternative algebra plans for universally quantified queries.

Dayal [Day83] proposed the graft operator which bears some resemblance with a binary grouping (that we used as one evaluation technique) except that tree scheme occurrences are used as a representation of (intermediate) results. Later, [Day87] proposed the G-Join, G-Aggr and G-Restr. The G-Join replaces the graft operator and a sequence of G-Aggr and G-Restr replaces the previously used prune operator.

[GL87] treated queries with quantification as a special case of nested queries. The quantifiers **exists** and **not exists** are replaced by count aggregations. More recently, Steenhagen et al. [SABdB94] investigated rules for unnesting queries in an object-oriented model.

In this paper we begin with a systematic classification of queries with universal quantification into 16 categories depending on the bound variables of the so-called *range* and *quantifier predicates*. Of these 16 classes we identify the three most important ones. For each of them we enumerate the known query evaluation plans and devise some new ones. Our discussion focuses on "modern" data models with set-valued attributes to represent $N : M$-relationships—such as the object-oriented model or the object-relational model. In such a data model queries with universal quantification can usually be formulated in a much more natural way than in a flat relational model. To see this point let us consider the example of Graefe and Cole's paper: Representing the $N : M$-relationship *enrolled* between *Students* and *Courses* requires a separate relation *Transcript* with *StudentId* and *CourseNo* attributes whereas this relationship can be represented as a set-valued attribute *enrolledCourses* of *Students* in an object-oriented or object-relational schema. In the relational model, finding the Students who have taken all database classes[1] is achieved by the OQL query on the left-hand side of Figure 1. The corresponding query based on

```
select s
from s in Students
where for all c in
    select c
    from c in Courses
    where c.Title like "%database%":
        exists t in
            select t
            from t in Transcript:
                (t.StudentId=s.StudentId
                and t.CourseNo = c.CourseNo)
```

```
select s
from s in Students
where for all c in
    select c
    from c in Courses
    where c.Title like "%database%":
            c in s.enrolledCourses
```

Figure 1: Relational and Object-Oriented Query Variant

an object-oriented or object-relational schema lacks the nested existential quantification which is replaced by a set containment predicate, as shown on the right-hand side of Figure 1. The latter query is certainly more natural to formulate—especially compared to an SQL-92 formulation of the first query which has to be converted into an equivalent, yet obscure formulation with two nested **not exists** clauses. Aside from user friendliness, we will also show that the object-oriented and object-relational models facilitate a much more efficient evaluation of such universally quantified queries.

The remainder of this paper is organized as follows. Section 2 presents our classification of universal quantification queries and example queries for the three most important

[1] We assume that database courses are those courses that contain the string 'database' in the title.

Class-No.	$q()$	$q(e_1)$	$q(e_2)$	$q(e_1, e_2)$
$p()$	1	2	3	4
$p(e_1)$	5	6	7	8
$p(e_2)$	9	10	11	$\boxed{\mathbf{12}}$
$p(e_1, e_2)$	13	14	$\boxed{\mathbf{15}}$	$\boxed{\mathbf{16}}$

Table 1: Classification Scheme According to the Variable Bindings

classes. In Section 3 alternative evaluation plans are presented for the three classes, both in general form and for the example queries. In addition, the treatment of null values is discussed. In Section 4 we introduce rewriting rules for the simplification of the remaining query classes. The rest of the paper is dedicated to a performance analysis: In Section 5 we sketch our query execution engine and the implementation of some special operators. They formed the basis for the experimental evaluation reported in Section 6. A basic cost model is sketched in Section 7. Section 8 concludes the paper with a summary.

2 Classification and Running Example

2.1 Classification

As pointed out in the introduction, the OQL query language of the ODMG standard supports universal quantification. Therefore, we formulate our example queries in OQL.

The prototypical query pattern upon which we base our discussion of universal quantifiers being nested within a query block is

$$
\begin{array}{rcl}
Q & \equiv & \textbf{select } e_1 \\
 & & \textbf{from } e_1 \textbf{ in } E_1 \\
 & & \textbf{where for all } e_2 \textbf{ in select } e_2 \\
 & & \qquad\qquad\qquad\qquad \textbf{from } e_2 \textbf{ in } E_2 \\
 & & \qquad\qquad\qquad\qquad \textbf{where } p: \\
 & & \qquad\qquad\quad q
\end{array}
$$

where p (called the *range predicate*) and q (called the *quantifier predicate*) are predicates in a subset of the variables $\{e_1, e_2\}$. This query pattern is denoted by Q. In a calculus, this query can be stated as follows:

$$Q \equiv \{e_1 \in E_1 \mid \forall e_2 \in E_2 : (p \Rightarrow q)\} \tag{1}$$

Depending on the subset of variables $\{e_1, e_2\}$ that occur in the *range* and *quantifier* predicates $p(\dots)$ and $q(\dots)$ we distinguish 16 classes which are enumerated in Table 1.

In the subsequent discussion we will concentrate on the following three most important classes:

(12) $p(e_2), q(e_1, e_2)$
The range predicate refers to e_2 and the quantifier predicate depends on both, e_1 and e_2.

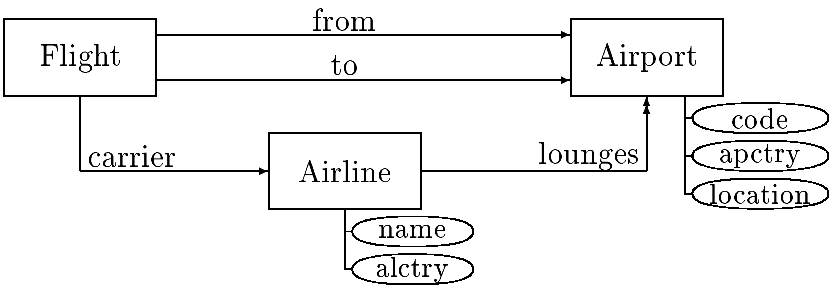


Figure 2: OO Schema of an Airline Reservation System

(15) $p(e_1, e_2), q(e_2)$
The range predicate compares (information of) e_1 and e_2 whereas the quantifier predicate is based on e_2 only.

(16) $p(e_1, e_2), q(e_1, e_2)$
Both, range and quantifier predicates compare (properties of) e_1 and e_2.

Let us briefly contemplate why these are the three most important and—as far as optimization is concerned—also the most difficult classes. If the range predicate p does not refer to variable e_2 the predicate p could be "moved up" to the outer level query block because it is independent of e_2. Basically the same holds for the quantifier predicate q: If it is independent of e_2 the query could be rewritten by pulling up the predicate q into the outer level query block and thereby simplifying the query evaluation. Furthermore, if neither the range predicate p nor the quantifier predicate q refers to e_1 the quantifier subquery is not correlated to the outer level query over E_1 and can be evaluated independently. Classes (12), (15), and (16) constitute all possible query patterns for a correlated quantified subquery in which both the range and the quantifier predicates refer to e_2. The remaining classes are handled in Section 4. There we present simplification rules that allow the reduction of those query classes either to plans similar to those for classes (12), (15), and (16) or to rather simple plans that can be evaluated efficiently.

2.2 Running Example

We want to base the subsequent discussion on the database schema shown in Figure 2. In this schema there are three object types: *Flight*, *Airport*, and *Airline*. The relationships *from* and *to* between *Flight* and *Airport* are single-valued—denoted by single-ended vectors. The relationship *carrier* between *Flight* and *Airline* is also single-valued. We assume that all three relationships are represented by correspondingly named relationships (reference attributes) in the object type *Flight*. The relationship *lounges* between *Airline* and *Airport* is multi-valued and is assumed to be represented as a multi-valued relationship (set-valued attribute) in object type *Airline*.

Example queries for the classes 12, 15, and 16 (cf. Table 1) are stated in Figure 3:

- Query 1 (Class 12) retrieves those airlines that have lounges in all US airports.
- Query 2 (Class 15) retrieves the airlines that do not fly to Libya.

- Query 3 (Class 16) retrieves the airlines that have lounges in all airports of their native country.

3 Alternative Query Evaluation Plans

In this section, we present evaluation plans for the three main query classes. Beforehand, we have to introduce the used algebra operators.

3.1 Algebra Operators

if-Expression For the subsequent evaluation plans, we enhance OQL by an "**if** ... **then** ... **else** ..."-expression. It is useful for rewriting outer restrictions as proposed in [Mur88, SPMK95, CM95a]. At the algebraic level, this is reflected by an algebra operator

$$if_p(E_{true}, E_{false})$$

where E_{true} is the result if p evaluates to *true* and otherwise E_{false} is the result. Actually, the **if**-constructs are much more often used in the simplification rules of Section 4 to optimize the 13 less important classes than in the plans derived for the three important classes.

Miscellaneous Basic Operators As basic operator for reading an object extent we use the notation

$$E[e, A_1, \dots, A_n]$$

for an extent belonging to object type E. It returns tuples consisting of the object identifier e and projects on the (possibly set-valued) attributes $A_1, \dots, A_n$. The algebraic counterpart of the "dot" operator in OQL is the *expand* operator χ, also called, e.g., *materialize* [BMG93]. It may be used both to retrieve attributes and to invoke member functions of a referenced object. In this paper, we only need the attribute access variant (The operator $\circ$ denotes tuple concatenation and g is a newly introduced attribute):

$$\chi_{g\,:\,e.a}(E) \quad := \quad \{e \circ [g\colon e.a] \mid e \in E\}$$

Query 1: Class 12	Query 2: Class 15	Query 3: Class 16
select al.name	**select** al.name	**select** al.name
from al **in** Airline	**from** al **in** Airline	**from** al **in** Airline
where for all ap **in**	**where for all** f **in**	**where for all** ap **in**
(**select** ap	(**select** f	(**select** ap
from ap **in** Airport	**from** f **in** Flight	**from** ap **in** Airport
where ap.apctry = "USA"):	**where** al = f.carrier):	**where** ap.apctry = al.alctry):
ap **in** al.lounges	f.to.apctry != "Libya"	ap **in** al.lounges

Figure 3: Example OQL Queries With Universal Quantification

To flatten (unnest) set-valued attributes we use the unnest operator μ. Applied on an object type E with a set of attributes A and a set-valued attribute $a \notin A$, it introduces a new atomic attribute g:

$$\mu_{g:a}(E[A, a]) \quad := \quad \{e_1.[A] \circ [g : e_2] \mid e_1 \in E, e_2 \in e_1.a\}$$

Furthermore, a scalar aggregation $\mathbf{count}(E)$ is used to calculate the cardinality of a collection E.

Division Relational division $E_1[A_1, A_2] \div E_2[A_2]$ is defined as follows, cf. [Mai83]:

$$E_1 \div E_2 := \{t | t \in \pi_{A_1}(E_1) \wedge (\{t\} \times E_2) \subseteq E_1\}$$

Anti-Semijoin The anti-semijoin is defined as the complement of the semijoin operator, cf. also [Gra93, Bry89b, RGL90]:

$$E_1 \,\bar{\ltimes}_p\, E_2 := \{e_1 | e_1 \in E_1 \wedge \neg \exists e_2 \in E_2 \colon p(e_1, e_2)\}$$

Binary Grouping The binary grouping operator Γ [CM95a] is similar to a join where the intermediate result is nested. That is, for every tuple in the left (outer) operand, a set of matching tuples from the right (inner) operand is constructed. This leads to more efficiency [RRS91] due to a smaller representation of the intermediate result. The nestjoin operator as defined in [SABdB94] has similar functionality. While the nestjoin applies a function to each element before it is added to the set, the binary grouping operator Γ may evaluate a function on the resulting group, replacing the group by the result value and thus further diminishing its size (e.g., in case of an aggregate function).

