# COST-BASED QUERY OPTIMIZATION FOR METADATA REPOSITORIES

Martin Staudt[1], René Soiron[2], Christoph Quix[3], Matthias Jarke[3]

[1] Swiss Life, Information Systems Research, P.O. Box, 8022 Zurich, Switzerland

[2] Ericsson Eurolab, Systemhouse GPRS, Ericsson Allee 1, 52134 Herzogenrath, Germany

[3] RWTH Aachen, Informatik V, Ahornstr. 55, 52056 Aachen, Germany

**Abstract**

Query optimization strategies for repository systems must take into account the rich and often unpredictable structure of metadata, as well as supporting complex analysis of relationships between those structures. This paper describes rationale, design, and system integration of a cost-based query optimizer offered in ConceptBase, a metadata manager that supports these capabilities by a deductive object-oriented data model. In contrast to most implemented DBMS, the optimizer is not based on the concept of join selectivity, but on detailed distribution information about object and literal fan-out, with heavy exploitation of materialized views and end-biased histograms. Experiences from two real-world applications in software configuration management and cooperative design demonstrate the practical advantages but also point to some lessons for further improvement.

## 1 INTRODUCTION

Novel models of database-centered information flows in organizations, such as the reconciliation of heterogeneous data in data warehouses [35, 9], the materialization of complex hierarchical, object-oriented or multi-dimensional views on such databases in client systems [8, 31], or the management of design repositories for information systems [25] have renewed the interest in metadata management systems that combine the structural richness of object-oriented data models with analytic capabilities.

In some systems, these analytic capabilities are provided by dedicated methods, e.g. the Microsoft repository [2]; generic analysis methods are specialized to particular cases via inheritance mechanisms but there is usually no coherent inferencing mechanism for the combination of such methods. Such a coherent inferencing mechanism can be provided by the integration of object-oriented and deductive database technologies in so-called *deductive object bases (DOBs)*. A number of such data models have been proposed, including Telos/ConceptBase [14, 24], F-Logic [21, 6] and CORAL++ [30]. Moreover, some of the deductive database systems developed in the late 1980's, including LDL [36], Declare [19], Glue-Nail [5], and Aditi [16] have studied at least some partial solutions which are potentially relevant for this context.

The analysis of models and model interactions in DOBs relies heavily on their querying capabilities. Most of the above systems have developed query languages whose semantics can somehow be related to extensions of Datalog [1, 20].

As meta databases grow beyond toy applications, the question whether existing approaches to query optimization are sufficient for DOBs needs to be addressed. Research in deductive query optimization has focused on the extension of logic-based query optimization to the deductive case (e.g. selection propagation in magic sets [34]) and heuristic join ordering strategies, such as evaluating literals with bound variables first (e.g. in Aditi, Coral, Declare, Glue-Nail).

In this paper, we describe rationale, implementation, and practical evaluation of a comprehensive query optimizer for the deductive object base ConceptBase [13, 29], a stable prototype system developed at RWTH Aachen and used by numerous communities for teaching, research and development tasks worldwide.

In contrast to most query optimizers of existing deductive database and object base systems, the ConceptBase query optimizer is cost-based, following a strategy that addresses the specific concerns of the kinds of applications mentioned above. The paper does not only justify the approach per se, but also its embedding in the overall system architecture and some experiences in real-world applications.

In Section 2, we describe the overall rationale of our approach from the viewpoint of metadata management, and contrast it with related work in the literature and in implemented systems. Section 3 presents a brief overview of the data model in ConceptBase, its logical semantics, and its storage structure, including the materialization of fan-out and histogram information. Based on this description, Section 4 elaborates the cost model while Section 5 presents details of the join ordering algorithm. In Section 6, the embedding of the method in the server architecture is discussed and the performance of the cost-based query optimizer is compared to that of the previous version which just used the traditional cost reduction heuristics. Section 7 offers some lessons learned and outlines future work.

## 2 OVERVIEW AND RELATED WORK

Several authors argue that cost-based join ordering strategies grow in importance in metadata of applications, as these tend to lead to large numbers of joins. A survey and simulation-based comparison of several methods has been

very recently conducted in parallel to, and independently from the work reported here [32].

Very few of the implemented deductive database systems support cost-based algorithms, and if, with very traditional cost estimation mechanisms. LDL supports the KBZ-heuristic [22] which is also used in the object-oriented main memory database WS-IRIS [23]. This approach, although known to be optimal for certain cases and working well with star queries, is criticized in [32] as it restricts the cost models to very simple ones, and was for basically the same reason not used in our work. Glue-Nail has implemented a version of Selinger's dynamic programming approach used also in the DB2 system [7] with platform-varying heuristics.

All of these approaches have in common that they rely on estimated *join selectivity* as the main basis for cost estimates. Join selectivity is a context-independent evaluation criterion that computes what percentage of the Cartesian product of two relations the join result will cover.

In contrast, we argue for a strategy based on the so-called *fan-out* of specific object or literal instances. Like some early work on logic program optimization [28] and some recent proposals for object-oriented databases [18], we see joins as directed rather than undirected operations. There are three reasons for this view:

- Complex object structures in meta databases often have *part-of*, *isA*, or *instanceOf* relationships which are multi-valued with wide variation of the fan-out from object to object. Thus, when combined with selection operations, join selectivity represents an average case which may say little about the actual distribution of matching value pairs. Cho et al. [4] propose an approach that makes partial participation of classes explicit, e.g. only certain subclasses participate in a join, distorting average selectivity measures.

- Many such systems, including ConceptBase, materialize these relationships in auxiliary structures that can be interpreted as extensions of join indexes [3, 17]; the effectiveness of such indexes has been shown impressively even for the relational case [26]. Taking this argument one step further, many systems also maintain other materialized views that can be reused in query optimization, leading to significant cost reductions.

- The view definitions in deductive object bases often contain subqueries that force a bushy rather than the traditional left-deep overall join order, and thus further strengthen the previous point. Important examples include the stratified handling of negated subqueries or subqueries with aggregate functions, but, at the level of cost functions or magic set algorithms, also recursive subqueries can be interpreted in this way.

In a sense, the fan-out approach requires a left-deep strategy in join order optimization which is known to not always find the optimal solutions [32]. However, the exploitation of materialized views or pre-computed subqueries introduces also an element of bushy query processing strategies. In ConceptBase, we use *Best First Search* to find the optimal solution where possible, augmented by simple heuristics beyond a time bound.

The variability of fan-outs also points to another crucial issue in query optimization that is neglected in most implemented deductive databases : the precision of cost estimates. Most systems, even commercial relational ones [10], still use the uniform distribution assumption; in queries with many joins, the errors of this kind of assumption tend to escalate [11]. To achieve the best possible precision, ConceptBase therefore follows a three-tier strategy:

- Where available, use the exactly known size of materialized views or pre-computed subqueries.

- Otherwise, maintain histograms about data distribution; following [12], end-biased histograms are employed as a rather precise and still cheaply maintainable data structure. Incidentally, ConceptBase maintains these diagrams simply as materialized aggregate views like any others. The object structure is in addition exploited to define special variants of the histograms differentiating the different kinds of semantic model abstractions (aggregation, instantiation, specialization).

- Only where relevant histograms are not available, fall back to their simplest special case – the uniform distribution assumption.

The implementation of these strategies in ConceptBase assumes a main-memory database approach, i.e. cost estimates are at the level of intermediate result sizes rather than secondary-storage page accesses. This assumption is justified in most ConceptBase applications where sufficiently large server memory is available; where it is not justified, some adaptions to the methods presented here would need to be developed.

## 3 THE DOB SYSTEM CONCEPTBASE

ConceptBase implements the O-Telos data model [24, 15]. The advantages of O-Telos are its very simple basic concepts, its straightforward notion of objects and their properties, its flexibility and genericity.

