(Published in IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6, December 1992)

Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates

Witold Litwin[*]   Tore Risch[†]

Abstract

Object-oriented DBMSs (OODBs) have created a demand for relationally complete, extensible, and declarative object-oriented (OO) query languages. Until now, run time performance of such languages was far behind that of procedural OO interfaces. One reason is the internal use of a relational engine with magnetic disk resident databases. We address the processing of the declarative OO language WS-OSQL, provided by the fully operational prototype OODB called WS-IRIS. A WS-IRIS database is MM resident. The system architecture, data structures, and optimization techniques are designed accordingly. WS-OSQL queries are compiled into an OO extension of Datalog called ObjectLog, providing for objects, typing, overloading, and foreign predicates for extensibility. We present cost based optimizations in WS-IRIS using ObjectLog. Performance tests show that WS-IRIS is about as fast as current OODBs with procedural interfaces only and is much faster than known relationally complete systems. These results would not be possible for a traditional disk based implementation. However, MM residency of a database appears only a necessary condition for better performance. An efficient optimization proves of crucial importance as well.

Key Words: Main Memory Database, Object-Oriented Database System, Object-Oriented Query Language, Query Optimization, Foreign Predicates.

[*]Dauphine University, Paris, France, `litwin@eclipse.stanford.edu`.

[†]Dept. of Computer Science, Linköping University, 581 83 Linköping, Sweden, `torri@ida.liu.se`.

1 INTRODUCTION

OO database systems (OODBs) have grown in popularity [13, 14, 27]. Most existing OODBs have navigational languages for object manipulation. A new trend has been to provide them with declarative query languages [3, 4, 10, 12, 20, 31], possibly extensible through foreign methods or functions [7, 30]. Implementation of query languages has led to the problem of query optimization. Little is yet published about this subject for OODBs, but two approaches can be identified:

1. The object model is internally processed by a relational engine. Queries are optimized accordingly [10, 30, 37] using relational algebra.
2. A dedicated object-oriented algebra is defined [9, 29, 31]. Queries are optimized using corresponding transformation rules.

We do not know any implemented system using the second approach. The current proposals are limited to optimization using algebraic operators, and cite the development of fully operational optimizers as a future goal [31]. The first approach seems at present the leading one for fully implemented systems [10, 30, 37], since relational optimization techniques are well understood. However, it is easy to see that this approach is limited. While one rationale for OODBs is much higher efficiency than RDBs, the first approach is basically limited by relational storage performance, although OO-dedicated index structures and statistics can help overcome this limitation [12]. Another problem is that the semantics of the relational model is more limited than that required for an OO query language [31].

Furthermore, relational optimization techniques were developed for classical (magnetic) disk-based very large shared databases, where only a small part of the database could fit the fast main memory (MM). They are partly obsoleted by the progress in hardware and by new working environments, especially the powerful interoperable workstations with individual (personal) databases, and main memories comparable to those of mainframes a few years ago[1]. A PC can now handle MM storage of 32-64 Mb at a cost of $36 per Mbyte, while servers can have GBytes [18]. As OO databases on workstations are usually not larger than that, it is now possible to fit all or most of an entire database in MM[2]. Also, MM has become reliable enough to support data for extended periods of time (e.g. weeks or months) without any crashes.

[1]Large RAM disks have become available as compatible substitutes for magnetic disk, but we consider such devices as MM.

[2]Future databases perhaps will be larger, but main memory will be larger as well, at a much lower price per Mb.

Hence, the interaction with the disk for reliability purpose can be much more infrequent, i.e. individual updates need not be committed to the disk. All this progress requires a revision of traditional database designs, but opens the way to performance that no classical implementation could achieve [18].

One can also observe that logical languages such as Datalog [34] are attractive candidates for the efficient processing of declarative OO queries. Their expressive power often exceeds that of the relational languages, primarily through their recursion capabilities [6, 23, 34], and their optimization principles are fairly well understood [6, 34]. In particular, the relational algebra of the first approach can be generated from a logical language for non-recursive queries [34] and subsequently optimized [15, 28]. However, Datalog lacks update semantics, and OO features such as typing and OID management which makes it impractical to use pure Datalog to process OO queries.

We have built a prototype OODB termed WS-IRIS (Workstation IRIS) that is based on these considerations. WS-IRIS' query language, called WS-OSQL, is an extension of the OO declarative query language of IRIS, termed OSQL. After optimization of type checking, the WS-OSQL queries are translated to an OO generalization of Datalog, termed ObjectLog. The ObjectLog queries are optimized through cost-based rule rewriting and through reordering for safety, which are well known techniques from logic query languages [6, 34]. For processing, the database is entirely in MM, and largely uses MM-oriented data structures such as arrays, hash tables, and linked lists. They prove more efficient than relational storage for OO operations, e.g., for type checking and method selection. WS-IRIS also provides concurrency control, logging, commit, rollback, and recovery facilities. The granularity of the concurrency control is at present the entire database, which is often sufficient for a personal DBMS. For recovery, there is a backup copy on the disk, and a background process saves the database on the disk using the well known copy-on-write Unix facility [3].

Unlike other OO systems we know of, WS-IRIS optimizes multi-way foreign function calls, i.e. where only the result of the function is known and the system finds the corresponding argument(s). This is done using *foreign predicates*, generalizing the concept of external predicates in $\mathcal{LDL}$ [6]. A multi-way foreign function can be transparently mapped to a set of foreign predicates sharing the function name but differently implemented in C[4]. For example, a function accessing an array can be mapped into two foreign predicates, one for directly accessing an array element when its index is known, and another resolving a call with unknown index value through scanning the

[3]In case of a system crash the database can be reloaded from the disk with the log rolled forward.

[4]Alt. LISP or IPL[2].

array. The resolvent choice is in general based on cost estimate from cost functions attached to foreign predicates. The overall benefit for the user is enhanced polymorphism of foreign function calls.

Foreign predicates further allow for inferencing through constraints [17]. For instance, the WS-OSQL user can define a function converting Fahrenheit to Celsius and the system will infer the reverse conversion from Celsius to Fahrenheit.

In what follows we present the WS-OSQL query processing and optimization. Section 2 overviews the languages of WS-IRIS. Section 3 shows the steps for query transformation. Section 4 describes the optimizations within each step. Section 5 discusses foreign predicates. Section 6 presents performance measurements.

The measures show that query evaluation according to the proposed principles can be several orders of magnitude faster than naive evaluation despite the MM processing speed. For instance, for a database of 10,000 objects, the improvement can be from 13 minutes to 1.1 msecs. Such a fast evaluation would not be possible for a traditional disk based implementation, being faster than even a single disk access. On the other hand, the results prove that without effective optimization the system would be too slow for many applications, despite its MM implementation.