The binary grouping operator is defined as follows, cf. [CM95a]:

$$E_1[e_1] \; \Gamma_{g;p;f} \; E_2[e_2] \quad := \quad \{e_1 \circ [g : G] | e_1 \in E_1, G = f(\{e_2 | e_2 \in E_2 \wedge p\})\} \tag{2}$$

For each tuple e_1 of E_1, the inner relation E_2 is selected by p, is mapped by f, and is assigned to the new attribute g.

3.2 Alternative Evaluation Plans for the Most Important Cases

Let us now enumerate alternative query evaluation plans for the three most important query classes 12, 15, and 16. For illustration, we present the concrete plans for the three example queries in parallel.

3.2.1 Query Class 12: $p(e_2)$, $q(e_1, e_2)$

Division This is the principal case for applying the relational division operator (see, e.g., [Nak90] and [GC95]):

$$\mathit{if}_{\sigma_{p(e_2)}(E_2[e_2]) \neq \emptyset}\big((E_1[e_1] \bowtie_{q(e_1,e_2)} E_2[e_2]) \div \sigma_{p(e_2)}(E_2[e_2]), E_1[e_1]\big) \tag{3}$$

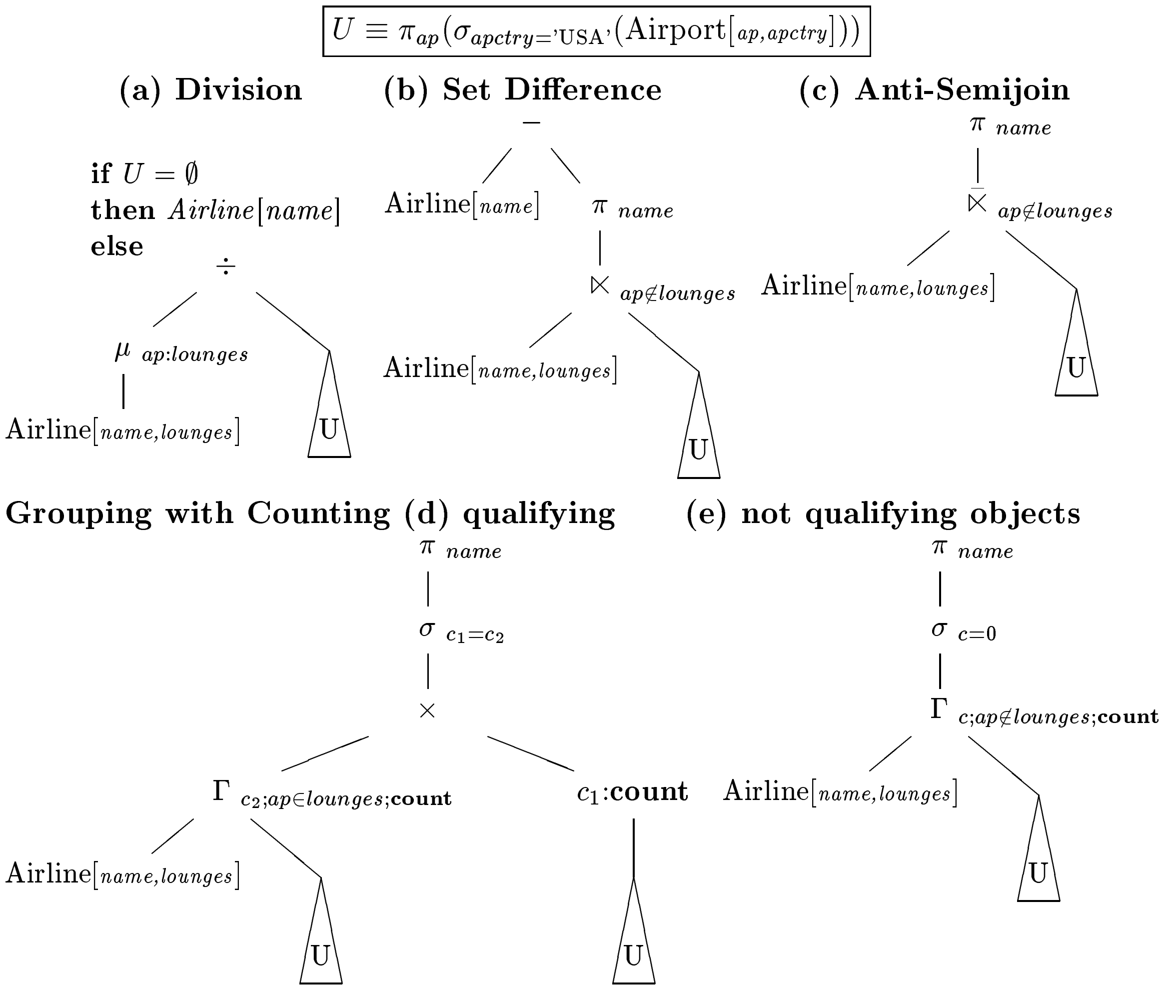


Figure 4: Evaluation Plans for Query 1 (Class 12)

If the selection $\sigma_{p(e_2)}(E_2[e_2])$ yields at least one object we can also apply the predicate p to the dividend. We obtain the following expression:

$$if_{\sigma_{p(e_2)}(E_2[e_2])\neq\emptyset}\big((E_1[e_1] \ltimes_{q(e_1,e_2)} \sigma_{p(e_2)}(E_2[e_2])) \div \sigma_{p(e_2)}(E_2[e_2]), E_1[e_1]\big) \tag{4}$$

If the predicate $q(e_1, e_2)$ is of the form $e_2 \in e_1.SetAttribute$—as will most often be the case in an object-oriented or object-relational schema—the join can be replaced by an unnest (μ) operator (see also the plan for query 1 in Figure 4a):

$$if_{\sigma_{p(e_2)}(E_2[e_2])\neq\emptyset}\big(\mu_{e_2:SetAttribute}(E_1[e_1, SetAttribute]) \div \sigma_{p(e_2)}(E_2[e_2]), E_1[e_1]\big) \tag{5}$$

Set Difference Using set difference, the translation is

$$E_1[e_1] - \pi_{e_1}\big((E_1[e_1] \times \sigma_{p(e_2)}(E_2[e_2])) - (E_1[e_1] \ltimes_{q(e_1,e_2)} \sigma_{p(e_2)}(E_2[e_2]))\big) \tag{6}$$

This may be optimized to

$$E_1[e_1] - \big(E_1[e_1] \overline{\ltimes}_{\neg q(e_1,e_2)} \sigma_{p(e_2)}(E_2[e_2])\big) \tag{7}$$

This plan is mentioned, e.g., in [Ste95], however using a regular join instead of a semijoin.

Anti-Semijoin The anti-semijoin can be employed to eliminate the set difference yielding the following plan (A similar plan—without range predicate—was proposed in [Ste95]):

$$E_1[e_1] \;\bar{\ltimes}_{\neg q(e_1,e_2)}\; \sigma_{p(e_2)}(E_2[e_2]) \tag{8}$$

The plan depends on the uniqueness of e_1, i.e., the attribute(s) e_1 must be a (super) key of E_1. This is especially fulfilled in the object-oriented context if e_1 constitutes the object identifier (OID).

Grouping with Count Aggregation A common approach to express universal quantification in SQL is counting. In the following evaluation plan resembling the **count**ing SQL query in Subsection 6.3 (page 23), c_1 materializes the number of objects satisfying the range predicate. On the left-hand side, for each $e_1^i \in E_1$ the number of objects in E_2 satisfying both the range and quantifier predicate is counted and materialized in c_2^i. The objects of E_1 with equal count values c_1 and c_2, i.e., the quantifier predicate is fulfilled for all elements of the range, qualify.

$$\Pi_{e_1}\big(\sigma_{c_1=c_2}\big(\underbrace{(E_1[e_1]\;\Gamma_{c_2;q(e_1,e_2);\mathbf{count}}\;\sigma_{p(e_2)}(E_2[e_2]))}_{\{[e_1^1,c_2^1],\ldots,[e_1^n,c_2^n]\}} \times \{[c_1 : \mathbf{count}(\sigma_{p(e_2)}(E_2[e_2]))]\}\big)\big) \tag{9}$$

Plan (10) is an optimization of (9). Instead of counting matches and comparing with the range count, mismatches are counted.

$$\Pi_{e_1}\big(\sigma_{c=0}(E_1[e_1]\;\Gamma_{c;\neg q(e_1,e_2);\mathbf{count}}\;\sigma_{p(e_2)}(E_2[e_2]))\big) \tag{10}$$

Actually, as we will see in the quantitative evaluation, plan (10) may be more costly than (9) due to the negation of the quantifier predicate q which may prevent the application of efficient join methods, e.g., hash join.

3.2.2 Query Class 15: $p(e_1, e_2)$, $q(e_2)$

Division The division operator is not directly applicable for this class of universal quantification queries. The division can only be applied if the divisor constitutes a closed formula not dependent on the dividend. Here, the quantifier's range formula $\sigma_{p(e_1,e_2)}(E_2[e_2])$ is obviously not closed since it has the free variable e_1 depending on the outer level query over E_1.

According to the reduction algorithm of [Cod72] a division plan would be

$$(E_1[e_1] \bowtie_{\neg p(e_1,e_2)\vee q(e_2)} E_2[e_2]) \div E_2[e_2] \tag{11}$$

This plan is certainly not competitive because typically p would be a selective predicate. Thus the join in (11) can be expected to produce almost the cartesian product. Therefore, this plan was not further considered in the quantitative evaluation.