### 3.1 The Data Model

An O-Telos object base is semantically equivalent to a deductive database (Datalog with negation) which includes a predefined set of rules and integrity constraints coding the object structure. O-Telos is a frame based language, its rules and constraints are included as first-order formulas defined over a basic set of predicates describing the abstraction principles of instantiation, specialization and attribution.

A deductive (O-Telos) object base is a triple $DOB = (OB, R, IC)$ where $OB$, is the extensional object base, $R$ is a set of deduction rules, and $IC$ contains integrity constraints. An extensional O-Telos object base is a finite subset

$$OB \subseteq \{P(o,x,l,y) \mid o,x,y \in ID, l \in LAB\},$$

where $ID$ is a set of object identifiers and $LAB$ is a set of labels. The elements of $OB$ are called objects with identifier $o$, source and destinitation components $x$ and $y$ and label $l$. An individual object with the label $l$ has the form $P(o,o,l,o)$.

As a running example, we model a source file tree under version control. It contains 432 files in 53 directories. Figure 1 shows an excerpt of the corresponding database schema in the form of O-Telos frames. A `File` has a `type`, a `size` and is located in a `Directory`. A `SourceFile` is a special

```
Class File with           Class SourceFile isA File with
  attribute                 attribute
    type : FileType;          imports : SourceFile;
    name : FileName;          dependsOn : SourceFile
    size : Integer;         end
    dir : Directory
end                       Class Directory with attribute
                              subDir : Directory
                            end
```

Figure 1: An O-Telos example schema

file, which `imports` and `dependsOn` on other `SourceFiles`. Furthermore, a `Directory` may have subdirectories.

In the extensional object base, the instantiation of an object $x$ to a class $c$ is stored as $P(o,x,in,c)$, a specialization $c$ of a class $d$ is represented by $P(o,c,isa,d)$. As a shorthand for instantiation and specialization, we define the literals $In$ and $Isa$[1]:

$$\begin{aligned} P(o,x,in,c) &\rightarrow In(x,c). \\ In(x,d) \wedge P(o,d,isa,c) &\rightarrow In(x,c). \\ P(o,c,isa,d) &\rightarrow Isa(c,d). \end{aligned}$$

All other elements of the extensional object base represent attributes of certain objects. The attribution is expressed by the literal $A.m$ and is defined as follows:

$$\forall o,x,l,y,p,c,m,d \quad P(o,x,l,y) \;\wedge\; P(p,c,m,d) \;\wedge\; In(o,p) \rightarrow A.m(x,y)$$

where $p$ is called the attribute class of $A.m$. For example, the literal `A.dir(f1,d1)` states that the file `f1` is stored in the directory `d1`.

A rule in ConceptBase is defined as an attribute of a class whose value is a first-order predicate logic formula. The conclusion of a rule can either be an attribute relationship between objects, or the class membership of an object. The following example defines a recursive rule for the attribute `dependsOn` of the class `SourceFile`. A source file `s1` depends on a source file `s2` if it imports the source file `s2`, or if it imports a source file `s3`, which depends on the source file `s2`.

```
Class SourceFile with
rule
    dependsOnRule :
        $ forall s1,s2/SourceFile A.imports(s1,s2) or
          (exists s3/SourceFile A.imports(s1,s3) and
           A.dependsOn(s3,s2)) ==> A.dependsOn(s1,s2) $
end
```

Queries are defined as special classes, query classes, whose instances are the answer objects to the query. The membership conditions for a query class are expressed in two ways. First, superclasses restrict the set of possible answer objects to their instances. In addition, a query may have a *constraint*, a logic formula restricting the set of results. The predefined variable `this` in a constraint of a query refers to the answer objects of this query. The following query checks the integrity of the object base. It finds all makefiles, such that another makefile with the same name exists in the same directory.

[1] All variables are assumed to be universally quantified unless stated otherwise.

```
QueryClass BadMakefiles isA File with
constraint
    c: $ exists f/File A.type(f,MakeFile_Type) and
         A.type(this,MakeFile_Type) and
         not (IDENTICAL(this,f)) and
         ( exists d/Directory n/FileName
           A.dir(this,d) and A.dir(f,d) and A.name(this,n)
           and A.name(f,n)) $
end
```

Rules and constraints are compiled to Datalog$^{\neg 2}$ in a straightforward manner. The **`dependsOnRule`** of the example above, is translated into two Datalog rules, each representing one part of the Or-expression. The head of the rules is the conclusion literal **`A.dependsOn(_s1,_s2)`**.

```
A.dependsOn(_s1,_s2) :- A.imports(_s1,_s2).
A.dependsOn(_s1,_s2) :- A.imports(_s1,_s3),
                        A.dependsOn(_s3,_s2).
```

The query is translated into one rule with head **`BadMakefiles(_m1,_m2)`**. The body of the rule contains the literals used in the constraint. Usually, the quantification of variables leads to $In$-literals, which often can be eliminated from the rule bodies because they are already guaranteed by the corresponding $A.p$ literals. This semantic optimization is done at compile time.

```
BadMakefiles(_m1,_m2) :-
    A.type(_m1,MakeFile_Type), A.type(_m2,MakeFile_Type),
    A.dir(_m1,_dir), A.dir(_m2,_dir),
    A.name(_m1,_name), A.name(_m2,_name),
    not(IDENTICAL(_m1,_m2)).
```

The problem of query optimization is to find an order for the literals in the body of a rule which has the minimal evaluation costs. We will continue to use this query in the following for demonstrating the principles of the ConceptBase query optimizer.

### 3.2 Statistical Profiles as Materialized Views

A statistical profile describes the quantitative properties of the data stored in an object base. The statistical profile consists of object profiles and histograms. Object profiles are attached to every object in the object base, whereas histograms are only relevant for objects representing attribute classes.

The storage manager of ConceptBase stores each P-tuple as a C++-object, representing an (inverted) indexed access structure as shown in Figure 2. The object has pointers to the source and destination objects of the P-tuple, and a pointer to the label, which is stored in a symbol table. These pointers are represented as the three boxes **`src`**, **`lab`** and **`dst`** in the top of the box in Figure 2.

For an efficient computation of the literals $In$, $Isa$ and $A.p$, each object has a set of incoming and outgoing links for instantiation, specialization and attributes. These sets are shown as round boxes in the figure and can be exploited as a kind of join index.

The object profile stores the cardinality of each of the six sets for an object. This information is used to estimate the number of solutions of a literal.

Histograms are used to estimate the size of the result of a $A.p$ literal. Although we store the size of sets for the incoming and outgoing attribute links, it is not sufficient to

[2] As usual we use leading underscores to distinguish variables from constants.

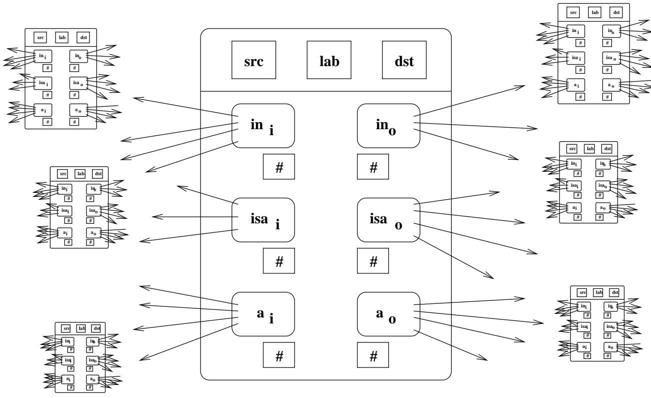


Figure 2: Storage structure for Telos objects

estimate the size of the result of an $A.p$ literal, if one argument of the literal is a constant. Usually, query optimizers assume in such a case uniform distribution for the values of a literal. As this assumption can lead to very inaccurate estimated costs, histograms can be attached to objects representing attribute classes, if the values of the attribute are not uniformly distributed. The use of histograms in query optimization is explained in more detail in Section 4.4. Both the object profiles and the histograms can be understood as certain kinds of materialized views which should reflect the data value distribution in a current object base. While the cardinalities in the profile are steadily maintained histogram updates are executed on explicit demand from outside only.