We show further that all together the speed of WS-IRIS is comparable to that of OODBs currently providing only navigational access [13, 21], even for large workstation databases[5]. WS-IRIS allows us therefore, to use a relationally complete declarative language for typical OO applications where high efficiency is the primary concern. This is a major advantage of WS-IRIS, since to write a declarative query is usually much faster than to write a navigational program[6] [32].

2 THE WS-OSQL LANGUAGE

2.1 The OSQL Language

WS-OSQL is a dialect of OSQL that is the query language for IRIS systems [10]. OSQL uses the concepts of *types*, *objects*, and *functions*. Objects are represented through typed atomic object identifiers (OIDs), and functions associate properties to objects or define relationships between objects.

[5]80,000 objects.

[6]WS-IRIS also provides a navigational C interface called the *fast path* interface, sometimes also called *call-level* application program interface, e.g. in SYBASE.

Functions model object attributes and relationships between objects through three basic *function types*:

1. *Stored functions* that are tables.
2. *Derived functions* that are defined through OSQL `select` statements.
3. *Foreign functions* that are defined using an external programming language.

Fig. 1 shows some of the OSQL functions and types used in this paper. Objects of type `Person` have the properties `Income, Bonus, Parent, GrossIncome`, and
`GrandSParentGrossIncome`. These are OSQL functions of a single argument bound to objects of type `Person`. Local variables are declared using the "`for each`" clause.

The `GrandSParentGrossIncome` function will be our running example. It illustrates important OO needs, such as efficient traversal of object hierarchies, static and dynamic type checking, inheritance, and overloading (the `Income` function)[7].

The function `Plus` in the derived function `GrossIncome` is a *foreign* function, implemented in C outside WS-OSQL. `GrossIncome` itself, as any derived function, is defined through the `select` statement that follow the "`as`" keyword. The stored functions `Parent`, and `GrandSparentGrossIncome` return sets of values.

In general, the arguments and results of a function together with their types are called the *signature* of the function. We denote signatures by

$\mathtt{f(T}_1\ \mathtt{P}_1\mathtt{,\ldots,T}_n\ \mathtt{P}_n\mathtt{)\ \text{->}\ <U}_1\ \mathtt{Q}_1\mathtt{,\ldots,U}_m\ \mathtt{Q}_m\mathtt{>}$[8]

$\mathtt{P}_1,\ldots,\mathtt{P}_n$ are the *arguments* of `f` whose values are of types $\mathtt{T}_1,\ldots,\mathtt{T}_n$. A function can have as the result a set of tuples of values, $\mathtt{Q}_1,\ldots,\mathtt{Q}_m$, with types $\mathtt{U}_1,\ldots,\mathtt{U}_m$.

OSQL has a SQL-like `select` statement both for ad hoc queries to the database and for defining derived functions.

Overloaded functions are OSQL functions sharing a name for different definitions. In Fig. 1 there are two variants, or *resolvents*, of the overloaded `Income` function. Both resolvents provide a person's income, the first one is given an OID of an object of type Person, while the other is given a name as a string. Resolvents can be any of the three basic function types. WS-IRIS

[7]It also illustrates an increasingly popular need of modern grandparents.

[8]The brackets around the result are optional for functions returning a single result.

```
create function Name(Person p) -> Charstring nm as stored;
create function Income(Person p) -> Integer i as stored;
create function Bonus(Person p) -> Integer i as stored;
create function Parent(Person c) -> Person p as stored;
create function GrossIncome(Person p) -> Integer gi as
    select  Plus(Income(p),Bonus(p));
                   /* Derived where plus is foreign */
create function SParent(Person c) -> Student s as
    select s                     /* Derived */
    where Parent(c) = s;         /* Parent if parent is student */
create function GrandSParentGrossIncome(Person c) -> Integer gi as
    select gi                    /* Gross income of grandparent
                                    if grandparent is student */
    for each Person gp, Person p
    where GrossIncome(gp) = gi and
          SParent(p) = gp and
          Parent(c) = p;
create function Income(Charstring nm) -> Integer i as
    select Income(p)        /* Derived  overloaded */
    for each Person p
    where Name(p) = nm;
```

Figure 1: Examples of OSQL functions

chooses then the resolvent according to the argument type, at compile or at run time.

2.2 New features in WS-OSQL Language

The main new features of WS-OSQL [26], are multi-way foreign functions, a limited form of recursion, late binding of overloaded functions, and 2nd order functions. WS-OSQL furthermore has aggregation operators, nested subqueries, disjunctive queries, and quantifiers, and is relationally complete. The user can also provide cost hints to the optimizer as OSQL functions. We discuss these possibilities more in depth in the sections that follows.

2.3 ObjectLog

WS-OSQL compiles into its intermediate language we termed ObjectLog. ObjectLog is inspired by Datalog and $\mathcal{LDL}$ but provides new facilities for effective processing of OO queries. ObjectLog generalizes the polymorphic type extensions proposed for Prolog [22] by providing a type hierarchy, late binding, update semantics, and foreign predicates. The most important features are:

- Predicate arguments are *objects*, where each object belongs to one or more *types* organized in a type hierarchy that corresponds to the type hierarchy of Iris [10].
- Object creation and deletion semantics maintains the referential integrity of the type hierarchy.
- Update semantics of predicates preserve type integrity of arguments. The optimizer relies on this to avoid dynamic type checking in queries (Sec. 4.1).
- Predicates can be overloaded on the types of their arguments. We call corresponding resolvent a *Type Resolved* (TR) predicate (Sec. 3.3).
- Predicates can be further overloaded on the binding patterns of their arguments, i.e. on which arguments are bound or free when the predicate is called. We call each corresponding resolvent a *Type and Binding Pattern Resolved* (TBR) predicate (Sec. 3.4).
- Predicates can be not only facts and rules, but also multi-way *foreign predicates* implemented in a procedural language[9]. Foreign pred-

[9]C, Lisp, or IPL [2].

icates implement foreign functions, especially multi-way foreign functions (Sec. 5).

- Predicates themselves as well as types are objects, and there are second order predicates that produce or apply other predicates. 2nd order predicates are crucial, e.g., for late binding (Sec. 4.1.1) and recursion (Sec. 5.4).

3 QUERY PROCESSING STEPS

A WS-OSQL query is defined through a `select` statement. WS-IRIS compiles ad hoc queries as if they were unnamed derived functions. Vice versa, functions can be seen as views defined through queries, like in SQL. A query or a function definition is compiled to ObjectLog representation. The compiler transforms a function or a query in several steps, shown top-down in Fig. 2. We now summarize these steps; the next section describes the query processing algorithms more in detail. Examples refer to the compilation of functions only.