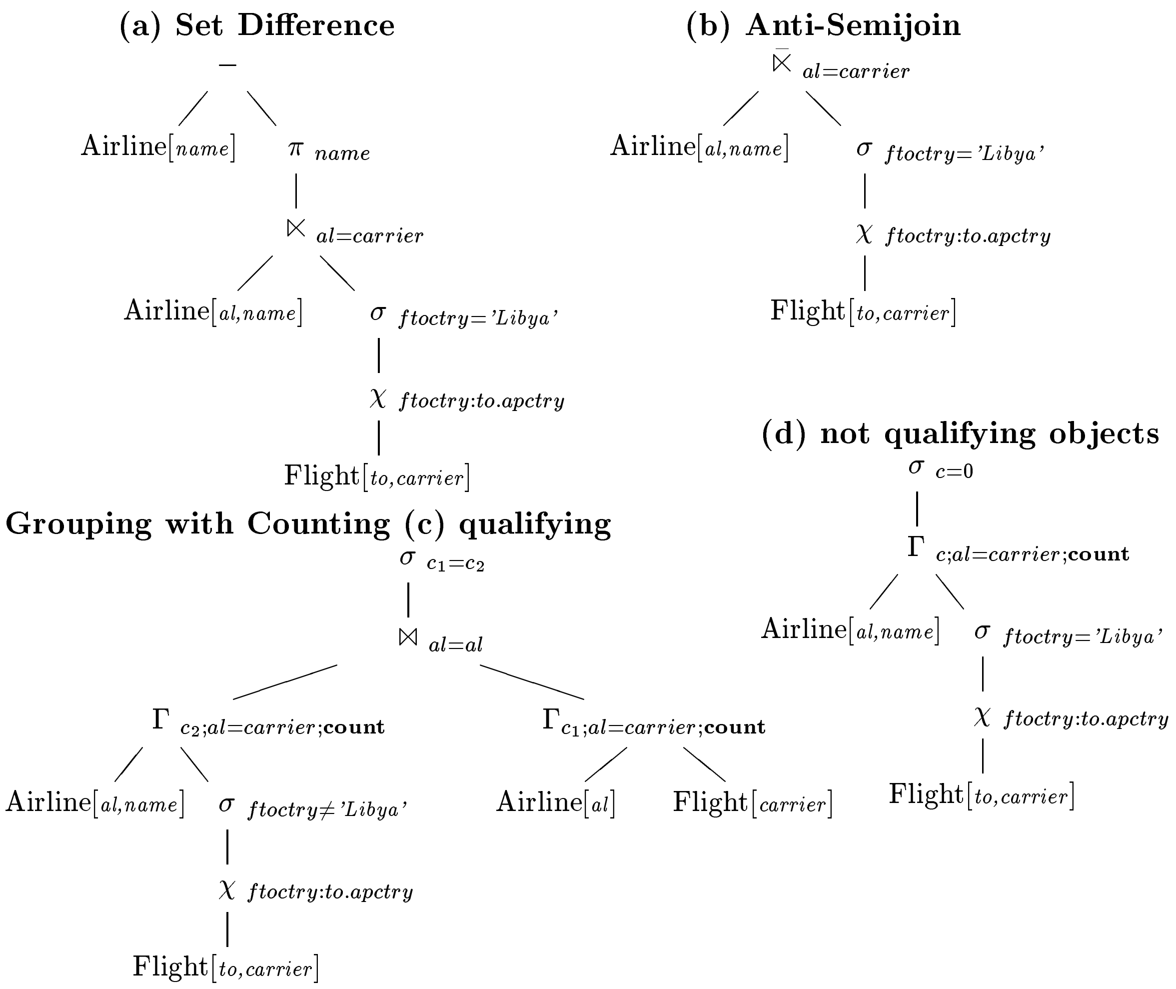


Figure 5: Evaluation Plans (Top-Level Projections Omitted) for Query 2 (Class 15)

Set Difference The set difference plan is

$$E_1[e_1] - \pi_{e_1}\big((E_1[e_1] \bowtie_{p(e_1,e_2)} E_2[e_2]) - (E_1[e_1] \bowtie_{p(e_1,e_2)} \sigma_{q(e_2)} E_2[e_2])\big) \tag{12}$$

Negating the quantifier predicate q and thus eliminating the inner difference results in the following plan (cf. Figure 5a for the plan for query 2):

$$E_1[e_1] - \big(E_1[e_1] \ltimes_{p(e_1,e_2)} \sigma_{\neg q(e_2)}(E_2[e_2])\big) \tag{13}$$

Anti-Semijoin The above "set difference" form can easily be transformed into an equivalent—and obviously more efficient—anti-semijoin formulation:

$$E_1[e_1] \; \bar{\ltimes}_{p(e_1,e_2)} \; \sigma_{\neg q(e_2)}(E_2[e_2]) \tag{14}$$

It is also possible to move the predicate $\neg q(e_2)$ into the anti-semijoin predicate—thereby creating a conjunctive join predicate. Again, the uniqueness constraint of e_1 as described for plan (8) applies.

Grouping with Count Aggregation

$$\Pi_{e_1}\Big(\sigma_{c_1=c_2}\big((E_1[e_1]\ \Gamma_{c_2;p(e_1,e_2);\mathbf{count}}\ \sigma_{q(e_2)}(E_2[e_2])) \bowtie (E_1[e_1]\ \Gamma_{c_1;p(e_1,e_2);\mathbf{count}}\ E_2[e_2])\big)\Big) \tag{15}$$

Let us explain the above plan from right to left. In the right-hand side's binary grouping, for each object $e_1 \in E_1$ the number of objects in the quantifier's range is counted and materialized in attribute c_1. In the left-hand side's binary grouping, for each object of E_1 the number of objects of E_2 that are in the qualifier's range **and** satisfy the qualifier predicate is counted in attribute c_2. The two relations are joined on object identity—i.e., on equal e_1-attributes—and then the values c_1 and c_2 are compared in the selection predicate. Equal count values guarantee that the corresponding object $e_1 \in E_1$ qualifies.

The above plan appears to be rather inefficient in comparison to the anti-semijoin plan because it determines the quantifier's range twice. There are two possible optimizations: we could factor out the range computation or, as we do in the next plan, we could collapse the two groupings into one by negating the quantifier predicate.

$$\Pi_{e_1}\big(\sigma_{c=0}\big(E_1[e_1]\ \Gamma_{c;p(e_1,e_2);\mathbf{count}}\ \sigma_{\neg q(e_2)}(E_2[e_2])\big)\big) \tag{16}$$

This plan is very similar to the anti-semijoin plan except that an object $e_1 \in E_1$ is not discarded as soon as the first disqualifying object $e_2 \in E_2$ is encountered; rather the number of objects of E_2 that disqualify e_1 is counted. Therefore, the plan does more work than is needed and, as a consequence, cannot be better than the anti-semijoin plan.

3.2.3 Query Class 16: $p(e_1, e_2)$, $q(e_1, e_2)$

Division Here, again, the range predicate depends on the outer level variable e_1. A valid division plan looks similar to the one for case 15.

Set Difference A translation using set difference is (see also Figure 6)

$$E_1[e_1] - \pi_{e_1}\big((E_1[e_1] \bowtie_{p(e_1,e_2)} E_2[e_2]) - (E_1[e_1] \bowtie_{p(e_1,e_2)\wedge q(e_1,e_2)} E_2[e_2])\big) \tag{17}$$

Anti-Semijoin The above query evaluation plan based on set difference can also be formulated as an equivalent anti-semijoin plan. First, the difference of the two join expressions can be replaced by a semijoin:

$$E_1[e_1] - \big(E_1[e_1] \ltimes_{p(e_1,e_2)\wedge\neg q(e_1,e_2)} E_2[e_2]\big)$$

Finally, the remaining set difference is transformed into an anti-semijoin which also "covers" the semijoin:

$$E_1[e_1]\ \bar{\ltimes}_{p(e_1,e_2)\wedge\neg q(e_1,e_2)}\ E_2[e_2] \tag{18}$$

The uniqueness constraint of e_1 applies as discussed before (cf. plan (8)).

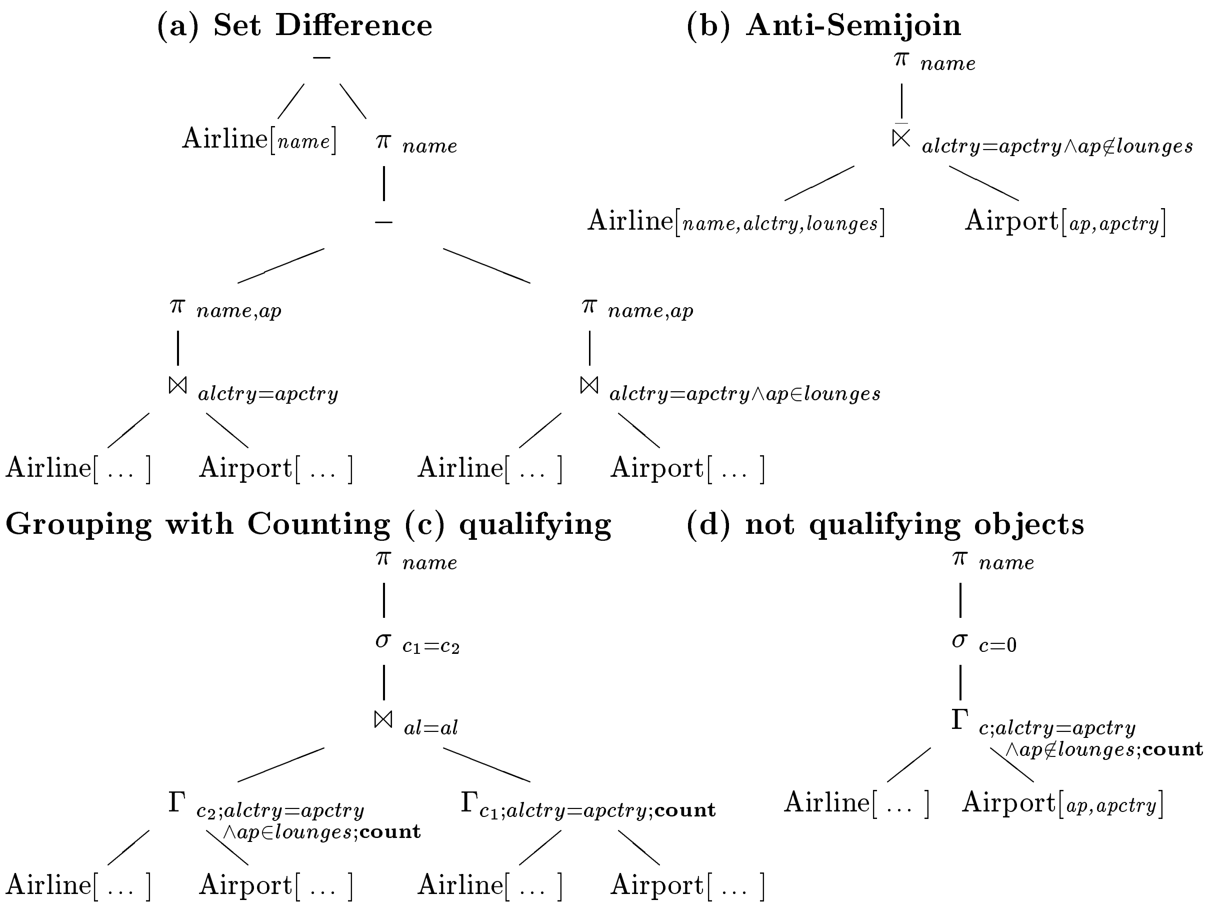


Figure 6: Evaluation Plans for Query 3 (Class 16)

Grouping with Count Aggregation The plans are basically the same as those devised for query class 15 above. However, the quantifier predicate $q(e_1, e_2)$ cannot be evaluated beforehand by a selection on E_2 but is transferred into the grouping predicate by a conjunction:

$$\Pi_{e_1}\Big(\sigma_{c_1=c_2}\big((E_1[e_1]\ \Gamma_{c_2;p(e_1,e_2)\wedge q(e_1,e_2);\mathbf{count}}\ E_2[e_2]) \bowtie (E_1[e_1]\ \Gamma_{c_1;p(e_1,e_2);\mathbf{count}}\ E_2[e_2])\big)\Big) \quad (19)$$

$$\Pi_{e_1}\big(\sigma_{c=0}\big(E_1[e_1]\ \Gamma_{c;p(e_1,e_2)\wedge\neg q(e_1,e_2);\mathbf{count}}\ E_2[e_2]\big)\big) \quad (20)$$

3.3 Null Values

In this subsection, we will revisit our equivalences under the aspect of *unknown* attribute values. The ODMG standard [Cat96] provides null values only for object references (*nil* references). Since null values are, however, integral part of SQL, we will assume SQL semantics [MS93] for null values, i.e., we use a three-valued logic with a third value *unknown*. In this three-valued logic the truth value of (*true*∧*unknown*) is *unknown*, of (*false*∧*unknown*) is *false*, of (*true*∨*unknown*) is *true*, of (*false*∨*unknown*) is *unknown*, and

($\neg$*unknown*) is *unknown*. An object qualifies for a subquery if the value of the selection predicate is *true*; an *unknown* value of the query predicate is implicitly mapped to *false*.