## 4 COST ESTIMATION OF QUERY EXECUTION PLANS

In cost-based query optimizers a cost function is applied to alternative query execution plans (QEPs) for a given query, then a plan with low expected costs is selected by a search strategy.

In classical relational systems the time-critical bottleneck when evaluating a query typically is to locate data on secondary memory and to transfer this data to main memory. In contrast, ConceptBase was designed as a main memory database. This justifies our assumption, that the query evaluation costs are dominated by the computation costs. For these costs the main factor is the size of the operands. Our cost-function therefore follows a classical approach: The evaluation costs of a query evaluation plan are defined as the sum of the intermediate result sizes [27, 28].

### 4.1 The Cost Function: Basic Considerations

Our first cost function describes the special case of an extensional $Datalog^{\neg}$ query.

**Definition 4.1** (Cost function $C_0$)
Let $S$ be a sequence of $n$ extensionally defined literals: $S = < l_1(\vec{a_1}), \ldots, l_n(\vec{a_n}) >$ and let $S_i$ denote the prefix $< l_1(\vec{a_1}), \ldots l_i(\vec{a_i}) >$ of $S$ and $Size_0(S_i)$ the size of the extensional result of $S_i$. Then $C_0$ defines the costs for the evaluation of $S$ as

$$C_0(S) = \sum_{i=1}^{n} Size_0(S_i)$$

□

**Example 4.1** (Perceived and Actual Costs)
Figure 3 demonstrates the relation between the perceived costs of evaluating a query (cpu time in seconds) and the measured value of $C_0$. For the example query `BadMakeFiles` 360 different plans were evaluated and the evaluation time was recorded. In a separate measurement the sum of the intermediate results for each plan was protocolled. In figure 3 the plans are sorted by this value. The strong relation between evaluation time and intermediate result sizes is obvious. □

In deductive databases, some of the literals occuring in a literal sequence might be defined via deductive rules. Solutions found via deductive rules increase the size of the intermediate results and they also increase the evaluation costs, because these additional solutions are produced via evaluation of literal sequences.

**Definition 4.2** (Cost function $C$)
Let $S$ be a sequence of $n$ literals: $S =< l_1(\vec{a_1}), \ldots, l_n(\vec{a_n}) >$ and let $S_i$ denote the prefix $< l_1(\vec{a_1}), \ldots, l_i(\vec{a_i}) >$ of $S$ and $Size(S_i)$ the size of the extension of $S_i$. Then $C$ defines the costs for the evaluation of $S$ as

$$C(S) = \sum_{i=1}^{n} Size(S_i) + \sum_{i=1}^{n} Eval(l_i)$$

where $Eval(l_i)$ is defined as:

$$Eval(l_i) = \begin{cases} 0 & \text{if } l_i \text{ is extensionally defined} \\ \sum_{j=1}^{k} C(T_j) & \text{if } l_i \text{ is defined via the} \\ & \text{literal sequences } T_1, \cdots, T_k \end{cases}$$

□

To compute a cost estimate according to this definition, for each literal sequence $T$ that defines (head) literals occuring in $S$ two values must be available for each possible instantiation pattern of their head literal: the estimated result size $Size(T) = Size(T_n)$ and the estimated evaluation costs $C(T)$. If $l_i$ matches the head literal of a literal sequence $T$, then $Size(T)$ is needed for the computation of $Size(S_i)$ and $C(T)$ is taken into account when computing $Eval(l_i)$.

The evaluation costs of the literal sequence $S$ can be computed directly via this definiton, if the evaluation costs and result sizes are known for the sequences $T_1, \cdots, T_k$ referenced by $S$. Two problems arise with this definition: cyclic dependencies between rules, and the addition of rules for the literals occuring in $S$ to the rule base after the costs of $S$ have been computed. The first problem is handled by a heuristic during cost computation shortly explained below. If a new rule is added to the rule base, the optimizer looks for all rules using the conclusion literal in their body. These rules have to be re-optimized after the new rule has been optimized, in order to respect the changed evaluation costs of this body literal.

Classical query optimizers [27] use the concept of *selectivity* to estimate the result size of a join operation:

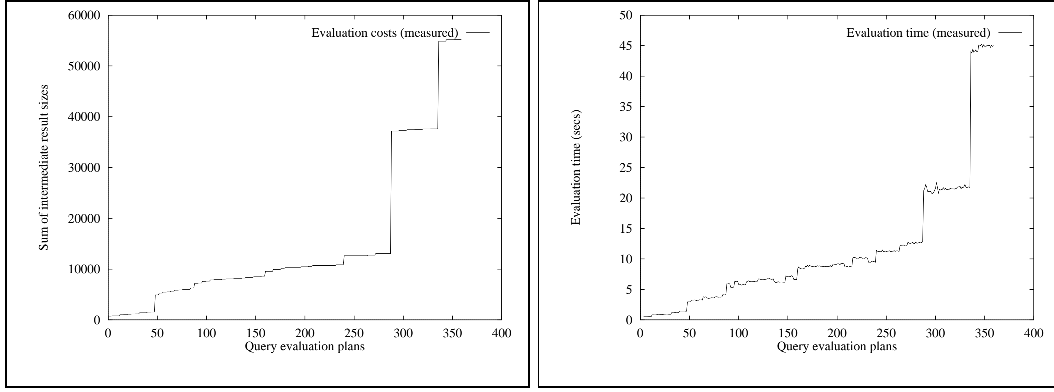


Figure 3: Measured costs vs. evaluation time

**Definition 4.3** (Selectivity)
Let $R$ and $S$ be two relations with size $n_R$ und $n_S$ and let $n_{R,S}$ be the size of the result of joining $R$ and $S$ with the join-predicate $A = B$ for two attributes $A$ and $B$ of $R$ and $S$ respectively. Then

$$s(R, S, A = B) = \frac{n_{R,S}}{n_R \cdot n_S}$$

is the selectivity of this join. □

Usually the following simplifying assumptions are made [33]:

- uniform distribution of the values in $A$ and $B$
- statistical independence of the values in $A$ and $B$
- an inclusion dependency between $A$ and $B$ or vice versa

The third assumption is needed because in general there exists no relation-independent domain concept for the attributes composing the relations.

In OODBs such as ConceptBase, the third assumption may be dropped, because an equi-join between two literals is expressed by a common variable of these literals and the cardinality of the domain (class) of this variable is known from the statistical profile. If we denote this common domain of $A$ and $B$ with $dom(A)$, then the selectivity can be estimated as:

$$s(R, S, A = B) = \frac{1}{|dom(A)|} \tag{1}$$

## 4.2 Estimating the Fan-Out

We now define the central function for the computation of *Size*: the *fan-out* of the literal $l_i$ in the sequence $S$. As preparation we define the unrestricted extension of a literal. We restrict our discussion here to the positive occurences of $In$, $A.p$, $Isa$ and $P$-Literals, negated and arithmetic literals are discussed later.

**Definition 4.4** (Unrestricted Extension)
If $l_i$ is the functor of a literal, then the unrestricted extension of $l_i$ $l_i(\vec{free})$ contains the set of all valid instances of $l_i$ in the database, either stored extensionally or derived via deductive rules. □

For the computation of the fan-out, we assume that the size of this unrestricted extension is known. The size of the extensional part is stored in the statistical profile. For the intensional solutions we assume that their quantity has been estimated by prior cost computations for all applicable literal sequences.