3.1 Flattener

As Datalog, ObjectLog does not allow function symbols to appear in arguments. Therefore, the compiler first transform WS-OSQL `select` statements into flattened `select` statements, without functions in the result list and without function nesting in the predicate. We flatten `select` statements by recursively introducing intermediate variables for each nested function call. Fig. 3 shows how the function `GrossIncome` of Fig. 1 is flattened by introducing intermediate variables `gi`, `_v1`, and `_v2`.

The flattener also detects and marks recursive functions, discussed in Sec. 5.4.

3.2 Type Checker

Type checking has three phases:

During the *type adornment phase*, the translator identifies *type-adorned (TA)* resolvents of a flattened function by annotating the name of the function with the names of its signature types. An overloaded function has a TA resolvent for each definition. For example, `Income` in Fig. 1 has the TA resolvents: $\texttt{Income}_{Person->Integer}$ and $\texttt{Income}_{Charstring->Integer}$.

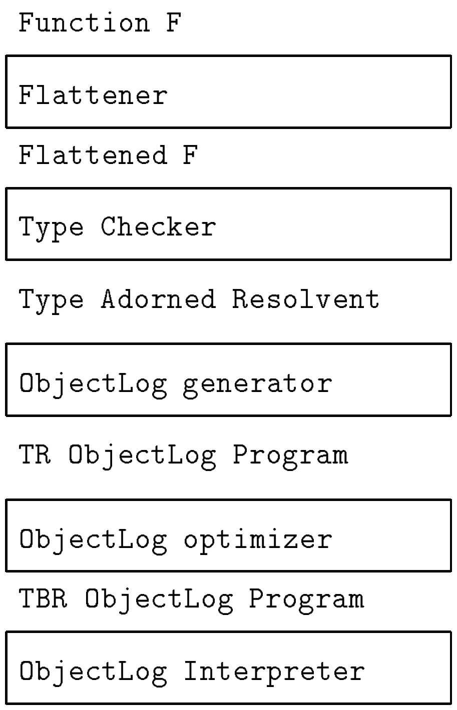

Figure 2: Translation steps

```
create function GrossIncome(Person p) -> Integer gi as
    select gi
    for each Integer _v1, _v2
    where _v1 = Income(p) and
          _v2 = Bonus(p) and
          gi  = Plus(_v1,_v2);
```

Figure 3: Query Flattening

```
create function GrandSParentGrossIncome_{Person->Integer}(c)
                -> Integer gi as
       select gi
       for each Student gp, Person p
       where GrossIncome_{Person->Integer}(gp) = gi and
             SParent_{Person->Student}(p) = gp and
             Parent_{Person->Person}(c) = p;
```

Figure 4: A TA resolvent

Then an *overload resolution* algorithm (Sec. 4.1) substitutes function calls in the flattened `select` statement with their TA resolvents. Fig. 4 shows the result for the function
`GrandSParentGrossIncome`. This algorithm cannot be applied to late-bound function calls, in which case the overload resolution has to be done at run time (Sec 4.1.1).

Finally, the optimizer adds dynamic type checks to the function definition whenever the type of a variable cannot be guaranteed to be of the desired type assuming the referential integrity of updates. This is the case of the function `SParent` (Fig. 1), where the system must dynamically check that the variable `s` is of type `Student`, since the function `Parent` returns `Person` objects not always being `Student` objects.

3.3 ObjectLog Generator

The ObjectLog generator transforms TA resolvents into Type Resolved (TR) ObjectLog programs:

Stored functions become TR facts. For example the TA resolvent `Income` $_{Person->Integer}$ would generate the TR fact `income`$_{Person,Integer}$`(P,I)`.

Derived functions become TR rules. The signature becomes the head of the rule and the `select` statement becomes the rule body. Fig. 5 shows the TR rule of the derived function `GrandSParentGrossIncome`.

Finally, the foreign functions become TR foreign predicates. We'll discuss this transformation in Sec. 5.

$\texttt{grandsparentgrossincome}_{Person,Integer}\texttt{(C,GI) :-}$
$\qquad\qquad\texttt{grossincome}_{Person,Integer}\texttt{(GP,GI) \&}$
$\qquad\qquad\texttt{sparent}_{Person,Student}\texttt{(P,GP) \&}$
$\qquad\qquad\texttt{parent}_{Person,Person}\texttt{(C,P).}$

Figure 5: TR Rule

3.4 ObjectLog Optimizer

Each WS-OSQL function is compiled into a TBR predicate where bound predicate arguments correspond to the arguments of the function, and the unbound arguments correspond to the results. The function is optimized for execution in the *forward* direction where arguments are known and results computed. The rationale is that functions are normally used as methods that compute properties of the arguments. Functions can nevertheless also be used inversely, in which case the optimizer will generate different TBR predicates.

The TR rules constitute the entry to the ObjectLog optimizer. The outcome are optimized TBR rules. The optimization algorithm use results in [15, 34] for rule reordering, and [6] for foreign predicate optimization. However, these results had to be revisited for ObjectLog, because of the overloading on binding patterns. The consequence is a larger space of safe reorderings to choose the optimal one from. The increase is obtained in two ways:

- The explicit definition of the set of resolvents, $S(P)$, overloading a TR predicate, P, provided by the user as part of a foreign function definition (Sec. 5.1).
- The completion algorithm (Sec. 5.3) that can infer from $S(P)$ yet other resolvents.

The search space is then explored using greedy heuristics to find the cheapest reordering according to the cost model in [15]. Two benefits result from the ObjectLog approach with respect to the algorithm in [6]:

- A more efficient optimized program can be chosen.
- There can be TBR programs that would have no solution in a smaller search space. This is, e.g., the case of the constraint inferencing (Sec. 5.1), and of examples in [16] (Sec. 8.3), and [33] (pp. 3-6).

$$\begin{array}{ll}
\texttt{grandsparentgrossincome}_{Person,Integer}(\texttt{C,GI})\ \texttt{:-} & \\
 & \texttt{income}_{Person,Integer}(\texttt{GP,\_V2})\ \texttt{\&} \\
 & \texttt{bonus}_{Person,Integer}(\texttt{GP,\_V1})\ \texttt{\&} \\
 & \texttt{plus}_{Integer,Integer,Integer}(\texttt{\_V2,\_V1,GI})\ \texttt{\&} \\
 & \texttt{typesof}_{Object,Type}(\texttt{GP,typeStudent})\ \texttt{\&} \\
 & \texttt{parent}_{Person,Person}(\texttt{P,GP})\ \texttt{\&} \\
 & \texttt{parent}_{Person,Person}(\texttt{C,P}).
\end{array}$$