In the presence of null values the semantics of the OQL query

$$\textbf{select } e_1 \textbf{ from } e_1 \textbf{ in } E_1 \textbf{ where for all } e_2 \textbf{ in } \textbf{select } e_2 \textbf{ from } e_2 \textbf{ in } E_2 \textbf{ where } p\text{: } q$$

has to be refined to the following calculus formula:

$$Q \equiv \{e_1 \in E_1 \mid \forall e_2 \in \{e_2 \in E_2 \mid p(e_1, e_2)\} \colon q(e_1, e_2)\} \tag{1'}$$

Note that in the presence of null values this expression has a different semantic than the previously (on page 4) stated calculus formula

$$Q \equiv \{e_1 \in E_1 \mid \forall e_2 \in E_2 : (p \Rightarrow q)\} \tag{1}$$

Take a fixed object $e_1' \in E_1$ and consider an object $e_2' \in E_2$ for which $p(e_1', e_2')$ evaluates to *unknown*. According to (1′) the object e_2' is discarded from the range such that the outcome of $q(e_1', e_2')$ is irrelevant for the "fate" of e_1'. However, in the calculus formula (1) the entire predicate $p(e_1', e_2') \Rightarrow q(e_1', e_2')$ with the standard meaning $\neg p(e_1', e_2') \vee q(e_1', e_2')$ is evaluated. Therefore, if $q(e_1', e_2')$ evaluates to *false* or *unknown* the composite predicate $p \Rightarrow q$ evaluates to *unknown*—given that $p(e_1', e_2')$ was *unknown*. Consequently, e_1' is discarded from the result.

In order to enforce the intended semantics of OQL queries we have to slightly modify the evaluation plans devised in Subsection 3.2. For this purpose we utilize a notation introduced by [vB91] which we call polarization: A predicate ϕ^- with negative polarization means that after evaluating ϕ a possibly obtained truth value *unknown* is mapped to *false*. A predicate ϕ^+ with positive polarization means that a truth value *unknown* obtained by evaluating ϕ is mapped to *true*. We will assume that $\neg\phi^-$ has the meaning $\neg(\phi^-)$; that is, the polarization has priority over negation. Then the following equivalence holds:

$$\neg\phi^- = (\neg\phi)^+ \tag{21}$$

Using this polarization notation we replace the range and quantifier predicates $p(\dots)$ and $q(\dots)$ in all evaluation plans (3)–(20) by $p^-(\dots)$ and $q^-(\dots)$. That way, *unknown* values obtained by evaluating p or q are always mapped to *false* before further processing the composite predicate. We will demonstrate the correctness of this approach on two example plans for query class 12: First, we consider the "null value robust" variant of plan (7):

$$E_1[e_1] - \big(E_1[e_1] \ltimes_{\neg q^-(e_1,e_2)} \sigma_{p^-(e_2)}(E_2[e_2])\big) \tag{7'}$$

The negatively polarized range predicate $p^-(e_2)$ maps *unknown* predicate values to *false*, thus dropping objects e_2 with *unknown* range predicate from the range subquery. According to the equivalence (21), the semijoin predicate $\neg q^-(e_1, e_2)$ yields *true* for an *unknown* truth value, such that an object pair (e_1, e_2) for which $p(e_2)$ holds but $q(e_1, e_2)$ is unknown qualifies for the semijoin result and is correctly subtracted from the final result.

$$\textbf{select } e \textbf{ from } e \textbf{ in } E \textbf{ where true} \rightarrow E \tag{1}$$

$$\textbf{select } e_i \textbf{ from } e_1 \textbf{ in } E_1, \ldots, e_n \textbf{ in } E_n \textbf{ where false} \rightarrow \emptyset \; (i \in \{1, \ldots, n\}) \tag{2}$$

$$\textbf{for all } e \textbf{ in } E: \textbf{ true} \rightarrow \textbf{true} \tag{3}$$

$$\textbf{for all } e \textbf{ in } E : \textbf{false} \rightarrow \textbf{if } E = \emptyset \textbf{ then true else false} \tag{4}$$

$$\textbf{for all } e \textbf{ in } \emptyset : p \rightarrow \textbf{true} \tag{5}$$

Table 2: Simplification Rules

Next, we consider the anti-semijoin plan:

$$E_1[e_1] \; \bar{\ltimes}_{\neg q^-(e_1,e_2)} \; \sigma_{p^-(e_2)}(E_2[e_2]) \tag{8'}$$

In this plan, corresponding to plan (8), the range predicate $p^-(e_2)$ remains the same as above, again discarding objects e_2 with *unknown* result of $p(e_2)$ from the range. The anti-semijoin predicate $\neg q^-(e_1, e_2)$ again becomes *true* for an *unknown* quantifier predicate q—because of equivalence (21). Consequently, the object e_1 does not qualify for the query result, since the anti-semijoin only returns objects e_1 with no match found.

It is fairly straightforward to verify the validity of this approach to treat *unknown* for the remaining plans.

4 Simplification Rules to Optimize other Variable Bindings

For the optimization of the remaining 13 query classes, we introduce query rewriting rules. The application of the rules is shown in two examples.

4.1 Rewriting Rules

The generic simplification rules are enumerated in Table 2. The first two simplifications treat the cases where the selection predicate of a query block can be simplified to *true* or *false*. The whole query block is then replaced by E or $\emptyset$, respectively. Simplifications 3 and 4 allow the elimination of a universally quantified expression if the quantifier predicate always evaluates to *true* or *false*. An empty range for a universally quantified variable allows application of Simplification 5 (cf., e.g., [JK84]), replacing the universally quantified expression by *true*.

The main case for application of the above simplifications results from replacing global predicates—i.e., those which are not dependent on local variables—by *true* or *false*. For this, we use the *if*-expression introduced in Section 3:

Simplification 6 *If a is some predicate occurring in subexpression E of some query and a has no free variables depending on E, then E can be replaced by*

$$\textbf{if } a \textbf{ then } E[a \leftarrow \textbf{true}]^2 \textbf{ else } E[a \leftarrow \textbf{false}]$$

[2] The expression $E[a \leftarrow b]$ denotes the replacement of all occurences of a in E by b.

Applying this simplification to our query pattern

$$
\begin{array}{rcl}
Q & \equiv & \textbf{select } e_1 \\
 & & \textbf{from } \ e_1 \textbf{ in } E_1 \\
 & & \textbf{where for all } e_2 \textbf{ in select } e_2 \\
 & & \qquad\qquad\qquad\qquad \textbf{from } \ e_2 \textbf{ in } E_2 \\
 & & \qquad\qquad\qquad\qquad \textbf{where } p\text{:} \\
 & & \qquad\qquad\qquad q
\end{array}
$$

results in three specific simplifications:

Simplification 7 *If $q = q()$—query classes 1, 5, 9, and 13 of Table 1—, then the evaluation of q is independent of e_1 and e_2. Hence, we can rewrite the query Q to*

$$\textbf{if } q \textbf{ then } Q[q \leftarrow \textbf{true}] \textbf{ else } Q[q \leftarrow \textbf{false}]$$

Simplification 8 *Similarly, if $p = p()$—query classes 1, 2, 3, and 4 of Table 1—, then we can rewrite the query to*

$$\textbf{if } p \textbf{ then } Q[p \leftarrow \textbf{true}] \textbf{ else } Q[p \leftarrow \textbf{false}]$$

The previous two simplifications can be combined in case neither p nor q depends on e_1:

Simplification 9 *If neither p nor q depend on e_1, that is, e_1 does not occur in either of p or q—query classes 1, 3, 9, and 11 of Table 1—, then their evaluation is independent of e_1 and by dependency-based rewriting [CM93] the query can be rewritten to*

$$
\begin{array}{l}
\textbf{if for all } e_2 \textbf{ in select } e_2 \\
\qquad\qquad\qquad \textbf{from } \ e_2 \textbf{ in } E_2 \\
\qquad\qquad\qquad \textbf{where } p\text{: } q \\
\textbf{then } E_1 \textbf{ else } \emptyset
\end{array}
$$

4.2 Example Applications of the Rewriting Rules

In order to demonstrate the viability of our rule-based query-rewriting approach, we give an example for query class 8—$p(e_1)$ and $q(e_1, e_2)$. Another example for query class 6—$p(e_1)$ and $q(e_1)$—is given in the appendix. Let us consider the query:

```
select f
from   f in Flight
where for all al in select al
                    from  al in Airline
                    where f.to.apctry != "USA":
              f.to in al.lounges
```

This query is a rather strange—and obscure—formulation for finding the flights which either go to a US airport or to an airport at which all airlines have a lounge. Nevertheless, the query system has to be able to handle even such cases; therefore, we want to simplify it into an algebraic representation. The range predicate depends on f, while the quantifier predicate uses both f and al. Consequently, this is an example of query class 8.

After applying Simplification 6, we obtain a query enhanced by an *if*-expression:

```
select f
from   f in Flight
where if f.to.apctry != "USA"
         then for all al in select  al
                              from   al in Airline
                              where true:
                          f.to in al.lounges
         else  for all al in select  al
                              from   al in Airline
                              where false:
                          f.to in al.lounges
```

Now, we can take advantage of Simplification 1 for the *true* branch and Simplifications 2 and 5 for the *false* branch yielding the following query expression:

```
select f
from   f in Flight
where if f.to.apctry != "USA"
         then for all al in Airline:
                          f.to in al.lounges
         else  true
```

Several alternative translations into the algebra are possible. We show the two based on division and anti-semijoin. The resulting query evaluation plans are depicted in Figure 7. As indicated by the textual formulation of the query, we distinguish between flights leading towards a US airport and flights going to a non-US airport. This split is performed by a so-called bypass selection operator $\sigma^{\pm}$ [KMPS94]. It provides an efficient way to split the input into records satisfying the predicate (denoted by σ^{+}) and records that do not satisfy a predicate (denoted by σ^{-}). The "positive" set is joined with the airlines running lounges at the destination airport. The output of the join is then divided by the set of all airlines (cf. Section 3). We obtain two disjoint sets: flights with a US airport destination and flights going to a non-US airport but offering lounges of all airlines. The sets are then combined by a merge union. The special case of an empty set of airlines is separately treated by the *if*-statement.

5 Query Evaluation

For comparison of the different evaluation plans, we executed them using our query engine. Its architecture and some special operators are described in the following.

5.1 Architecture of our Query Engine

The query engine is based on the *Merlin* client/server storage system [Ger96]. The Merlin system consists of a multi-threaded page server and a C++ library that provides the client run time system, including basic components like storage manager and page buffer. The query engine consists of a query compiler and an operator library. The compiler accepts evaluation plans as input and generates a C++ driver program that is linked with the

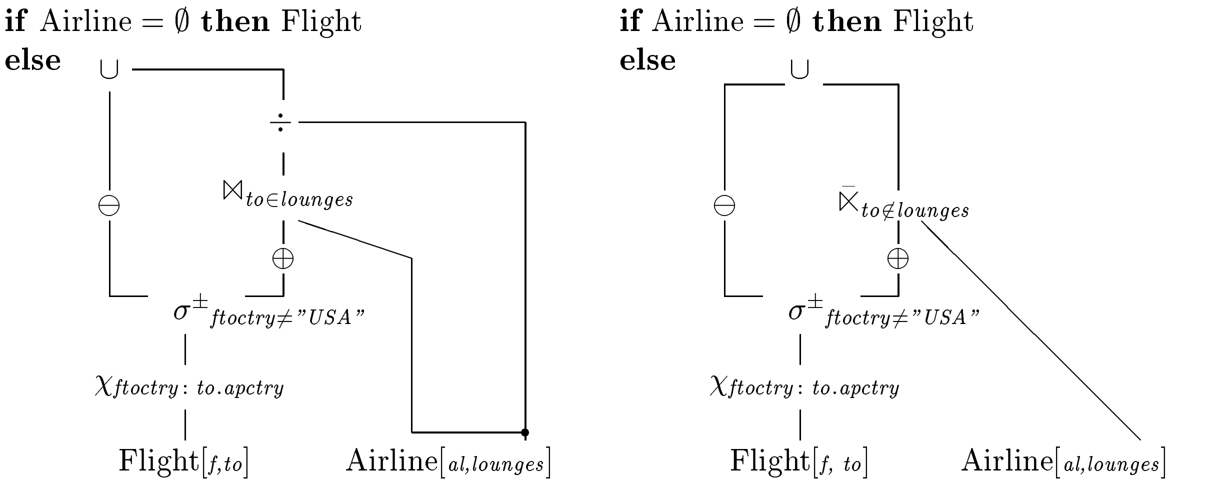


Figure 7: Alternative Evaluation Plans for Query Class 8, based on Division (left) and Anti-Semijoin (right)

operator library. The library provides common relational and object-oriented algebra operators, each encapsulated into a C++ class as an iterator [Gra93]. The hashing variants of matching operators, e.g., hash join, set operations, and duplicate elimination use hybrid hashing [Sha86, Gra93].