**Definition 4.5** (Fan-Out)
Let $S =< l_1(\vec{a_1}), \ldots, l_n(\vec{a_n}) >$ and $l_i(\vec{free})$ denote the unrestricted extension of literal $l_i$. If $\mathcal{B}_i$ is the union of the constants occuring in $\vec{a_i}$ and those variables in $S_{i-1}$ which occur among the $a_i$, and if $dom(v)$ denotes the class to which a variable or constant $v$ is bound, then the fan-out of $l_i(\vec{a_i})$ in S is defined as

$$fo(l_i(\vec{a_i}), \mathcal{B}_i) = \frac{|l_i(\vec{free})|}{\prod_{b \in \mathcal{B}} |dom(b)|}$$

□

The fan out of an extensional literal in a given sequence is directly related to the selectivity. Let $L =< l_1(\vec{a_1}), l_2(\vec{a_2}) >$ be a join operation defined by the common variable $v$ with domain (Class) $V$, then the result size of $L$ is estimated as:

$$n_{l1,l2} = n_{l1} \cdot n_{l2} \cdot \frac{1}{|V|} = |l_1(\vec{a_1}))| \cdot fo(l_2, \{v\})$$

**Example 4.2** (Fan-Out vs. Selectivity)
We assume an example database containing 432 files of 12 different types, where every file has a unique type. The estimation of the cardinality of $S =< \texttt{A.type}(_F_1, _T), \texttt{A.type}(_F_2, _T) >$ is according to equation 1:

$$n_{l1,l2} = |\texttt{A.type}| \cdot |\texttt{A.type}| \cdot \frac{1}{|\texttt{FileType}|} = 432 \cdot 432 \cdot \frac{1}{12} = 15552$$

When using the fan-out of $\texttt{A.type}(_F_2, _T)$ in the given sequence, the estimate is:

$$\begin{aligned} n_{l1,l2} &= |\texttt{A.type}(_F_1, _T)| \cdot fo(\texttt{A.type}(_F_2, \texttt{T}), \{_T\}) \\ &= 432 \cdot \frac{|\texttt{A.type}(\vec{\texttt{free}})|}{|\texttt{FileType}|} = 432 \cdot \frac{432}{12} = 15552 \end{aligned}$$

□

Though both equations in the example deliver the same result, fan-out and selectivity are slightly different concepts. Selectivity interprets a join as selection from the cross product of two relations. When using the fan-out, the right hand literal can be seen as operand, that is applied on the left hand literal. In contrast to the selectivity the fan out of a literal directly allows to distinguish between "good" and "bad" literals in a sequence: literals with a fan-out less than 1 are "good", because they reduce the intermediate result size, while literals with a fan-out larger than 1 enlarge the intermediate result size.

*Negated literals* are evaluated according to the "closed world assumption": $not(l(\vec{a}))$ can be evaluated only if every argument in $\vec{a}$ is bound. In this case the fan-out of $l(\vec{a})$ is (theoretically) guaranteed to be less than or equal to 1, and we define the fan-out of $not(l(\vec{a}))$ as $fo(not(l(\vec{a})), \mathcal{B}) = 1 - fo(l(\vec{a}), \mathcal{B})$. In practice the fan-out $fo(l(\vec{a}), \mathcal{B})$ can be greater than one because of the additional solutions produced by deductive rules. In this case, the computation above would result in a negative value. Therefore $fo(l(\vec{a}), \mathcal{B})$ in this case is restricted to the solutions stored extensionally and the intensional fan-outs are added afterwards to $fo(not(l(\vec{a})), \mathcal{B})$. Although it is theoretically not correct, it can be justified by the higher computation costs caused by deductive rules.

The fan-out of the *arithmetic comparison literals* $LT$ ($<$), $GT$ ($>$), $LE$ ($\leq$), $GE$ ($\geq$) is estimated as 0.5, the fan-out of $EQ$ ($=$) is estimated as $\frac{1}{|\texttt{Integer}||\texttt{Integer}|}$ and the fan-out of $NE$ ($neq$) is estimated as $1 - \frac{1}{|\texttt{Integer}||\texttt{Integer}|}$[3]

*Cyclic dependencies* are handled similar to negated literals. If the query optimizer detects a cycle in the rule dependency graph, all recursively defined literals are removed from the rules that occur in the cycle. After the optimization of the reduced rules, the literals are inserted at the end of the sequence, because we assume high costs to evaluate recursive literals.

### 4.3 The *Size* Function

The *Size* function used in Definition 4.2 can be defined as follows using the fan-out:

**Definition 4.6** (Size)
Let $S_n =< l_1(\vec{a_1}), \ldots, l_n(\vec{a_n}) >$ be a literal sequence and let $S_i$ denote the prefix with length $i$ of this sequence. Then $Size(S_i)$ is defined as

$$Size(S_i) = \begin{cases} fo(l_1(\vec{a_1}), \mathcal{B}_1) & : \quad i = 1 \\ Size(S_{i-1}) \cdot fo(l_i(\vec{a_i}), \mathcal{B}_i) & : \quad i > 1 \end{cases}$$

Since we consider rule sets, $S_n$ may occur as body of a rule with head literal $l$. This literal $l$ on the other hand can be used in other rule bodies with specific instantiation patterns. Therefore $\mathcal{B}_1$ not only contains the constants of $l_1$ but also its variables that occur as bound variables in $l$ already.[4] □

This definition fills the gap left in Definition 4.2 and our cost function is completely defined now. The following concluding example illustrates the cost estimation, when deductive rules are applied.

[3]The extension of class `Integer` contains all integers occuring as attribute values for a stored object.

[4]As an obvious consequence we have to optimize each rule body wrt. *all* relevant instantiation patterns of its corresponding head.

**Example 4.3** (`BigFile`)
Assume that the class `BigFile` is defined via the rule

```
In(_F,BigFile) :- A.size(_F,_S),GT(_S,10000)
```

then the estimated cost for this sequence would be

$$\begin{aligned} C(< \texttt{A.size}(_F, _S), \texttt{GT}(_S, 10000) >) &= Size(< \texttt{A.size}(_F, _S) >) \\ &+ Size(< \texttt{A.size}(_F, _S) >) \cdot fo(\texttt{GT}(_S, 10000), \{_S, 10000\}) \\ &= fo(\texttt{A.size}(_F, _S), \emptyset) \\ &+ fo(\texttt{A.size}(_F, _S), \emptyset) \cdot fo(\texttt{GT}(_S, 10000), \{_S, 10000\}) \\ &= 432 + 432 \cdot 0.5 = 648 \end{aligned}$$

if `_F` is free and

$$\begin{aligned} C(< \texttt{A.size}(_F, _S), \texttt{GT}(_S, 10000) >) &= Size(< \texttt{A.size}(_F, _S) >) \\ &+ Size(< \texttt{A.size}(_F, _S) >) \cdot fo(\texttt{GT}(_S, 10000), \{_S, 10000\}) \\ &= fo(\texttt{A.size}(_F, _S), \{_F\}) \\ &+ fo(\texttt{A.size}(_F, _S), \{_F\}) \cdot fo(\texttt{GT}(_S, 10000), \{_S, 10000\}) \\ &= 1 + 1 \cdot 0.5 = 1.5 \end{aligned}$$

if `_F` is bound to an object of class `SourceFile`. The estimated size of the Class `BigFile` would be $Size(S_2, \{_F, _S\}) = 216$.