Figure 6: A Substituted TR Rule

$$\begin{array}{ll}
\texttt{grandsparentgrossincome}^{bf}_{Person,Integer}(\texttt{C,GI})\ \texttt{:-} & \\
 & \texttt{parent}^{bf}_{Person,Person}(\texttt{C,P})\ \texttt{\&} \\
 & \texttt{parent}^{bf}_{Person,Person}(\texttt{P,GP})\ \texttt{\&} \\
 & \texttt{typesof}^{bb}_{Object,Type}(\texttt{GP,typeStudent})\ \texttt{\&} \\
 & \texttt{income}^{bf}_{Person,Integer}(\texttt{GP,\_V2})\ \texttt{\&} \\
 & \texttt{bonus}^{bf}_{Person,Integer}(\texttt{GP,\_V1})\ \texttt{\&} \\
 & \texttt{plus}^{bbf}_{Integer,Integer,Integer}(\texttt{\_V2,\_V1,GI}).
\end{array}$$

Figure 7: An optimized TBR rule

Fig. 7 shows the optimal TBR program for the function `GrandSParentGrossIncome` that will be justified later. Each literal refers to TBR predicate names, where superscripts '`b`' stands for *bound argument* and '`f`' stands for *free argument.*

3.5 ObjectLog Interpreter

The ObjectLog interpreter executes the TBR program with `b` marked arguments bound to produce the corresponding result tuples. For instance, $\texttt{grandsparentgrossincome}^{bf}_{Person,Integer}$ would be invoked if a query about `GrandSParentGrossIncome` for a given `Person` was submitted. The interpreter uses a top down interpretation method that corresponds to the nested-loop method in relational databases [15][10].

4 OPTIMIZATION ALGORITHMS

4.1 Type Checking

It is advantageous for efficiency to perform overload resolution at compile time whenever possible. In particular, it is advantageous for choosing the best TBR program, because late binding makes rule substitution and thus global optimization impossible. Thus late binding should be used only when semantically necessary.

For dynamic type checking there is a built-in foreign function, `TypesOf`, that returns the set of types to which a given object belongs:

```
   TypesOf(Object o) -> Type t
```

The simplest processing strategy is to add `TypesOf` to each variable declared in a function body (Fig. 8)[11]. This strategy, however, sometimes introduce unnecessary checks. The type checker avoids such checks through the following rule:

[Type Check Removal:] Consider a TA function `f`, invoked in a `select` statement:

$\texttt{f}_{T_1,\ldots,T_m->T_{m+1},\ldots,T_{m+n}}(\texttt{A}_1,\ldots,\texttt{A}_m) = \texttt{<}\texttt{A}_{m+1},\ldots,\texttt{A}_{m+n}\texttt{>}$

[10]By contrast, NAIL! [34] uses a bottom-up method for efficient handling of a class of recursive queries.

[11]The variables `typeInteger`, `typeStudent`, and `typePerson` refer to objects representing the types named `'Integer'`, `'Student'`, and `'Person'`, respectively.

```
create function GrandSParentGrossIncome$_{Person->Integer}$(c)
                -> Integer gi as
       select gi
       for each Person c, Student gp, Person p
       where TypesOf$_{Object->Type}$(c)=typePerson and
             TypesOf$_{Object->Type}$(gi)=typeInteger and
             TypesOf$_{Object->Type}$(gp)=typeStudent and
             TypesOf$_{Object->Type}$(p)=typePerson and
             GrossIncome$_{Person->Integer}$(gp) = gi and
             SParent$_{Person->Student}$(p) = gp and
             Parent$_{Person->Person}$(c) = p;
```

Figure 8: Function Definition with Type Checks .

Variables, A_j are declared of type D_j. If T_j is D_j or a subtype of D_j, then one may remove the type check for A_j

The type check removal rule is valid since types of the arguments and results of `f` are constrained by the referential integrity system for stored functions and by type checking for derived functions.

It turns out that all `TypesOf` calls in Fig. 8 prove unnecessary:
The type check `TypesOf`$_{Object->Type}$`(c)=typePerson` is unnecessary since the argument of `Parent`$_{Person->Person}$ must be of type `Person`. Analogously, `TypesOf`$_{Object->Type}$`(gi)=typeInteger` is unnecessary since `GrossIncome`$_{Person->Integer}$ returns integers, etc.

By contrast, in the definition of `SParent`, a type check is needed, checking that the variable `s` is of type `Student`, since `Parent`$_{Person->Person}$ returns a `Person` which is a more general type than `Student`.

4.1.1 Late Binding

It is advantageous for efficiency to perform overload resolution at compile time whenever possible. In particular, it is advantageous for choosing the best TBR program, as late binding makes some optimizations impossible. Thus late binding should be used only when semantically necessary. In WS-OSQL a special keyword, `late`, indicates to the type checker when late binding is to be used in an OSQL function call.

The query processor generates a 'generic' TBR predicate on the universal type `Object` for every overloaded OSQL function. The generic predicate does dynamic overload resolution to handle late binding. For example, the OSQL function `Income` in Fig. 1 will generate a generic predicate, $\texttt{Income}^{bf}_{Object,Integer}$. The generic TBR predicate is implemented as a call to a second order foreign system predicate, `apply`. This predicate first does the type resolution at run time based on the types of the actual argument values, and calls the selected TBR predicate.

4.2 ObjectLog Optimizer

4.2.1 Rule Substitution

A TR rule can refer to other TR rules. For example, the TR rule $\texttt{GrandSParentGrossIncome}_{Person,Integer}$ (Fig. 5) refers to the TR rules $\texttt{SParent}_{Person,Student}$, and $\texttt{GrossIncome}_{Person,Integer}$. The rule substitution phase combines such rules into one larger rule, whenever possible, e.g. there is no recursion (see [34] vol. II Sec. 13.4). The reason is that the global optimization of a set of substituted rules often proves more efficient than to local optimization of individual rules. Fig. 6 shows the rule substituted TR program of $\texttt{GrandSParentGrossIncome}_{Person,Integer}$.

4.2.2 Cost Model for Rule Reordering

The Basis

Choice of bindings through rule reordering (e.g., join ordering, selection pushing) and of access methods (e.g., join method, index creation/use) are among the most important techniques for optimizing Datalog queries [34]; as well as for relational queries in general [28].

Traditional optimizers have exponential optimization time over the number of literals in a rule (joins) [28]. This proves inconvenient for WS-OSQL, and WS-IRIS default is a heuristic optimization [15] which produces query plans in quadratic time. The rest of this section presents the principles of the heuristics. They are MM oriented, as all ObjectLog data structures are in MM.