5.2 Implementation of the Algebra Operators

In the following, some implementation details of hash division, anti-semijoin, and grouping operators are described.

Hash Division We have implemented the relational division based on hashing as proposed in [GC95]. The algorithm employs two hash tables, a divisor hash table to map divisor objects to a number and a quotient hash table to map each quotient candidate to a bit vector. The bit vector contains one bit position for each divisor object to keep track of the matched divisor objects (quotient candidates with all bits set are returned as result). Since the bit vector size scales proportionally to the number of divisor objects, a large number of divisor objects causes large bit vectors, leading quite early to quotient partitioning.

Anti-Semijoin For an anti-semijoin $E_1 \bar{\ltimes}_p E_2$ all common implementation alternatives like sort merge, hash, and nested-loops come into account. We have implemented block nested-loop and hybrid hash variants. Since a semijoin is not symmetric, there are two variants of each algorithm.

As a nested-loop algorithm, the input stream that will be returned from the operator (E_1) is used as outer loop. The inner loop is scanned once for each cluster of outer blocks. A bit vector containing one bit for each outer record is used to mark if a match has been found for the record. The inner scan may be terminated early if a match has been found for all records. Those records with their bit *not* set are returned. The other variant of the nested-loops join algorithm (inner loop to be returned as result) does not seem to be useful since for all records of the inner input, the operator has to remember which records have already been returned, either by a bit vector or by writing the remaining records to a temporary file for each scan. An index nested-loops implementation might be advantageous, especially if the join predicate contains only the index key attribute, such that the retrieval of the record (object) itself is not even necessary.

The hash variants of the anti-semijoin have been derived from the full hash join which uses the aforementioned hybrid hashing scheme. Again, two variants are possible: one returning records from the build input ($build \mathbin{\bar{\ltimes}} probe$, called semi-build), the other returning probe records ($build \mathbin{\bar{\rtimes}} probe$, called semi-probe). The semi-probe algorithm is straightforward: As soon as a matching build record is found in the hash table, the probe record is dropped, otherwise it qualifies for the result. The semi-build uses a bit vector like in the nested-loop implementation. Both hash variants work without problems if one or more partitioning levels are required.

In comparison to the nested-loop algorithm, hashing suffers from the restriction that it is only generally applicable for equi-joins. This condition may be relaxed to the demand that at least one logical factor in a conjunctive join predicate must be an equality-comparison. This means that hashing is not directly applicable for predicates like $e_2 \in e_1.SetAttribute$, but works for a conjunctive predicate $e_2 \in e_1.SetAttribute \wedge e_1.a = e_2.b$ by performing hashing over the second factor and then verifying the truth of the first [Gra93]. A signature-based solution to employ hashing for predicates with set-valued attributes is proposed in [HM96].

Grouping The implementation of a binary grouping operator $E_1 \, \Gamma_{g;p;\mathbf{aggr}} \, E_2$ as used for our application, i.e., performing an aggregation on the groups, is similar to a semijoin. The hash implementations are based on the corresponding semijoin variants semi-build and semi-probe. The result set consists of *all* objects $e_1 \in E_1$, each augmented by an attribute g for the aggregate value. If no matches are found for a specific e_1, g is set to a default value (e.g., 0 for **count** aggregation). Based on the semijoin implementation, partitioning is applicable. The intermediate aggregate results are merged as discussed in [CM95b]. Since the group members may be dropped immediately after they are processed by the aggregate function, the operator will perform more efficiently than a full join, however more costly than a semijoin, since *all* records of E_1 are returned and no early abort (after first match) is possible. A nested-loops implementation is straightforward.

Element Test and Set Comparison For predicates like $e_2 \in e_1.SetAttribute$ a set element test is needed. In our object model implementation, sets are stored as variable-length unsorted lists. Apart from a naive scan through the list, sorting in combination with binary search is feasible. The lists are sorted on demand as soon as an element test is carried out.

For anti-semijoin plans, the repeated element test $e_2 \in e_1.SetAttribute$ iterating through a fixed set of elements $e_2 \in S$, can be replaced by a subset test $S \subseteq e_1.SetAttribute$. This allows to introduce a cardinality test: If the number of elements in S is larger than the number of elements in $e_1.SetAttribute$, the subset test returns *false* immediately (Of course, the presence of duplicates in the set S has to be precluded). Otherwise, the subset test must really be carried out, i.e., the sets are sorted (if necessary) and compared in a single linear scan. The "smart anti-semijoin" variant of query 1 employs this subset test. Details about set comparison techniques in join predicates, especially signature-based set comparison, are discussed in [HM96].

6 Benchmarking

In this section, we present performance experiments comparing the alternative evaluation plans that we have discussed in Section 3.

6.1 Benchmark Platform Parameters

The experiments were performed with the query engine as described in the previous section. The query client and page server were run on two separate two-processor SUN SparcStation 20/502MP under Solaris 2.5. The database was held on the server's disk of type Seagate ST31230WC with an average access time of 10.4/11.4ms (read/write).

For the first set of experiments, we have generated two different databases. A small one for initial assessment and a larger one. Table 8 shows the size and cardinality of both databases. All attribute values except for *Airline.lounges* were pseudo-randomly generated and distributed uniformly. The cardinality of the set-valued attribute *Airline.lounges* was also distributed uniformly in a certain range. For an easy modification of the selectivity of $\sigma_{apctry='USA'}$ and $\sigma_{apctry!='Libya'}$, *apctry* is an integer attribute and the predicate is in fact a comparison with an integer constant, e.g., $\sigma_{apctry\leq 20}$. Note that this transition from an equality predicate to a range predicate does not change the examined evaluation plans. The individual set elements of *lounges* have been filtered such that the number of US-airports is higher than average, in order to get a non-empty result for query 1.

In our query engine, memory allocation is performed on a per-operator basis. For each hash table and for each extension scan operator a memory area of 1.5MB was used, such that more complex plans employing several hash tables get more resources than simpler ones. If we had assigned a unique global amount of memory to all plans, the performance gap between cheap ones (especially anti-semijoin plans) and more expensive plans like set difference would have become even larger. Since we wanted to assess the "pure" evaluation plans, we have not created any indexes for the experiments.

6.2 Benchmark Results

Small DB (Figure 9 – Figure 11) Let us start with query 1. Figure 9 shows run times for all evaluation plans presented in Section 3. In addition, a "canonical" plan with a nested-loop implementation is evaluated. On the x-axis the selectivity of the range predicate, i.e., $\sigma_{apctry='USA'}$, has been varied. This influences the number of *Airport* records in the divisor respectively in the join input. (Note the logarithmic scaling in this and some of the following plots.) The query result cardinality ranges from 576 objects at the leftmost point (selectivity=0.002) to 0 records on the right. The run times of the hash-based evaluation plans are all in the same order of magnitude. Division shows a

	small	large
total size [MB]	1.7	17.2
#Airports (E_2)	1000	10,000
#Airlines (E_1)	1000	10,000
#Flights	10,000	10,000
avg. #Lounges per Airline	35	40

Figure 8: Database Configurations

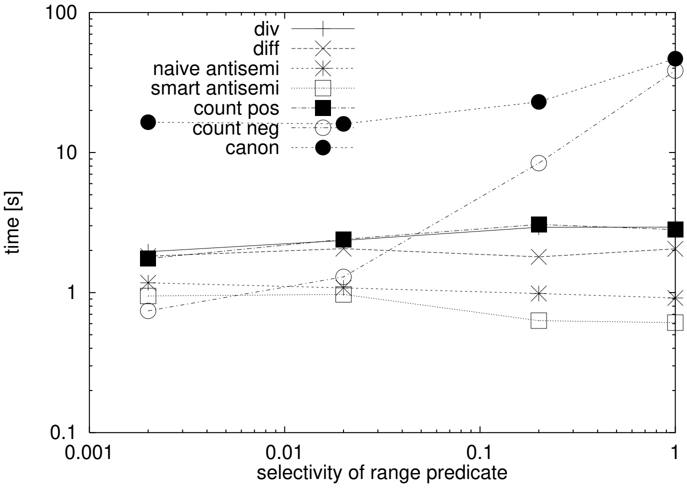


Figure 9: Query 1, Small Database

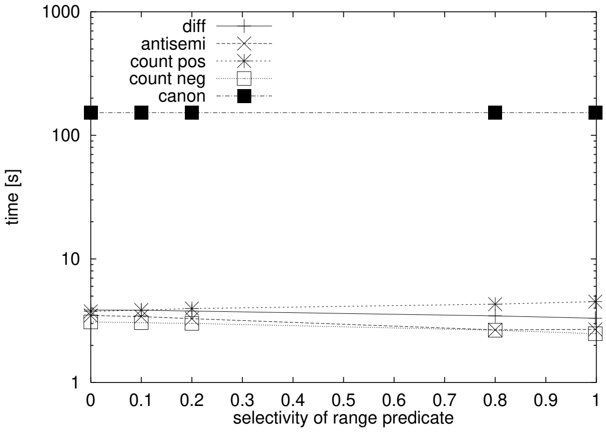


Figure 10: Query 2, Small Database

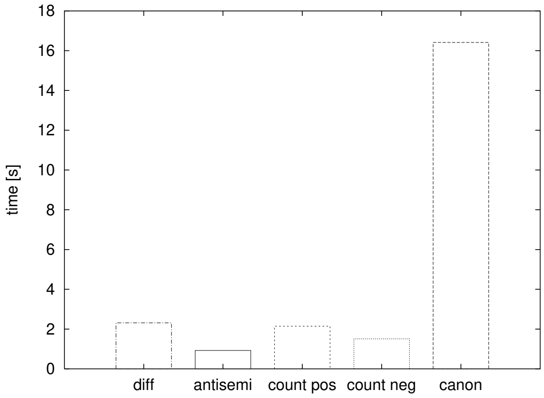


Figure 11: Query 3, Small Database

slight run time increase with growing number of airports qualifying for the range. This is due to the increasing number of divisor records, resulting in more entries in the divisor hash table and larger bitmaps in the quotient hash table. Counting matches—denoted "count pos" in the figures—shows the same tendency. Set difference shows nearly constant run time, while anti-semijoin run-time decreases with increasing number of airports in the range. The reason is that a larger number of airports causes an earlier disqualification of airlines, especially in the "smart" implementation where the cardinalities of both sets are compared first. For counting mismatches—denoted "count neg"—no hash implementation is possible. Consequently its nested-loops implementation is only competitive for a very small number of airports in the range and behaves similar to the "canonical" variant for a larger airport count.