If the literal `In(F,BigFile)` occurs in a conjunction, than the fan-out of it can be estimated as 216 if `_F` is free and e.g. as 0.5 if `_F` is bound to an object of class `SourceFile`. The estimate of the evaluation costs for the conjunction are increased by 648 ($= Eval(\texttt{In}(_F, \texttt{BigFile}))$) if `_F` is free and by 1.5 if `_F` is bound to an object of class `SourceFile`. □

### 4.4 Histograms

The above formulas base costs on the assumptions of uniform distribution and statistical independence. In some cases more detailed distribution information can be applied. For $A.p$ literals, the distribution of values for the source and destination component can be stored in the statistical profile. This information can be used directly if one of the arguments of an $A.p$ literal is a constant. If the other argument is a free variable, the histogram entry is the exact fan-out, if the other argument is bound, the fan-out can be computed based on the number of solutions for the constant rather than on all solutions of $A.p$ as the following example illustrates:

**Example 4.4** (Use of Histogram Entries)
Assume that the directory `d` occurs 20 times as destination component in the literal `A.dir` and that 400 source-file objects and 80 directories are stored in the database. Then according to the histogram entries for the destination component of `A.dir` the fan-out of `A.dir(_F,d)` is 20 if `_F` is a free and can be estimated as $\frac{20}{400} = \frac{1}{20}$ if `_F` is bound. The latter value expresses the probability, that the value of `_F` is among the 20 values, which fulfill $\texttt{A.dir}(_F, \texttt{d})$. If the histogram would not be available, the constant `d` in the cost estimate has to be treated like a bound variable whose concrete value is unknown. Then the fan-out would be $\frac{400}{80} = 5$ if `_F` is free and $\frac{400}{80 \cdot 400} = \frac{1}{80}$ if `_F` is bound. □

The histograms stored in the statistical profile are *end-biased* histograms ([12]). This type of histograms requires less memory than normal histograms, because they represent distribution information in a compressed form. The variant used here represents some objects of the highest and

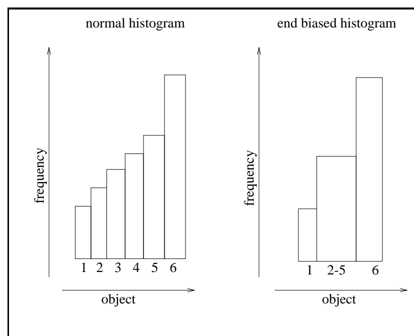


Figure 4: Normal histograms vs. end-biased histograms

lowest frequency (the "ends" of the histogram) explicitly and uses one average frequency for the remaining values. Figure 4 illustrates this principle.

Furthermore, the number of tuples in the result of a binary equi-join $A.p_1(A,B), A.p_2(C,B)$ with $A$, $B$ and $C$ free can be known exactly if histograms for $A.p_1$ and $A.p_2$ exist, as the following example illustrates:

**Example 4.5** (Uniform Distribution vs. Histograms)
Consider the sequence $S = < \texttt{A.type}(_A, _B), \texttt{A.type}(_C, _B) >$ and a database with 10 files of type `a`, 10 files of type `b` and 70 files of type `c`. Using the uniform distribution assumption, 30 files would be assumed to exist for each type. $C(S_2)$ would then be estimated as $Size(S_1) + Size(S_2) = 90 + 90\frac{90}{3} = 2790$. But we know that for each of the file of type `a` in the first relation, 10 matching files in the second relation exist. If we take the histogram for the destination component of `A.type` as vector and take the scalar-product of this vector by itself, we obtain the exact size of the extension of $S_2$: $Size(S_2) = 10 \cdot 10 + 10 \cdot 10 + 70 \cdot 70 = 5100$ and $C(S) = Size(S_1, S_2) = 5190$. □

Better approximations can be reached by using *specialized* and *dynamic* histograms. Specialized histograms have attached selection literals for a certain column, i.e. they specify distributions for subsets of column entries. Dynamic histograms allow the continuous maintenance of distributions during the estimation by operations executed on the histograms, e.g. specializing columns wrt. subclasses of values.

### 4.5 Exploitation of Materialized Views

The usage of histograms described before is one possible way of improving the accuracy of cost estimate. The cost function implemented additionally exploits further background knowledge stored in the system catalogue, for example knowledge about the size of materialized views. Views in ConceptBase are defined like query classes and are translated to Datalog rules in the same way as queries and deductive rules. If the body of a Datalog rule for a materialized view is a subset of the current set of literals to optimize, the statistical knowledge about the extension of the view is employed during cost estimate as shown in the following example. Details of the necessary query subsumption algorithm can be found in [29].

**Example 4.6** (`BigFile` as Materialized View)
Assume that the class `BigFile` defined in Example 4.3 would have been materialized as view, than the number of files in this view is known. Now consider the literal sequence

$\texttt{A.size}(_F, _S), \texttt{GT}(_S, 10000), \texttt{A.type}(_F, \texttt{prologFile})$.

The optimizer would detect that the first two literals match the view definition of `BigFile`. Then $Size(S_2)$ would be estimated as the number of tuples contained in the `BigFile` view. This quantity is known exactly. □

### 4.6 Assumptions and Overall Cost Estimation Strategy

The predecing examples illustrated the application of more sophisticated statistics for obtaining cost estimates which avoids significant underestimates as in example 4.5. The decision about applicability of available histograms and materialized views is based on the following considerations: For a given set of literals the result size computed is always the same. If for example literal $A.p_1(A,C)$ precedes the literal $A.p_2(B,C)$ in a literal sequence, but some literals are in between we can procede as in example 4.5 if $A$ and $C$ are free variables in $A.p_1(A,C)$, $B$ is still a free variable in $A.p_2(B,C)$ and the corresponding histograms for $A.p_1$ and $A.p_2$ exist. The relative frequency of the objects occuring as instances of $C$ before the evaluation of $A.p_2(B,C)$ is assumed to be the same as described by the histogram for the destination component of $A.p_1$. For the computation of the size of the intermediate result up to $A.p_2(B,C)$ the histogram is multiplied with the product of the fan-outs of the literals between $A.p_1(A,C)$ and $A.p_2(B,C)$. This relies on the assumption of *uniform selection*, which states, that the shape of the distribution of the instances of a class is kept during evaluation of the literals in between. This assumption treats the sequence above as if the literals between $A.p_1(A,C)$ and $A.p_2(B,C)$ were *not* in between these two literals, but follow $A.p_2(B,C)$ in the sequence.

A similar argument is used for the exploitation of materialized views. If for a literal sequence a matching view definition has been found, the size of the view is multiplied by the fan-out of the remaining literals.

The overall cost function of the ConceptBase query optimizer applies the following strategy:

1. if a view based estimate is applicable, use it
2. if a histogram based estimate is applicable, use it
3. else use the "standard" fan-out method

## 5 SEARCH STRATEGIES AND HEURISTICS

The computational costs for query optimization are dominated by the search space. Computation of the cost-function for a given literal sequence can be done in linear time (linear in the number of literals). But even when only considering left-deep evaluation, for a sequence of n literals there are n! possible literal sequences. Therefore, the selection of a suited search algorithm is a crucial issue for building a query optimizer. In addition heuristics can be employed to restrict the number of possible QEP's that are considered by this algorithm.