Our optimization method is a variant of the nested-loop join method, generalized for foreign predicates. No disk access costs are considered, since the entire database is in MM and eventual disk backups are done asynchronously in the background. By the same token, cluster orderings are not considered which simplifies rule reordering. Also, the MM residency costs of common primitive operations are comparable to data access costs, unlike for disk resident databases. These operations are arithmetic operators, foreign function calls, etc.

Optimization Heuristics

Let P be a TBR rule or fact. We call the *input tuple* the tuple corresponding to variable(s) that are bound in P. For a given input tuple there are zero, one, or several *output tuples*, corresponding to unbound variable(s) in P. For each TBR predicate P two cost estimates are calculated:

1. The *execution cost* of P, C_P, defined as the number of visited tuples, given that all variables of the input tuple are bound.

2. The *fanout*, F_P, which is the estimated number of output tuples produced by P for a given input tuple.

The optimizer minimizes the total cost, C, to join a conjunction of literals, $\{P_i\}_1^n$:

$$C = \sum_{i=1}^{n}(C_{P_i} \prod_{j=1}^{i-1} F_{P_j})$$

As in [15] the optimizer computes a rank, R_{P_i}, for each P_i in $\{P_i\}_1^n$:

$$R_{P_i} = \frac{F_{P_i} - 1}{C_{P_i}}$$

Then next literal to evaluate is the literal minimizing R_{P_i}, provided that it is executable (safe) at the position to be taken in the rule. The calculus is iterated until there are no literals left. The motivation for this heuristics is that to repeatedly minimize R_{P_i} also minimizes C.

Default Values for Cost Parameters.

In a populated database the system estimates C_P and F_P from the cardinality of stored predicates, join selectivity, and index kind and availability[12]. To get reasonable optimization even before the database is populated, the system uses the default cost parameters below. After the database has been populated the user can instruct the system to optimze all OSQL functions using the statistics of the populated database.

The optimizer distinguishes between joining on unique indexes, non-unique indexes, and unindexed input tuples. On experimental basis, defaults are currently as follows:

- F_P=1 if the input tuple has a unique index.

- F_P=2 if it has a non-unique index.

[12]OSQL allows the DBA to put indexes on any argument or result of a function, and the system by default puts an index on the first argument of OSQL functions.

- F_P=4 otherwise.

The rationale for these defaults is that input tuples with unique indexes will have a maximal possible F_P value of 1. A non-unique index is usually efficient only for a fanout slightly larger than one. Unindexed input tuples have usually fanout larger than indexed ones; that is why they remain unindexed.

The default size of a stored predicate is assumed 100 tuples. The corresponding defaults for C_P are:

- C_P=F_P if the input tuple has an index.
- C_P=100 if it is unindexed, since the system has to scan the entire table.

Foreign predicates have default F_P=1 and C_P=1, assuming that they are cheap to execute and return a single result tuple.

Cost Hints

The DBA can provide *cost hints* for each TBR predicate, which override default assumptions about C_P and F_P. Hints are particularly useful for evaluating foreign predicates. These hints are provided by the DBA as a WS-OSQL function that for a given TBR predicate returns the two estimates C_P and F_P.

For example, the system TBR predicate $\texttt{typesof}^{bf}_{Object,Type}\texttt{(X,T)}$ computes the type of a given object `X`, while $\texttt{typesof}^{fb}_{Object,Type}\texttt{(X,T)}$ computes the objects belonging to type `T`. Computing the type of a WS-IRIS object is very cheap, since a pointer to the types of each object is stored directly in the OID. However, finding all the objects of a given type is expensive and proportional to the number objects of the type. The cost hint functions for $\texttt{typesof}^{bf}_{Object,Type}\texttt{(X,T)}$ and $\texttt{typesof}^{fb}_{Object,Type}\texttt{(X,T)}$ specify those hints.

4.2.3 Example

The ObjectLog interpreter would interpret the unoptimized program of Fig. 6 using the nested loop algorithm of Fig. 9.

Our test database has a database populated with 10,000 persons of which 2,500 are students, and the stored predicates `parent,` `income`, and `bonus` have hash-based indexes on their first argument. Since we use hash based MM indexing we can assume a constant cost of, e.g., 2 to access an index, while a basic operation (e.g. arithmetic) has cost 1.

The execution plan of Fig. 9 is very inefficient, because it first iterates on line `2` over the entire extension of the stored predicate `income` (10,000 iterations). Furthermore, on line `6` it also iterates over the extension of `parent`,

```
grandsparentgrossincome(C) -> GI:
forall GP,_V2 when income(GP,_V2) do
  forall _V1 when bonus(GP,_V1) do
     GI = _V1 + _V2
      when typesof(GP,typeStudent) do
        forall P in parent(P,GP) do
           when parent(C,P) do
                emit(GI);
```

Figure 9: Unoptimized interpretation of `grandsparentgrossincome`

since `parent` has no index on its second argument. Assuming that all other extensional predicates have indexes and that type checking is very fast, the cost of the above execution plan will be $O(card(\texttt{income}) \cdot card(\texttt{parent}))$, i.e. $O(10^8)$.

Fig. 7 shows the final definition of `GrandSParentGrossIncome` after applying our cost heuristics. Appendix A shows the details of the calculus. The plan has a constant execution cost of 10, since indexes are used on `income`, `bonus`, and `parent`, and `plus` and `typesof` are basic operations.

In Section 6 we show empirical results on `grandsparentgrossincome` that verifies the importance of this optimization.

5 FOREIGN PREDICATES

5.1 The Rationale

Many system functions in WS-IRIS are implemented as foreign functions, e.g. for late binding (Sec. 4.1.1), quantification, and aggregate functions [26]. Some of them have to be multi-way, e.g. `TypesOf` (Sec. 4.1), and foreign predicates reveal a very efficient way for their implementation. Foreign predicates also open new possibilities for user defined foreign functions. For instance, they allow for constraint inferencing (compilation) [17] as Fig. 12 illustrates.

Foreign functions can have several TBR foreign predicate resolvents for different binding patterns[13]. Each TBR predicate has associated cost functions (either default or user defined). The whole concept of TBR foreign predicates generalizes that of external predicates in $\mathcal{LDL}$ [6], that were neither

[13]The resolvents are normally implemented in C, Lisp, or IPL.

```
sparent$_{Person,Student}$(C,S) :-
        typesof$^{fb}_{Object,Type}$(S,typeStudent) &
        parent$^{bf}_{Person,Person}$(C,S).
```

Figure 10: Unoptimized TBR ObjectLog program for `SParent`

```
sparent$_{Person,Student}$(C,S) :-
        parent$^{bf}_{Person,Person}$(C,S) &
        typesof$^{bb}_{Object,Type}$(S,typeStudent).
```

Figure 11: Optimal TBR ObjectLog program for `SParent`

polymorphic over binding patterns nor over types.