For query 2 (Figure 10), the result ranged from one record at the left-most point to 959 records at the right-most end. Anti-semijoin and count negative are the fastest plans. Both use only a single hash table, as opposed to set difference and count positive, using two and three hash tables, respectively. Since in this query, the quantifier predicate is not

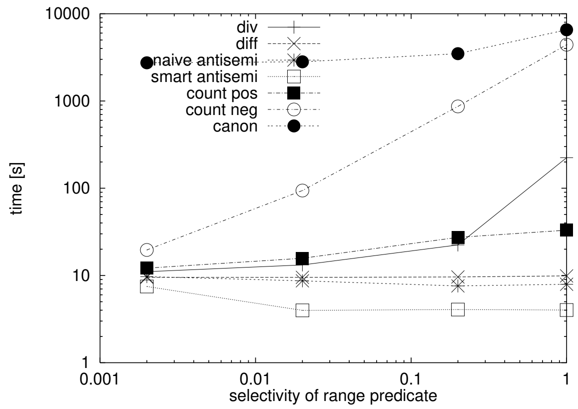


Figure 12: Query 1, Large Database

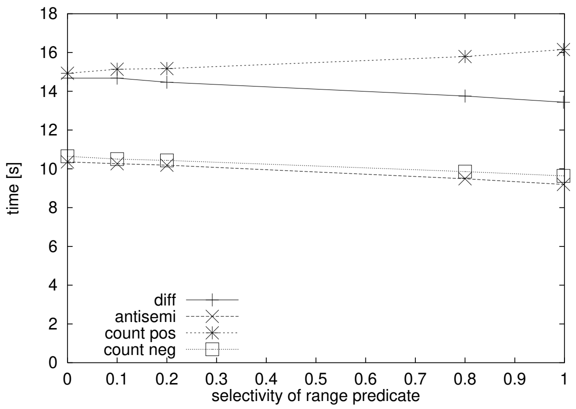


Figure 13: Query 2, Large Database

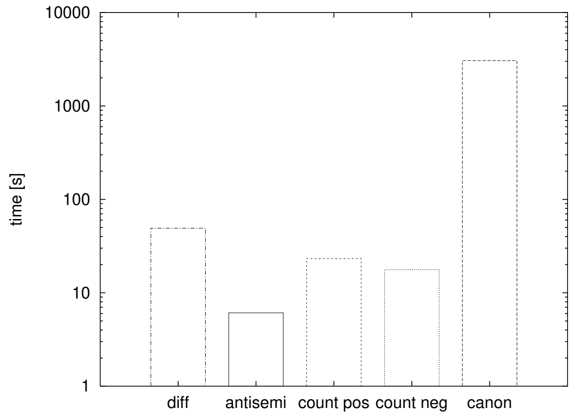


Figure 14: Query 3, Large Database

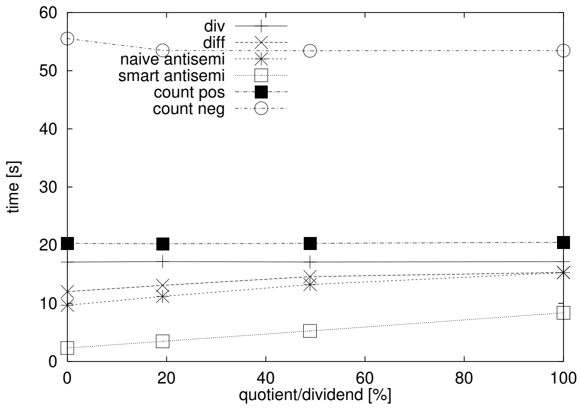


Figure 15: Query 1: Changing the Result Cardinality

a set comparison, anti-semijoin cannot exploit the comparison of set cardinalities. The growing execution time for count positive is caused by the increasing number of matches in the final join, whereas the final difference in the set difference plan becomes cheaper.

The result of query 3 cannot be changed by simply modifying a constant in a selection predicate, since both range and quantifier predicate are join predicates, i.e., they both depend on e_1 and e_2. For this reason, we present only single bars for each plan (Figure 11). Again, anti-semijoin is the best plan, followed by count negative. Set difference and count positive show similar run times around 2 seconds, while the "canonical" nested-loop implementation again is an order of magnitude more expensive than anti-semijoin.

Large DB (Figure 12 – Figure 14) After this initial assessment, we have scaled our database to an amount of 10.000 objects of each type. The results (Figures 12–14) confirm the initial assessment. For all three queries, the anti-semijoin plan remains the winner. Figure 12 shows the run-times for query 1. Division causes moderate costs by the quite large bit vectors stored in the quotient table (up to 1250 bytes for 10000 divisor records), leading even to quotient partitioning—leading to the drastic increase in run time at the

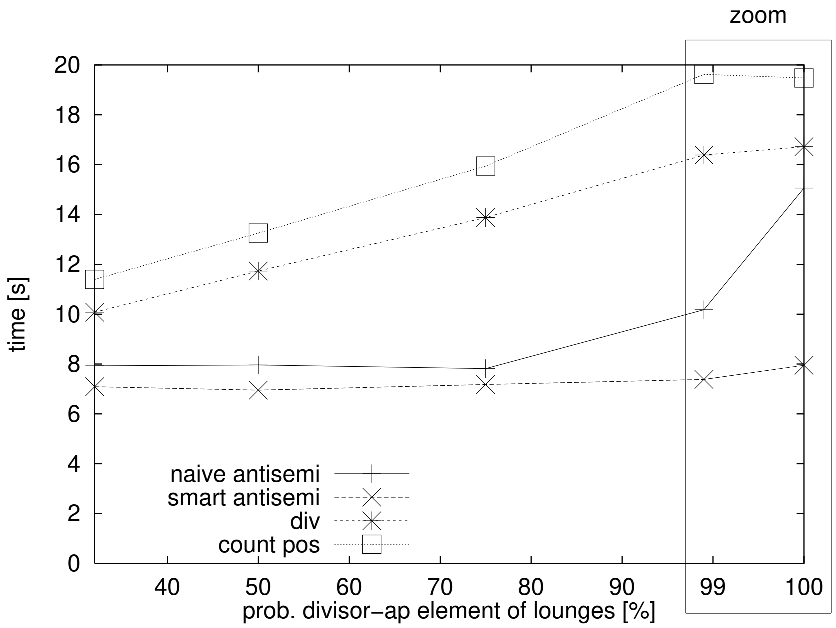


Figure 16: Query 1: Changing Probability for divisor-ap ∈ lounges

right-hand side of the figure. Set difference shows nearly constant run time, while anti-semijoin draws profit from the “early abort”. Especially “smart antisemi” is very cheap because of the cardinality test. Count positive contains three hash operations and thus performs only moderately, while the run times for the nested-loop plans “count negative” and “canon” are several orders of magnitudes higher.

In the plot for query 2 (Figure 13), the canonical variant is omitted (run time of approx. 2000 sec). The remaining four plans behave as before. Figure 14 shows a similar scenario for query 3.

Looking at the Result Cardinality In the following experiment, we wanted to investigate the influence of the result cardinality upon query run time. For this purpose, query 1 was run on variants of the small database with modified *lounges* attribute. Given a range of 318 airport objects, three databases were built. The first one with a *lounges* cardinality uniformly distributed between 317 and 318 with airport references chosen from the range. On this database, about 50 percent of the airline objects qualified for the result. In the same way, two further databases were built, one with constant *lounges* cardinality of 318 (i.e., all airlines qualified for the result), and another one with *lounges* cardinality between 315 and 318, selecting roughly 25 percent of the airlines for the result. The 0 percent mark was obtained by raising the range to 319, such that no airline qualified. Figure 15 shows the run time for the different plans of query 1. While division and counting are hardly influenced by the result cardinality, both anti-semijoin variants draw profit on the fact. This gain is caused by cardinality comparison and a cheap element/subset test by means of sorting. The set difference plan requires an additional hash operation and is thus more expensive than naive anti-semijoin.

To avoid the early abort of the smart anti-semijoin due to mismatching cardinalities, the (rather unrealistic) scenario was built that all *lounges* sets and the range have the same cardinality of 318 elements. Instead of the cardinality, the probability for each of the 318 airports in the range (i.e., USA airports) to be element of a *lounges* set has been varied. Figure 16 depicts the run times of the different plans for query 1. The anti-semijoin plans show nearly constant run times (although the run time for the naive variant increases at a probability close to 100 percent due to an increasing number of element tests), while the run time of division and counting plans increases with the number of hits for the quantifier predicate. Since the query result set is empty except for probabilities close to 99 percent, we have zoomed the area around 99 percent in the plot.

6.3 Comparison to Existing Commercial Systems

To facilitate a comparison with commercially available database management systems, we have run our three example queries both on an object base management system (OBMS) and on a relational DBMS. In both cases, the small database containing 1000 airports, 1000 airlines, and 10000 flights has been loaded into the system. The range predicate was turned into an exact match predicate and resulted in a selectivity of 0.002 for queries 1 and 2. The OBMS supported OQL and set-valued attributes, such that the queries could be directly formulated as shown in Figure 3. The page buffer was left at the default size of 4MB for both the client and the server. Query 1 took 15 seconds, a factor of about ten times the run time of our anti-semijoin plan. The run time for query 2 shows a dramatical difference, 2150 seconds against 3 seconds. This may be due to the "dot" operator in the quantifier predicate *f.to.apctry!='Libya'* in combination with an inefficient plan. Query 3 took 180 seconds on the commercial OBMS system, more than a factor 100 against our anti-semijoin plan.

To execute the example queries on a relational DBMS, some more effort was necessary. To represent the relationship *lounges*, the set-valued attribute *lounges* was replaced by a separate table *Lounges* containing the foreign keys *airlineName* and *airportCode*. The queries we derived in Figure 3 were rephrased in SQL for the different schema; in particular, we had to convert the universal quantification which is not part of SQL-92. Here are the SQL variants of query 1 using "not exists" and counting to formulate the universal quantification.

```
select al.name
from Airline al
where not exists
        (select *
        from Airports ap
        where ap.apctry = 'USA' and not exists
                (select *
                from Lounges l
                where l.airportCode = ap.code and
                        l.airlineName = al.name))
```

```
select l.airlineName
from Airports ap, Lounges l
where ap.apctry = 'USA'
        and l.airportCode = ap.code
group by l.airlineName
    having count (l.airportCode) =
            (select count (*)
            from Airports ap2
            where ap2.apctry = 'USA')
```

We have run "not exists" and counting variants for all three queries. Apart from the counting variant of query 1 all queries show run times that are orders of magnitudes larger than our anti-semijoin plans (cf. Table 3). Note the dramatic difference between the **not exists** and **count**ing variants of query 1. This shows that the RDBMS we

run time [s]	**not exists**-formulation	**count**-formulation
query 1	1902	2
query 2	131	126
query 3	313	45

Table 3: Query Run Times for a RDBMS (Small Database)

Parameter	Time [ms]	Description
SeqIO	7	average sequential I/O 1 page
RndIO	15	average random I/O 1 page
RdRec	0.06	CPU-cost per record when reading a permanent file
RdTmp	0.04	CPU-cost per record when reading a temporary file
Comp	0.003	comparison of two keys or predicate evaluation
Exch	0.001	exchange of two set elements
Aggr	0.003	call of aggregate function for a tuple
Hash	0.005	calculating a hash value from a tuple
Move	0.04	memory to memory copy of one record
Bit	0.001	setting a bit in a bit map
Map	0.0001	initializing or scanning a bit in a bitmap
Unnest	0.002	unnest operation for one set element

Table 4: Cost Model Parameters

employed was not able to optimize the hidden universal quantification. The "not exists" plan used a straightforward nested-loop evaluation. The user was obliged to explicitly phrase the counting variant (resembling our evaluation plan (9)). The counting variant of query 1 exploited the fact that the range was closed, i.e., the count value for the range is a constant. The queries probably suffered from the large overhead of the *Lounges* table. As opposed to the compact representation of the lounges in a set-valued attribute, in the relational schema a row exists for each set element.