### 5.1 Searching the Optimal QEP

Search algorithms can be categorized into global and local approaches. While global methods consider the whole search

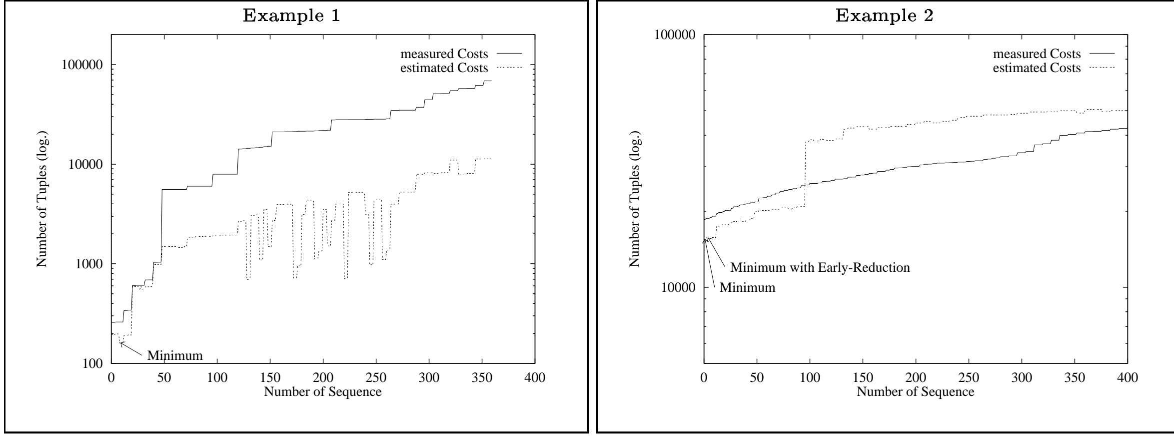


Figure 5: Approximation of measured costs

space, local methods start with a certain element of the search space and try to satisfy the search goal by looking at its neighbours or other elements reachable by following certain criteria. As an example, iterative improvement allows only "moves" in the search space that in our case improve, i.e., reduce, the evaluation costs of the considered QEP's. Whether local approaches are sufficient for a given search problem depends on the function used to assign (estimated) costs to each element of the search space. In our case, the estimated costs of a QEP in general have no monotonicity property compared to the measured costs. Therefore a local minimum is far away from providing a guaranteed near-optimal solution.

For illustrating this, Figure 5 shows two different types of cost curves relevant for two sample queries. As in Section 4 we measure the costs of a literal sequence by explicitly summing up the intermediate results, order the sequences by increasing costs, and compare them with the costs resulting from estimation. It should be noted that typically the measured costs are completely out of consideration when discussing the selected search strategies in papers on query optimization. Once the cost framework has been defined the only goal is to find the minimal estimated costs without further looking at the "ideal" cost measure that should be approximated. We can observe that in the right part of figure 5 the estimated costs grow with the same tendency than the measured costs. Only for these cases local approaches work fine, otherwise the local minima are bad solutions. Of course the search algorithm does not process QEP's in the order with respect to the measured costs which are in fact not available. However, any processing order would have to deal with similar "valleys" of the corresponding estimated costs.

Hence, we decided to combine our cost function with a global (and complete) search method, but stay pragmatic: The optimizer performs a *best-first* algorithm until a certain upper time bound is reached. Then the execution is suspended and continued as *cheapest-first* search.

The best-first search proceeds as proposed in [28]: Starting with a set $S$ of literal sequences of length one formed by the literals of the query, the sequences are sucessively extended by including literals not contained before. Selection criterion for a sequence to be further expanded are minimal costs compared to all other sequences currently available. If the sequence already contains all literals then it represents the optimal QEP. Otherwise it is replaced in $S$ by its successor and its neighbour with minimal costs among all other successors and neighbours. The neighbours $nb(s,R)$ of a sequence are all sequences identical to $s$ apart from the last element but containing one of the other literals (in $R$) not yet included in $s$. In the same way the successors $succ(s,R)$ are built by appending one literal from $R$ to $s$.

**Algorithm 5.1** (Best-First search on QEP's)
Input: A literal set $L = \{l_1, ..., l_n\}$
Output: A sequence of literals from $L$ with estimated costs or **nil**

**begin**
$S$ := $\{< l_1 >, ..., < l_n >\}$;
$i$ := 0;
**repeat**
  **select** $s \in S$ **such that**
    $Cost(s) = min\{Cost(y) \mid y \in S\}$;
  $S$:=$S \setminus \{s\}$;
  $R$:=$L \setminus \{l_j \mid l_j \in s\}$;
  **if** $R = \emptyset$ **then return**$(s, Cost(s))$
   **else**
    **select** $n_1 \in nb(s,R)$ **such that**
      $Cost(n_1) = min\{Cost(x) | x \in nb(s,R)\}$;
    **select** $n_2 \in succ(s,R)$ **such that**
      $Cost(n_2) = min\{Cost(x) | x \in succ(s,R)\}$;
    $S \cup \{n_1, n_2\}$ **fi**
  $i$ := $i+1$
**until** $i = max$;
**return(nil)**
**end**

The constant $max$ limits the number of iterations and avoids the worst case where best-first checks the whole search space and behaves like naive enumerating all possible QEP's. Compared to checking costs of several partial QEP's as candidates for an extension during best-first search, an eventually subsequent cheapest-first search starts with the cheapest single literal and stepwise includes the other literals with increasing costs required for their evaluation based on the existing partial sequence.

### 5.2 Heuristics

The cost-based approach is complemented by certain "Early-Reduction" heuristics. These heuristics aim at identifying literals that can be evaluated with low costs and which are suited to reduce intermediate result sizes as early as possible. The first two categories of heuristics eliminate literals before executing the cost-based optimization and insert them afterwards, the others perform an additional reordering of literals positioned in the optimized sequence.

**Literals with constant arguments** have always the same fan-out. They don't produce solutions but represent simple test conditions which make an evaluation of further literals obsolete if the test fails. Therefore these literals can be eliminated from the sequence and inserted later at the beginning of the optimized QEP.

**Negated literals** must only contain variables that have been bound before. They elimininate tuples from an existing intermediate result, i.e. have a Fan-Out of less than one. Negated literals are elimininated before the cost-based optimization starts and inserted at the earliest position in the optimized sequence where the safety condition is fulfilled. In the same way we handle also arithmetic comparison literals or aggregate functions which usually require the same safety condition.

**Selection literals** Literals that have in all possible orderings at most one free variable $x$ perform selections on all possible values for $x$ wrt. the common occurence with the other arguments. If $x$ is free, it is in fact an *active selection*, otherwise *passive*, since no new solutions for $x$ are produced and the size of intermediate results will decrease or remain the same atleast. The distinction between active and passive selections can be done *after* computing the optimized sequence only. We can then reorder the passive selections such that they contribute to reducing the result sizes as early as possible without loosing their passive property, i.e. inserting them in the optimized sequence directly after $x$ is freshly bound.

**Selection chains** Literals with arguments that have a functional dependency on some other argument only occuring in a set of passive selections, form together with this set so called *selection chains*. This holds in particular in the O-Telos context for literals $A.p(x, y)$ with a single-valued attribute $p$ wrt. $y$. If all other arguments $x$ of the literal are bound the intermediate result remains the same and is even reduced by the passive selections. Complete selection chains can therefore removed from the optimized plan and inserted in the same way as single passive selections.

We don't use explicitly a connectivity heuristics which is often employed in deductive query optimizers to avoid cross products by requiring shared variables for two neighboured literals in the sequence. The fan-out criterion automatically gurantees this property whenever possible. The application of heuristics of course affects the optimality of the final literal sequence. On the other hand optimization costs can be drastically reduced. If in a set of $n$ literals $k$ such literals are removed, the search space is reduced from $n!$ to $(n-k)!$ possible sequences. Furthermore, negated literals often have similar costs close to one. This similarity might reduce the performance of the best-first search, when these literals are included.

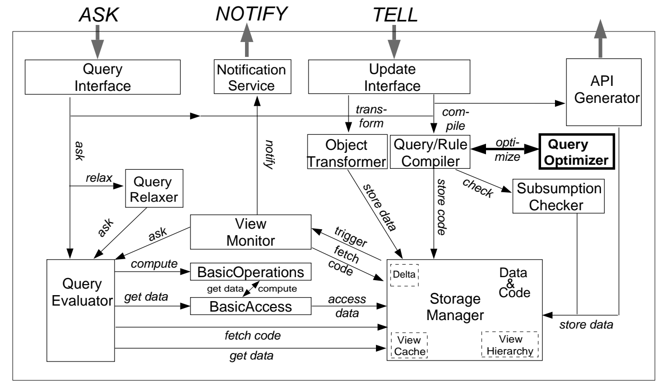


Figure 6: The extended ConceptBase architecture

## 6 IMPLEMENTATION AND OPTIMIZATION RESULTS

The techniques presented above were implemented in the ConceptBase query optimizer [29]. In the following we sketch the extended implementation architecture, illustrate the accuracy of cost estimation and its relationship to the amount of information available to the optimizer, and provide some figures demonstrating the actual runtime improvements.