As an example of the importance of foreign predicate optimization, regard the function `SParent` of Fig. 1 whose unoptimized TBR representation is shown in Fig. 10. The type checker has added a call to the built in foreign function `typesof` to test if the parent is a student. The above TA definition is, however, inefficient, because `S` is not known when `typesof`$^{fb}_{Object,Type}$ is called. Thus `typesof`$^{fb}_{Object,Type}$ iterates through and emits all objects of type `Student`[14]. The execution time will be proportional to the number of students in the database.

By contrast, the optimal TBR program is shown in Fig. 11. Its execution time is constant, since `parent` has a hash-based index on `C` and the foreign predicate `typesof`$^{bb}_{Object,Type}$ is implemented as a simple check of the type tag of the object `S`.

The foreign predicates allow to implement elegantly the constraint compilation of [17], as Fig. 12 illustrates. The function `ftoc` in Fig. 12 converts Fahrenheit degrees into Celsius. Assuming the Celsius temperature is stored using `ctemp`, the constraint compilation allows to use `ftoc` as a constraint to inversely infer `ftemp` given `ctemp`. Otherwise a different function, let it be `ctof`, would be needed, and the user would manually choose between both, depending on the query. Similarly, the functions `Minus`, and `Div` are defined as derived functions in terms of two inverted foreign functions, `Plus` and `Times`. Without the constraint compilation `Minus` and `Div` would have to be implemented as separate foreign functions, where the user would have to

[14]Foreign predicates, such as `typesof`$^{fb}_{Object,Type}$ are defined as *generators* that call a system provided C function, `emit`, for each result of a set valued foreign predicate. The generator method avoids materializing the result set, which saves MM usage significantly.

```
create function ftoc(Real f) -> Real c as
  select Div(Times(Minus(f,32.),5.),9.);
create function ctemp(Person p)-> Real c as stored;
create function ftemp(Person p)-> Real f as
  select f
  where ftoc(f) = ctemp(p);
create function Minus(Real x, Real y) -> Real r as
  select r
  where Plus(y,r) = x;
create function Div(Real x, Real y) -> Real r as
  select r
  where Times(y,r) = x;
```

Figure 12: Example of Multiway use of Foreign Functions

$$\begin{array}{lll} \texttt{ftemp}^{bf}_{Person,Real}\texttt{(P,F)} & \texttt{:-} & \texttt{plus}^{bff}_{Real,Real,Real}\texttt{(32,\_V3,F) \&} \\ & & \texttt{times}^{bbf}_{Real,Real,Real}\texttt{(\_V3,5,\_V2) \&} \\ & & \texttt{times}^{bfb}_{Real,Real,Real}\texttt{(9,\_V1,\_V2) \&} \\ & & \texttt{ctemp}^{bb}_{Person,Real}\texttt{(P,\_V1).} \end{array}$$

Figure 13: An unexecutable TBR rule

choose the correct implementation depending on the query, and the system would not have been able to infer the inverse of `ftoc`.

5.2 Reordering for Safety

While facts allow any binding patterns, foreign predicates do not. A TBR rule that refers to undefined TBR predicates is not executable at all, i.e. it is *unsafe* [6]. For example, Fig. 13 shows the rule substituted TBR program of `ftemp`. It cannot be executed without rule reordering, since the foreign TBR predicate $\texttt{plus}^{bff}_{Real,Real,Real}$ is undefined. The body of `ftemp` has to be reordered for safety, where every TBR predicate is defined or can be inferred by the *completion algorithm* to be discussed in Sec. 5.3.

For every TR predicate, P, there will be a corresponding set of defined TBR predicates, $S(P)$, realizing the overloading on P. For the TR predicates `plus` and `times` we have the following implementations[15]:

[15]We omit the type adornments for simplicity.

$\texttt{ftemp}^{bf}_{Person,Real}\texttt{(P,F) :- ctemp}^{bf}_{Person,Real}\texttt{(P,\_V1) \&}$
$\qquad\qquad\qquad\qquad\quad\texttt{times}^{bbf}_{Real,Real,Real}\texttt{(9,\_V1,\_V2) \&}$
$\qquad\qquad\qquad\qquad\quad\texttt{times}^{fbb}_{Real,Real,Real}\texttt{(\_V3,5,\_V2) \&}$
$\qquad\qquad\qquad\qquad\quad\texttt{plus}^{bbf}_{Real,Real,Real}\texttt{(32,\_V3,F).}$

Figure 14: Optimized Constraints

$S(\texttt{plus}) = \{\texttt{plus}^{bbf}, \texttt{plus}^{bfb}, \texttt{plus}^{fbb}\}$
$S(\texttt{times}) = \{\texttt{times}^{bbf}, \texttt{times}^{bfb}, \texttt{times}^{fbb}\}$

Assume that in the process of reordering a rule for the position j in the reordered rule, the literal P_j chosen gets a binding pattern X such that $P_j^X \notin S(P)$, (i.e. the TBR predicate P_j^X is undefined) and P_j^X cannot be inferred from S using the *completion algorithm* to be defined in next section. P_j is then disregarded for this position and another literal is examined by the heuristic or the exhaustive optimization algorithm, whichever is used.

In our example, the optimal TBR program for $\texttt{ftemp}_{Person}$ is at Fig. 14. It results from the following steps:

1. Initially only `P` is bound, $\texttt{ctemp}^{bf}_{Person,Real}\texttt{(P,\_V1)}$ is chosen as the implemented TBR predicate in the rule body.
2. Once `ctemp` is called both `P` and `_V1` are bound, and $\texttt{times}^{bbf}_{Real,Real,Real}\texttt{(9,\_V1,\_V2)}$ is the only implemented TBR predicate.
3. Then `P`, `_V1`, and `_V2` are known which makes $\texttt{times}^{fbb}_{Real,Real,Real}\texttt{(\_V3,5,\_V2)}$ the (only) implemented TBR predicate.
4. Finally $\texttt{plus}^{bbf}_{Real,Real,Real}\texttt{(32,\_V3,F)}$ is chosen.

Note that this example would have been unsafe with the optimization principles of [16], because of smaller search space.

5.3 Completion Algorithm

Not every possible TBR foreign predicate need to be implemented for a given TR foreign predicate P. Some TBR predicates can be automatically inferred from $S(P)$ through the *completion algorithm*. The idea is to avoid to implement TBR foreign predicates, that are *covered* by elements of $S(P)$.

Some programming effort can then be saved. Informally speaking, a covering element has fewer bindings, e.g. `plus`^{bbf} covers `plus`^{bbb}.