7 Cost Estimation

As a foundation of a cost-based query optimizer we sketch a simple cost model for the presented evaluation plans. It is based on the cost model used in [GC95]. To simplify our approach, we assume that all temporary data structures (hash tables, inner loop of a nested-loop join) will fit into memory. This was mainly the case in the experiments above.

7.1 Cost of Operations

As building blocks for the evaluation plans to be discussed we first estimate the cost of some basic operations used in several queries. Constant parameters are accumulated in Table 4. The variables used in the following formulas—depending on the database—are summarized in Table 5.

Variable	Description
$\|E\|$	cardinality of an extent E
p_E	number of pages occupied by extent E
σ_x	selectivity for predicate x
$\|E_1.set\|$	average cardinality of set-valued attribute *set*
$\tau_{(mis)match}$	average fraction of records of an inner loop that have to be tested until a (mis)match has been found for an outer record
θ	fraction of set valued attributes that actually have to be sorted
ε	fraction of comparisons actually performed in a subset test
δ	probability of a match dividend $\in$ divisor
q	|quotient| / |dividend|
hc	average number of hash collisions per bucket

Table 5: Cost Model Variables

A single sequential scan through an extent for type E causes sequential I/O and needs some CPU time:

$$\text{Scan}(E) = p_E \cdot \text{SeqIO} + |E| \cdot \text{RdRec}$$

Repeated scans through the same extent, then kept in memory, are approximated by the CPU part of the above formula.

The set-valued attributes were stored as unsorted arrays in the experiments. Sorting a set of n elements using quick sort costs

$$\text{SortSet}(n) = 2n \cdot \ln n \cdot \text{Comp} + n/3 \cdot \ln n \cdot \text{Exch}$$

If the sets are kept sorted by the system these costs may be omitted. An element test on a set—represented as a sorted array—of cardinality n is performed by binary search with $\log_2 n$ comparisons:

$$\text{ElemTest}(n) = \text{Comp} \cdot \log_2 n \qquad \text{SubSetTest}(n_1, n_2) = \text{Comp} \cdot \varepsilon \cdot (n_1 + n_2)$$

A subset test between two sorted sets with cardinalities n_1 and n_2 may be implemented as a single scan through the sets, requiring $\varepsilon \cdot (n_1 + n_2)$ comparisons. The factor ε expresses the average savings due to a cardinality pretest and early aborts as soon as satisfaction of the test is impossible.

The cost for a nested loop join is approximated by the following formula.

$$\text{NLJoin}(\mathit{OuterRecs}, \mathit{InnerRecs}, \mathit{pred}) = \mathit{InnerRecs} \cdot \mathit{OuterRecs} \cdot (\mathit{pred} + \text{RdRec}) \quad (22)$$

Since we assume that the inner input is cached in memory, no distinction is made between naive and block nested loop join. For each pair of inner and outer records, one record is read from cache and the predicate is evaluated. In the case of a semijoin, the inner loop is only scanned to a certain fraction. For an anti-semijoin, this fraction is adapted to the probability of finding mismatches.

The times for building and probing a hash table are estimated as follows:

$$\text{HashBuild}(\mathit{BuildRecs}) = \mathit{BuildRecs} \cdot (\text{Hash} + hc \cdot \text{Comp} + \text{Move}) \tag{23}$$
$$\text{HashProbe}(\mathit{ProbeRecs}) = \mathit{ProbeRecs} \cdot (\text{Hash} + hc \cdot \text{Comp}) \tag{24}$$

The component $hc \cdot \text{Comp}$ is caused by the search for duplicates in the case of hash collisions and may be omitted if no duplicates are present.

The cost for a hash division $\mathit{Dividend} \div \mathit{Divisor}$ consists of several components: First, the divisor hash table is built, assuming there are no duplicates present. Then, all dividend records are probed against the divisor table; on success a record is probed against the quotient table and inserted into the quotient table. The final term represents the cost of initializing and scanning the bitmaps.

$$\begin{aligned}\text{HashDiv}(\mathit{DendRecs}, \mathit{DsorRecs}) = {} & \mathit{DsorRecs} \cdot \text{Hash} + \mathit{DendRecs} \cdot (\text{Hash} + hc \cdot \text{Comp}) \\ & + \mathit{DendRecs} \cdot \delta \cdot (\text{Hash} + hc \cdot \text{Comp} + \text{Bit}) + 2 \cdot q \cdot \mathit{DendRecs} \cdot \text{Map}\end{aligned} \tag{25}$$

The hash implementation of the grouping operator $E_1 \, \Gamma \, E_2$ processes the build input and aggregates the probe input:

$$\begin{aligned}\text{GroupHash}(E_1 recs, E_2 recs, \mathit{pred}) = {} & \text{HashBuild}(E_1 recs) \\ & + E_2 recs \cdot (\text{Hash} + hc \cdot \text{Comp} + \mathit{pred} + \text{Aggr})\end{aligned} \tag{26}$$

7.2 Cost of Evaluation Plans of Query Class 12

In the following, we will first state cost formulas for the evaluation plans presented in Section 3 for query class 12 and then compare the estimations with the run time experienced in the previous section. All of the following formulas share the cost of scanning both input extents and evaluating the range predicate. Writing the final query output has been omitted for all plans (In the experiments the result size was very small).

Division The division plan (cf. plan (5) on page 8) unnests the set-valued attribute, evaluates the range predicate, and employs the relational division operator:

$$\begin{aligned}& \text{Scan}(E_1) + \text{Scan}(E_2) + \text{Unnest} \cdot |E_1| \cdot |E_1.set| + |E_2| \cdot \text{Comp} \\ & \qquad + \text{HashDiv}(|E_1| \cdot |E_1.set|, |E_2| \cdot \sigma_p)\end{aligned} \tag{27}$$

Set Difference The set difference plan (cf. plan (7)) consists of a nested loop semijoin followed by a hash difference, totally estimated by

$$\begin{aligned}& \text{Scan}(E_1) + \text{Scan}(E_2) + |E_1| \cdot \text{Sort}(|E_1.set|) + |E_2| \cdot \text{Comp} \\ & \qquad + \text{NLJoin}(|E_1|, |E_2| \cdot \sigma_p \cdot \tau_{match}, \text{ElemTest}(|E_1.set|)) \\ & \qquad\qquad + \text{HashBuild}(|E_2| \cdot \sigma_p) + \text{HashProbe}(|E_1|)\end{aligned} \tag{28}$$

Although sorting the sets $E_1.set$ is accounted globally for all *Airline* objects, it is actually performed on demand for each element test. The semijoin semantic is expressed by the factor τ_{match}, indicating that on average a scan of the inner loop may be terminated after a fraction of τ_{match}.

Anti-Semijoin The factor $\tau_{mismatch}$ represents the average fraction of the inner extent that needs to be scanned until a mismatch is found.

$$\begin{aligned}\mathrm{Scan}(E_1) + \mathrm{Scan}(E_2) + |E_1| \cdot \mathrm{Sort}(|E_1.set|) + |E_2| \cdot \mathrm{Comp}\\ + \mathrm{NLJoin}(|E_1|, |E_2| \cdot \sigma_p \cdot \tau_{mismatch}, \mathrm{ElemTest}(|E_1.set|))\end{aligned} \quad (29)$$

"Smart" Anti-Semijoin The "smart" implementation converts the inner loop of the join into a set, such that the element test for each object e_2 of E_2 is replaced by a single subset test for each object of E_1. The factor θ indicates that sets are only sorted if a comparison of set cardinalities does not preclude a match in advance.

$$\begin{aligned}\mathrm{Scan}(E_1) + \mathrm{Scan}(E_2) + (|E_1| \cdot \theta + 1) \cdot \mathrm{Sort}(|E_1.set|) + |E_2| \cdot \mathrm{Comp}\\ + |E_1| \cdot \mathrm{SubSetTest}(|E_1.set|)\end{aligned} \quad (30)$$

Counting Matches The first counting plan (cf. plan (9)) employs the hash-based variant of the Γ-operator, performing like a hash join with an additional aggregate function call. Since the right input of the cross product contains merely one record, the cross operation comes at nearly no cost. The final term reflects the top-level selection.

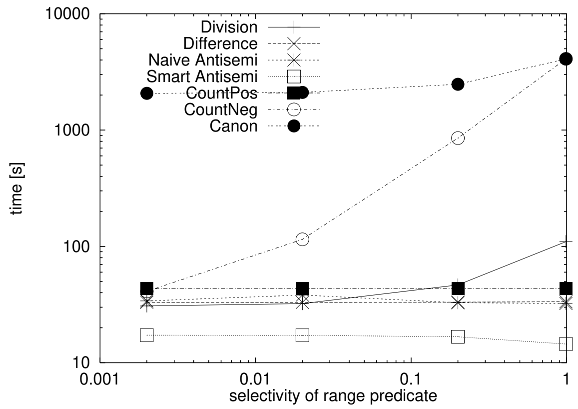
$$\begin{aligned}\mathrm{Scan}(E_1) + \mathrm{Scan}(E_2) + |E_1| \cdot \mathrm{Sort}(|E_1.set|) + |E_2| \cdot \mathrm{Comp}\\ + \mathrm{HashBuild}(|E_1|) + \mathrm{HashProbe}(|E_2| \cdot \sigma_p)\\ + |E_2| \cdot \sigma_p \cdot (\mathrm{ElemTest}(|E_1.set|) + \sigma_q \cdot \mathrm{Aggr}) + |E_1| \cdot \mathrm{Comp}\end{aligned} \quad (31)$$

Counting Mismatches The variant counting mismatches is forced to use a nested loop implementation, using E_1 as outer loop.

$$\begin{aligned}\mathrm{Scan}(E_1) + \mathrm{Scan}(E_2) + |E_1| \cdot \mathrm{Sort}(|E_1.set|) + |E_2| \cdot \mathrm{Comp}\\ + \mathrm{NLJoin}(|E_1|, |E_2| \cdot \sigma_p, \mathrm{ElemTest}(|E_1.set|) + \sigma_q \cdot \mathrm{Aggr}) + |E_1| \cdot \mathrm{Comp}\end{aligned} \quad (32)$$

The above cost formulas have been instantiated with the constants from Table 4 and the parameters of the larger database used in some of the experiments of the previous section. Figure 17 shows the cost estimations for query 1, corresponding to the experiments plotted in Figure 12. Both diagrams show similar tendencies and absolute run times for the evaluation plans. The cost estimations are slightly higher than the actually observed run times, probably because some of the parameters overestimate the running time of the respective operation. For example, the operation *Comp* heavily depends on the kind of predicate, *Move* depends on the number of bytes moved, and *Hash* is influenced by the type of key (e.g., *integer* or *string*) and thus the hash functions employed. For these parameters, we have chosen worst-case constants.

For further assessment, we have adapted the cost functions to the evaluation plans for query 2. The results are depicted in Figure 18, corresponding to the benchmark results shown in Figure 13. Again, the tendency is the same as observed in the real experiments, while the absolute run time estimate is slightly higher than the measured running time.