Figure 6 shows how the query optimizer is embedded in the ConceptBase server architecture [13]. The `Query/Rule Compiler` is responsible for the actual code generation. One of its intermediate representations is Datalog and the task of reordering the body literals in rules is taken over by the `QueryOptimizer`. The resulting code is stored together with the low level and indexed object representation. The execution of query evaluation plans is performed by the `QueryEvaluator` which accesses the `StorageManager` both for fetching code and data. Its core component is a standard fixpoint procedure for the evaluation of algebraic equations and is based on a dedicated algebra (`BasicOperations`).

### 6.1 The Accuracy of Estimated Costs

In Section 4 we discussed how the basic cost function in ConceptBase can respect additional information such as histograms and materialized views. We will demonstrate the relationship beween accuracy of the cost estimation and the eploited additional information with a sample query from the first application: The Query `BadMakeFiles` searches makefiles with ambiguous names in the same directory (cf. Section 3. The very uneven distribution of file types in directories makes this query an interesting challenge for testing the optimizer.

We look at the cost estimation for `BadMakeFiles` with respect to four different constellations of background information available to the optimizer. After applying the early reduction heuristics 360 possible QEP's remain. Figure 7 compares the estimated costs with the costs measured by directly computing the cost function with the actual *Size* values for all intermediate results.

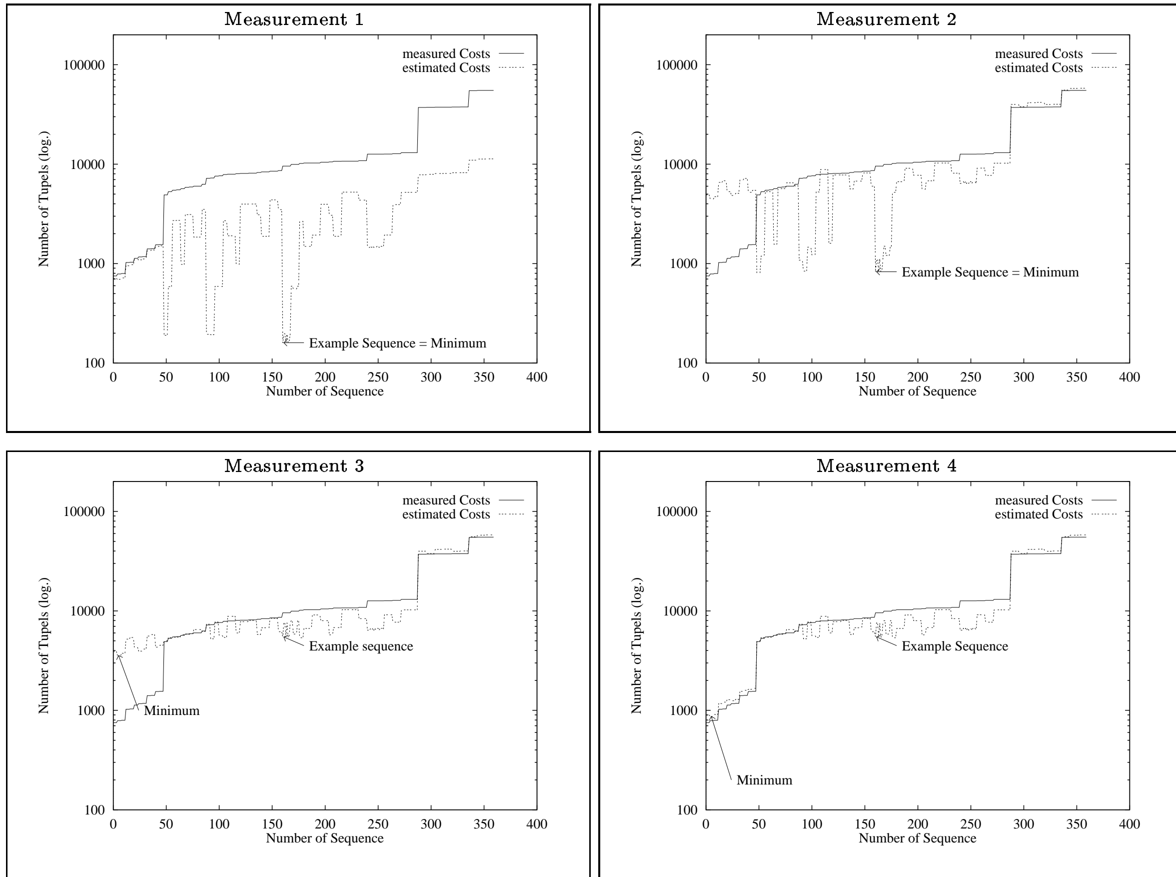


Figure 7: Accuracy of Estimation

**Measurement 1** The cost estimation relies on the uniform distribution assumption only. Obviously we get a clear underestimation, in particular for the minimum value.

**Measurement 2** For the attributes occuring in the query the optimzer can access histograms and assumes uniform selection. In certain parts of the curve we get a better approximation.

**Measurement 3** In addition to histograms the optimizer also respects two materialized views which correspond to certain subexpressions of `BadMakeFiles`. Although still underestimated the global minimum is found.

**Measurement 4** Employing specialized and dynamic histograms (maintained during the QEP analysis) yields a further improvement of the cost approximation, in particular in the lower part of the curve.

Fig. 7 illustrates that more information available to the query optimizer yields a higher accuracy for the estimated costs of QEP's. The main goal of query optimization, however, lies in avoiding bad QEP's, i.e. the chosen QEP with minimal estimated costs should be as near at the real minimum as possible. This does not always require full materialization.

## 6.2 Practical Impact

For validation purposes we employ two applications: The first (application I) concerns the software source and change management for the ConceptBase system development. The second (II) stems from a business process modelling project carried out with ConceptBase. Very similar to the program library example used in the previous sections we defined for application I an O-Telos schema which models all necessary entities relevant for recording the development history and the relationships between the software components. The object base contains about 38,000 objects for the 432 modules of the system. In application II, ConceptBase was used by a German consulting company to record actual state and future requirements analyses and in particular for stepwise refining the resulting models. This task was supported by a set of about 80 query classes that served for analysing typical lacks of given models [25].

For both applications we undertook measurements for a number of queries and view definitions. Without explaining their semantics (see [37, 29]), Tables 1 and 2 compare the query evaluation time ($t_{new}$) for a subset of them with the time $t_{old}$ needed by the same ConceptBase release without the cost-based optimizer. The cost of optimization are covered by $t_{opt}$. As in measurement 1 of 6.1 we only use the uniform distribution assumption. The search of QEP's is done with Best-First. The acceleration factor $c$ results from

dividing the old by the new response time.

**Application I:** For each query we measured the running time for all possible QEP's. Table 1 contains the best ($t_{best}$) and worst ($t_{worst}$) values found.

**Application II:** The measured values (Table 2) show a significant improvement of the response time compared to using the previous ConceptBase optimizer which applied only some very rough heuristics. In some cases we observe a negative overall effect if we look at $t_{new} + t_{opt}$. This can be explained through the anyway good QEP selected before using the rough heuristics or just by accident. Additional efforts for systematical optimization in these cases yield to relatively small improvements only. The optimizing time can in particular neglected for views which are stored together with a fixed QEP to be executed whenever the view is accessed.