Formally, a TBR predicate, $\mathtt{P}^{p_1,\ldots,p_n}$, *covers* another TBR predicate, $\mathtt{P}^{q_1,\ldots,q_n}$, when $\forall i : p_i = q_i$ or $q_i =^b$. The following example illustrates the problem.

Consider the OSQL query testing whether two numbers add up to a given sum:

```
select where Plus(1,2) = 3;
```

This query leads to the TBR query[16] `plus`^{bbb}`(1,2,3)?`. It is, however, not necessary to implement `plus`^{bbb}, i.e. `plus`^{bbb} $\notin S($`plus`$)$. The system can instead use `plus`^{bbf}`(1,2,_V1)` to compute `_V1` and then test whether `_V1=3`. The ObjectLog query would be:
`plus`^{bbf}`(1,2,_V1) & EQ(_V1,3)?`

Any covered TBR predicate can be substituted with its cover. The general algorithm is as follows:
Consider $P^y \in S(P)$, and $P^x \notin S(P)$ covered by P^y. The optimizer will replace each

$P^X(a_1, ..., a_n)$ (1)

with the expression:

$P^Y(b_1, ..., b_n) \& eq(V_{i_1}, a_{i_1}) \& ... eq(V_{i_m}, a_{i_m})$ (2)

where:

(i) $\mathtt{b}_i$ `=` $\mathtt{a}_i$ if both are bound.
(ii) $\mathtt{b}_{i_k}$ `=` $\mathtt{V}_{i_k}$ if $\mathtt{a}_{i_k}, k = 1, .., m$ is bound in P^X and $\mathtt{b}_{i_k}$ is free in P^Y.
The case when a_i is free but b_i is bound is impossible, since P^Y covers P^X.

An interesting problem is that of a minimal $S(P)$. While the substituted covering form (2) is semantically equivalent to the implementation of P^y, it is less efficient. The difference is often negligible, e.g. for `plus`^{bbf} and `plus`^{bbb}, and we need not implement the covered predicate. However, the inverse may be true as well, and the covered predicate can be significantly faster than its cover. It is then advantageous to implement it anyhow.

For example, consider the TR predicate $\mathtt{typesof}_{Object,Type}$. It would be sufficient. to implement the TBR predicate $\mathtt{typesof}^{ff}_{Object,Type}$[17], which would return for every object in the system all its types. However, for large databases the execution cost of $\mathtt{typesof}^{ff}_{Object,Type}$ is prohibitive, while the cost to execute $\mathtt{typesof}^{fb}_{Object,Type}$ is proportional to the number of objects of the given

[16]We omit type adornments.
[17]Since `ff` covers every other binary binding pattern.

```
create function ancestor(Person p) -> Person a
  as select a
     for each Person par
     where ( (a = ancestor(par) or
              a = par) and
             par = parent(p));
```

Figure 15: A Left Recursive Function

type, and often used $\texttt{typesof}^{bf}_{Object,Type}$ is implemented in a way it is very fast. Therefore, only the two latter foreign TBR resolvents were implemented, with appropriate cost functions attached. On the other hand, $\texttt{typesof}^{bb}_{Object,Type}$ is covered by $\texttt{typesof}^{bf}_{Object,Type}$, but no implementation significantly faster than its cover could be found. Therefore this foreign predicate was not implemented.

5.4 Safety of Recursive Functions

Our safety checking algorithm also supports recursive functions, e,g. in Fig. 15. We recall that the flattener detects recursive functions and then adds a call to a system function, $\texttt{Apply(fn,a}_1\texttt{,..,a}_n\texttt{) = <r}_1\texttt{,..,r}_m\texttt{>}$, that applies an arbitrary OSQL function `fn` on its arguments $\texttt{a}_1\texttt{,..,a}_n$ producing the results $\texttt{<r}_1\texttt{,..,r}_m\texttt{>}$. The function `Apply` is compiled into a system TR predicate, $\texttt{apply(p,a}_1\texttt{,..,a}_n\texttt{,r}_1\texttt{,..,r}_m\texttt{)}$, where `p` is the recursive TBR predicate[18]. `apply` is executable only if the TBR rule it is used in can be reordered so that all a_i are bound.

Since ObjectLog programs are evaluated top-down, the left recursive calls would be unsafe. The consequence of the use of the `apply` predicate is that such calls are transformed into right-recursive calls. That is why the recursive function in Fig. 15 is executable in ObjectLog. It is known that right recursive calls are safe unless there are circularities in the data or the definition. These cases are not detected by WS-IRIS optimizer at present. For this the semantics of ObjectLog should be extended with memoing [35] or a bottom-up approach [34].

[18]$\texttt{ancestor}^{bf}_{Person,Person}$ in the example.

GrandSParentGrossIncome	*Optimization levels*		
Database size	F	NTC	NRR
100 objects	1.1ms	2.5ms	120ms
1,000 objects	1.1ms	2.5ms	7.3s
10,000 objects	1.1ms	2.6ms	13.3min

F = Full optimization, i.e rule reordering and type check removal (Fig. 7).
NTC = Rule reordering, but no type check removal.
NRR = Type check removal, but no rule reordering.

Figure 16: Performance measurement

6 PERFORMANCE MEASUREMENTS

Performance measures concern the efficiency of WS-IRIS and its comparison to other OODBs and relational systems. We evaluated some query execution times with and without optimization. We also benchmarked WS-IRIS according to the OO1 benchmark [5].

6.1 Query Evaluation

To evaluate the efficiency of the optimization algorithm, we have measured the execution time of `GrandSParentGrossIncome` for a given person. This query combines some important OO features: navigation through a hierarchy, type checking, and foreign functions. The database contained 100, 1000, and 10000 `Person` objects. The measurements were run on a SUN4/470 (SPARC). We measured the speed up due to the full optimization, or to partial optimizations. Fig. 16 summarizes our experiences.

The figure shows that rule reordering is by far the most important optimization. As columns F and NRR show, the gain is over 100 times already for 100 objects, and over seven hundred thousand times for 10,000 objects. Note that the time for the optimal case is in practice constant, one reason being the direct use of MM hash tables as internal implementation. The contribution of type check removal is comparatively modest and constant by factor of about 2, as columns F and NTC show.

Fig. 17 shows the performance of full optimization of the recursive `Ancestor` function of Fig. 15 on the same sample data. Six ancestors were returned in each call. It shows excellent performance of about 4 millisecond per query, thus close to that of `GrandSParentGrossIncome`. The time is again constant because of the use of MM hash tables.