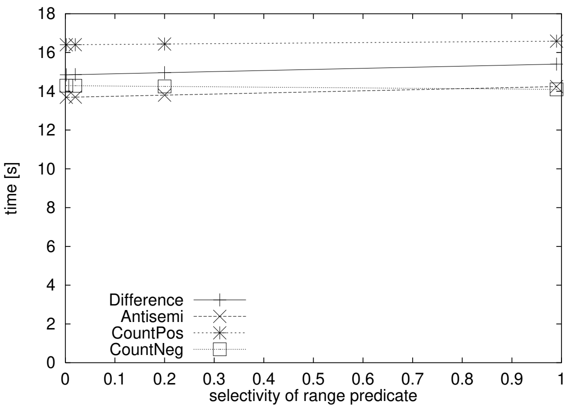
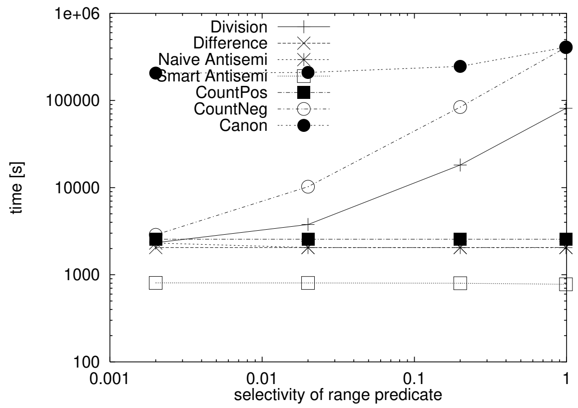


Figure 17: Cost Model applied on Query 1 (cf. Figure 12)

Figure 18: Cost Model applied on Query 2 (cf. Figure 13)



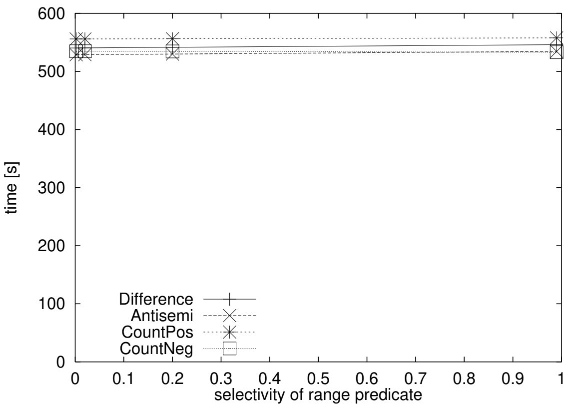


Figure 19: Cost Model applied for Query 1, Database scaled by Factor 10

Figure 20: Cost Model applied for Query 2, Database scaled by Factor 10

In Figures 19 and 20 we have scaled the extent cardinalities and the average set cardinality of *lounges* by a factor of ten. While the ratio remains nearly the same for query 1 and thus scales well, the run times for query 2 are dominated by the fix cost of extent scans. Since the set-valued attribute *lounges* is stored within the corresponding object, the Airline extent grows by a factor of 100 in total, increasing the cost for a scan through this extent by this factor. Query 2 does not access the set-valued attribute, such that the anti-semijoin is unable to exploit its advantage in processing these sets. Additionally, the cost model does take the different resource consumptions of the plans into account. As explained before, the anti-semijoin plan is the only plan to employ only one hash operator, while, e.g., the set difference plan uses three. This would result either in minimal resource consumption of the anti-semijoin plan or in drastically different run times. In spite of these limitations, anti-semijoin is still the fastest plan in Figure 20.

8 Conclusion

We investigated the processing of queries with universal quantification from the source level over algebraic rewriting down to query plan generation and evaluation. Due to our main focus on object-oriented and object-relational data models, we were able to derive more valid and much more efficient algebraic rewritings than known from the relational context. The correct handling of null values was incorporated into the equivalences.

The quality of the different evaluation plans was evaluated by a performance analysis on some sample database configurations. The quantitative analysis has revealed that—especially if set-valued attributes can be employed—the new anti-semijoin-based plans are superior to all other alternatives, even if we employ the most sophisticated division algorithms. This is due to the fact that the anti-semijoin is able to draw profit from object-oriented features like object identity and the compact representation of multi-valued relationships.

Acknowledgements

This work was supported by the German Research Council DFG under contract Ke 401/6-2. We thank Günther Buchner for implementing the division operator.

References

[BMG93] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the Open OODB Query Optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 287–295, Washington, DC, USA, May 1993.

[Bry89a] F. Bry. Logical rewritings for improving the evaluation of quantified queries. In *Int. Conf. on Mathematical Fundamentals of Database Systems*, volume 364 of *Lecture Notes in Computer Science (LNCS)*, pages 100–116, June 1989.

[Bry89b] F. Bry. Towards an efficient evaluation of general queries: Quantifiers and disjunction processing revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 193–204, Portland, OR, USA, May 1989.

[Car86] J.V. Carlis. A relational algebra operator, or divide is not enough to conquer. In *Proc. IEEE Conf. on Data Engineering*, pages 254–261, New York, USA, 1986.

[Cat96] R. G. G. Cattell. *The Object Database Standard – ODMG-93.* Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1996.

[CM93] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, 1993.

[CM95a] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[CM95b] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proc. Int. Workshop on Database Programming Languages*, 1995.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Database Systems*, pages 65–98. Prentice Hall, Englewood Cliffs, NJ, USA, 1972.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 197–208, Brighton, England, 1987.

[Day83] U. Dayal. Processing queries with quantifiers: A horticultural approach. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 125–136, Atlanta, USA, March 83.

[GC95] G. Graefe and R. L. Cole. Fast algorithms for universal quantification in large databases. *ACM Trans. on Database Systems*, 20(2):187–236, June 1995.

[Ger96] C. A. Gerlhof. *Optimierung von Speicherzugriffskosten in Objektbanken: Clustering und Prefetching*, volume 18 of *Dissertationen zu Datenbanken und Informationssystemen.* infix-Verlag, Ringstr. 32, 53757 Sankt Augustin, 1996. (in German).

[GL87] R. A. Ganski and H. K. T. Long. Optimization of nested SQL queries revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 22–33, San Francisco, USA, May 1987.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[HM96] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison predicates. Technical Report 13/1996, University of Mannheim, October 1996.

[HP95] P.-Y. Hsu and D. S. Parker. Improving SQL with generalized quantifiers. In *Proc. IEEE Conf. on Data Engineering*, pages 298–305, Taipeh, Taiwan, 1995.

[JK83] M. Jarke and J. Koch. Range nesting: A fast method to evaluate quantified queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 196–206, San José, USA, 1983.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KMPS94] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 336–347, Minneapolis, MI, USA, May 1994.

[Mai83] D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, MD, USA, 1983.

[Mel94] J. Melton. Framework for SQL. ANSI X3H2-94-079/SOU-003 (ISO Working Draft), March 1994.

[MS93] J. Melton and A. R. Simon. *Understanding the new SQL: a complete guide.* Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1993.

[Mur88] M. Muralikrishna. Optimization of multiple-disjunct queries in a relational database system. Technical Report #750, University of Wisconsin–Madison, February 1988.

[Nak90] R. Nakano. Translation with optimization from Relational Calculus to Relational Algebra having aggregate functions. *ACM Trans. on Database Systems*, 15(4):518–557, December 1990.

[RBG96] S. G. Rao, A. Badia, and D. Van Gucht. Providing better support for a class of decision support queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 217–227, Montreal, Canada, June 1996.

[RGL90] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees and freely-reorderable outerjoins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 291–299. acmpress, May 1990.

[RRS91] A. Rosenthal, C. Rich, and M. Scholl. Reducing duplicate work in relational join(s): a modular approach using nested relations. Technical report, ETH Zürich, 1991.

[SABdB94] H. J. Steenhagen, P. M. G. Apers, H. M. Blanken, and R. A. de By. From nested-loop to join queries in OODB. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 618–629, Santiago, Chile, September 1994.

[Sha86] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(9):239–264, September 1986.

[SPMK95] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Bypassing joins in disjunctive queries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 228–238, Zürich, Switzerland, September 1995.

[Ste95] H. J. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, University of Twente, October 1995.

[Sto96] Stonebraker. *Object-Relational DBMSs: The Next Great Wave*. Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1996.

[vB91] G. von Bültzingsloewen. *SQL-Anfragen - Optimierung für parallele Bearbeitung*. FZI-Berichte Informatik. Springer-Verlag, New York, Berlin, etc., 1991. (in German).

[WMSB90] K. Y. Whang, A. Malhotra, G. Sockut, and L. Burns. Supporting universal quantification in a two-dimensional database query language. In *Proc. IEEE Conf. on Data Engineering*, pages 68–75, L.A., CA, February 1990.

Appendix: Another Example Application of the Simplification Rules

As an example for query class 6, assume the OQL query below is given.

```
select f
from   f in Flight
where for all al in select al
                    from  al in Airline
                    where f.from.apctry != "USA":
               f.to.apctry = "USA"
```

If *Airline* contains no objects, all *Flight*s qualify for the query result. If *Airline* contains at least one object, a particular *Flight* qualifies if it originates in the USA or if its destination is in the USA. Since both the range predicate *f.to.apctry != "USA"* and the quantifier predicate *f.from.apctry = "USA"* are not correlated with the subquery, we can apply Simplification 6 twice to pull both predicates out of the subquery:

```
select f
from   f in Flight
where if f.from.apctry != "USA"
         then if f.to.apctry = "USA"
                 then for all al in select al
                                    from  al in Airline
                                    where true:
                               true
                 else  for all al in select al
                                    from  al in Airline
                                    where true:
                               false
         else  if f.to.apctry = "USA"
                 then for all al in select al
                                    from  al in Airline
                                    where false:
                               true
                 else  for all al in select al
                                    from  al in Airline
                                    where false:
                               false
```

Applying Simplifications 1 and 2 to eliminate the constant range predicate yields

```
select f
from   f in Flight
where if f.from.apctry != "USA"
         then if f.to.apctry = "USA"
                 then for all al in Airline: true
                 else  for all al in Airline: false
         else  if f.to.apctry = "USA"
```

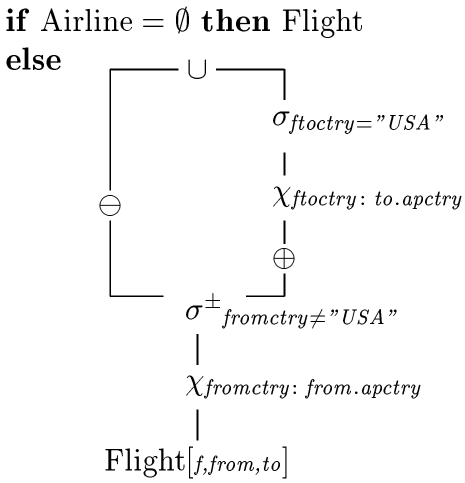


Figure 21: Evaluation Plan for Query Class 6

then for all al **in** ∅: **true**
else for all al **in** ∅: **false**

Simplifications 3, 4 and 5 remove the universal quantifications:

select f
from f **in** Flight
where if f.from.apctry != "USA"
  then if f.to.apctry = "USA"
    then true
    else if Airline = ∅
      then true
      else false
  else true

The test if the airline extension is empty is finally pushed up as outermost operation (Simplifications 6 and 1). For the translation of the *if*-operator into the algebra we again employ the bypass select operator $\sigma^{\pm}$ [KMPS94]. Using a bypass selection, the above query yields the bypass plan depicted in Figure 21. Thanks to the bypass operator the predicate *f.from.apctry!="USA"* is evaluated exactly once for each flight. Furthermore, since the two output streams are disjoint, the union operator $\cup$ used to recombine the two streams does not require duplicate elimination. While the first χ operator that retrieves the *from* airports is evaluated for all flights, the second χ operator accessing the destination airports is only evaluated for the positive output stream of the bypass selection operator, such that it is bypassed by the negative stream.