## 7 CONCLUSIONS

We have described design and implementation of the ConceptBase query optimizer which explicitly takes into account the complexities introduced by the combination of complex structures and sophisticated analytic rule-based capabilities required by meta data managers in applications such as data warehousing or information systems design.

Our cost-based query optimization approach is based on the asymmetric principle of fan-out rather than the symmetric principle of join selectivity. It supports this principle with adapted indexing and view materialization techniques that do not only accelerate query execution but also improve cost estimates. Experiences in a number of real-world applications demonstrate the value of these more precise cost estimates as well as the performance advantages over purely heuristic strategies used in most implemented deductive databases.

A number of issues remain to be investigated. The current query *execution* strategy in ConceptBase supports ad-hoc queries only, even though query *optimization* is heavily based on query reuse. A compiled approach that requires less iteration between the reasoning/planning level and the object store level should result in a significant, albeit constant-factor improvement, of query performance in database application programs.

More fundamentally, the experiments in Section 6 show that even though more precise statistical profiles and materialized views are important in approximating the overall cost estimates, the true goal of query optimization is just to find some plan which in practice has close to minimal costs. The question of what views to materialize and what histograms to keep is, similar to the general question of view materialization strategies, still subject of ongoing research.

**Acknowledgments.** This work was supported in part by the Swiss Federal Office of Professional Education and Technology under grant KTI-3979.1 (SMART) and by the European Commission under ESPRIT Long Term Research Project DWQ (Foundations of Data Warehouse Quality).

| Query | $t_{new}$ | $t_{best}$ | $t_{worst}$ | $t_{opt}$ | $t_{old}$ | $c$ |
|---|---|---|---|---|---|---|
| BadMakeFiles | 8.4 | 0.48 | 45.00 | 0.3 | 33.11 | **3.9** |
| BadPrologFiles | 0.23 | 0.23 | 58.45 | 0.3 | 32.87 | **142.9** |
| SameComment | 7.35 | 7.35 | 426.65 | 0.64 | 80045.3 | **10890.5** |
| MultiUser | 12.44 | 11.56 | 48.89 | 2.26 | 446.02 | **35.9** |
| DoubleSize | 0.9 | 0.82 | 10.45 | 0.85 | 12.85 | **14.3** |
| QuixAndMe | 1.6 | 1.51 | 19.51 | 0.37 | 5.74 | **3.6** |
| PowerUser | 0.47 | 0.47 | 7.62 | 0.08 | 1001.42 | **2130.7** |

Table 1: Optimization results I

| Query | $t_{new}$ | $t_{opt}$ | $t_{old}$ | $c$ |
|---|---|---|---|---|
| $Q_1^{II}$ | 0.07 | 0.13 | 0.09 | **1.3** |
| $Q_2^{II}$ | 0.20 | 0.84 | 0.21 | **1.1** |
| $Q_3^{II}$ | 0.29 | 0.6 | 0.77 | **2.7** |
| $Q_4^{II}$ | 0.17 | 0.07 | 3.65 | **21.5** |
| $Q_5^{II}$ | 0.68 | 2.88 | 73.16 | **107.6** |
| $Q_6^{II}$ | 0.24 | 0.31 | 3.99 | **16.6** |
| $Q_7^{II}$ | 0.23 | 0.31 | 2.04 | **8.9** |
| $Q_8^{II}$ | 0.68 | 0.09 | 28.82 | **42.4** |
| $Q_9^{II}$ | 0.73 | 0.1 | 28.38 | **38.9** |
| $Q_{10}^{II}$ | 0.69 | 0.41 | 14.15 | **20.5** |
| $Q_{11}^{II}$ | 0.73 | 0.41 | 14.34 | **19.6** |

Table 2: Optimization results II

## References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] P.A. Bernstein, B. Harry, P. Sanders, D. Shutt, and J. Zander. The Micosoft repository. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 3–12, 1997.

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

[4] W.-S. Cho, C.-M. Park, K.-Y. Whang, and S.-H. Son. A new method for estimating the number of objects satisfying an object-oriented query involving partial participation of classes. *Information Systems*, 21(3), 1996.

[5] M. A. Derr, S. Morishita, and G. Phipps. The Glue-Nail Deductive Database system: Design, implementation, and evaluation. *VLDB Journal*, 3(2), 1994.

[6] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A prototype for F-Logic. In *Proceedings of the Thirteenth International Conference on Data Engineering*, April 1997.

[7] P. Gassner, K.B. Schiefer, G.M. Lohmann, and Y. Wang. Query optimization in the IBM DB2 family. *Bulletin of the Technical Comittee on Data Engineering*, 16(4):4–18, 1993.

[8] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–19, June 1995.

[9] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford data warehousing project. *Data Engineering, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

[10] IEEE Computer Society. Bulletin of the Technical Comittee on Data Engineering, Special Issue on Query Processing in Commercial Database Systems, 1993.

[11] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, 1991.

[12] Y.E. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems*, 18(4), 1993.

[13] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems, Special Issue on Deductive and Object-Oriented Databases*, 4(2):167–192, March 1995.

[14] M. Jarke and T. Rose. Managing knowledge about information system evolution. In *ACM SIGMOD Conference on Management of Data*, pages 303–311, 1988.

[15] M.A. Jeusfeld. *Update Control in Deductive Object Bases.* PhD thesis, University of Passau (in German), 1992.

[16] J.Vaghani, K. Ramamohanarao, D.B. Kemp, Z. Somogyi, P.J. Stuckey, T.S. Leask, and J. Harland. The Aditi deductive database system. *VLDB Journal*, 3(2):245–288, 1994.

[17] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. ACM SIGMOD Conference*, pages 364–374, 1990.

[18] A. Kemper and G. Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 1992.

[19] W. Kiessling, H Schmidt, W Strauss, and G. Dünzinger. DECLARE and SDS: Early efforts to commercialize deductive database technology. *VLDB Journal*, 3(2), 1994.

[20] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 393–402, 1992.

[21] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and scheme. *ACM-SIGMOD Record*, 18(2):322–341, June 1989.

[22] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *International Conference on Very Large Data Bases*, pages 128–137, 1986.

[23] W. Litwin and T. Risch. Main memory oriented optimization of OO queries using typed Datalog with foreign predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):517–528, 1992.

[24] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[25] H.W. Nissen, M.A. Jeusfeld, M. Jarke, G.V. Zemanek, and H. Huber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, pages 37–47, March 1996.

[26] P. Pucheral and J.-M. Thevenin. Pipelined query processing in the DBGraph storage model. In *Int. Conf. on Extending Database Technology*, pages 516–533, 1992.

[27] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 1979.

[28] D.E. Smith and M.R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26:171 – 215, 1985.

[29] R. Soiron. Cost-based query optimization in deductive databases (in german). Master's thesis, Lehrstuhl Informatik V, RWTH Aachen, 1996.

[30] D. Srivastava and R. Ramakrishnan. Coral++: Adding object-orientation to a logic database language. In *VLDB 93*, pages 158–170, 1993.

[31] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proc. of the 22nd Intl. Conf. on Very Large Data Bases (VLDB'96)*, pages 75–86, Bombay, India, September 1996.

[32] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.

[33] A.N. Swami and K.B. Schiefer. On the estimation of join result sizes. In *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology*, pages 287 – 300, 1994.

[34] J.D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 2. Computer Science Press, 1989.

[35] J. Widom. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM)*, 1995.

[36] C. Zaniolo. Efficient processing of declarative rule-based languages for databases. In *Processing Declarative Knowledge, LNCS 567*, 1991.

[37] G.V. Zemanek. *Methodenhandbuch zur Modellierung der PFR-Methode in ConceptBase.* USU GmbH, Möglingen, 1995.