It is instructive to compare these figures to performance of typical disk based implementations. The database used in the test would be about one Mbyte large. Assuming a typical page size of 4K, it would span over 250 pages. The testbed query accessed seven persons (the person, its parents and grandparents). Without clustering the performance of the first query would need at least seven accesses to the corresponding data pages plus some accesses to index pages, typically about seven as well. In the best case of perfect matching of the clustering scheme, the query would need two accesses[19]. The usual average disk access time is 15-40 msec. The processing of the query by WS-IRIS is hence about 30-70 times faster than for a disk based system in the best case, and about 200-500 times faster for the usual case of a query not matching the clustering criteria. Similar results hold for recursive queries.

These figures show the critical importance of MM residency for access performance, known to be a critical requirement for OO applications. On the other hand, MM alone is not sufficient for good performance. Without the optimization, our MM implementation could be even slower than a disk based one.

6.2 The OO1 benchmark

The OO1 benchmark focuses on important characteristics of OO applications. It simulates a CAD database with 20,000 parts and 60,000 connections. Fig. 18 shows OO1 benchmark for WS-IRIS and the systems originally benchmarked[20]. The test of WS-IRIS was run on a SUN3/280 with 16MBytes of main memory, that was also used by the original benchmark as server, and has the same speed as the SUN3/260 used in OO1 as client machine. To measure the best performance, the benchmarked systems are called through fast-path interfaces assuming 'warm' database state. Hence the systems are allowed to cash data in MM as much as they can for best execution time performance. For WS-IRIS the entire database is always warm. Columns 2-5 indicate the 'warm' values of the original OODBs. These are the latest best verified performance figures of OODBs we could find. The 6th column shows the 'warm' figure for a commercial relational DBMS (SYBASE[21]) also using a fast-path interface.

The WS-IRIS figure for building the database does not include saving a backup image on disk, unnecessary for an MM database. The time to save the backup image in the background is 20s.

[19]One to the index and one to the data page.

[20]The OO1 benchmark was run on Objectivity/DB, Object Design ObjectStore, Ontologic ONTOS, and VERSANT^{TM}.

[21]SYBASE is among the most efficient RDBMS, specifically designed for OLTP.

Ancestor	*Optimization level*
Database size	F
100 objects	4.4ms
1,000 objects	4.3ms
10,000 objects	4.3ms

Figure 17: Performance measurement of a Recursive Function

Measure	*WS-IRIS*	OODB1	OODB2	OODB3	OODB4	RDBMS
Db. size (Mb)	9.1	5.6	3.4	7.0	3.7	4.5
Db. build time (s)	85	133	50	47	267	370
Lookup (s)	0.07	0.1	0.03	1.1	1.0	19
Traverse (s)	0.3	0.7	0.1	1.2	1.2	84
Insert (s)	0.4	3.7	3.1	1.0	2.9	20
L+T+I (s)	0.8	4.5	3.2	3.3	6	123

Figure 18: OO1 Benchmark

The results show that the time to build the database, as well as the 'lookup' and 'traversal' time, are for WS-IRIS about those of other OODBs. Small differences are not meaningful as the original numbers are already not strictly comparable [5]. In contrast, the overall performance (L+T+I) is about four times in favor of WS-IRIS. The improvement comes from the MM residency, as updates do not need to be committed to the disk.

The figures also show that an RDBMS has significantly lower performance for OO applications than any of the OODBs. For WS-IRIS the overall performance is more than 150 times faster. This figure confirms our evaluations of the importance of MM residency in Sec. 6.1.

7 CONCLUSIONS AND FUTURE WORK

We have described the query processing in WS-IRIS. The WS-IRIS database is MM resident, and the disk is used only for backup through background operations. The optimizations and the physical data structures are large MM oriented. The locality of data is not a primary concern, allowing for data structures more efficient for OO needs. All algorithms we have addressed have been implemented. WS-IRIS is in experimental use at HP and is distributed to universities.

The overall results prove that WS-IRIS offers a very efficient interface for both navigational and relationally complete declarative use. The declarative interface also includes foreign functions, constraint inferencing, and one popular kind of recursion. None of the benchmarked OODBs offered such a relationally complete interface, while the existing relational systems prove too slow for OO needs[22]. Furthermore, we are not aware of any other OODB that would provide a declarative interface as powerful as that of WS-IRIS, including [8] and [24][23].

Efficient processing is a primary concern for OO applications. Declarative query languages are important for ad hoc queries and for mission critical application programs. Experiences reported in [32] show, that the difference in programming time between declarative and procedural languages can be a couple of minutes versus several hours. Hence, the new capabilities WS-IRIS provides are very important.

MM residency proved necessary for the performance of our system. Extensive optimization proved nevertheless necessary as well. Only the conjunction of both capabilities allowed for performance unattainable for a disk database.

For the future, the system can be extended to incorporate more query optimization techniques, e.g. to enhance the processing of recursive queries [34, 35] or perhaps optimization of the choice operator [23]. These techniques have to be re-examined in the light of their feasibility with respect to OO characteristics of ObjectLog. One should also carefully evaluate the costs of their functioning versus the expected gains for an MM resident database.

One can also add to WS-IRIS new general features, e.g. versioning, or data monitoring [25]. Again, we believe that MM residency is necessary for these extensions.

At present the access performance of a WS-IRIS database as well as its size is limited by the capabilities of a single workstation. One way to larger databases is to enhance the compactness of WS-IRIS' internal data structures. Another possibility is to use distributed data structures, especially the dynamic ones [19]. Such data structures allow to store objects over several machines linked through a fast LAN and allow for parallel low-level operations. A WS-IRIS database could then attain many gigabytes.

WS-IRIS could also be extended to a front end to an IRIS server or even a relational server [36]. The benefit would be more efficient use of these databases. Primitives would be required to check in/out objects between the

[22]Note that SQL is only relationally complete, i.e. currently does not offer recursion, constraint inferencing, or foreign functions.

[23]Neither could we find any performance evaluations of these interfaces.

server database and a private database. Techniques could be developed to post WS-OSQL queries spanning both the private and the shared database [11, 14].

Another direction should be an extension of WS-IRIS with multi-database capabilities for federated management of collections of WS-IRIS databases. This is another way to manage in an enterprise much larger data sets than a single WS-IRIS database could handle.

Finally, WS-IRIS could (and should) be extended with heterogeneous multi-database capabilities, e.g. along the lines started by Pegasus [1].

ACKNOWLEDGEMENTS:

This work was started while Tore Risch was at HP's Stanford Science Center, directed by Steven Rosenberg The support by Steven was crucial for the success of this project, as well as support by Dan Fishman, Marie-Anne Neimat, and Ming Shan Discussions with Ravi Krishnamurthy and Gio Wiederhold were helpful. Peter Lyngbaek, Jürgen Annevelink, and the Database Technology Department group provided insights into IRIS. Sang Cha contributed to the benchmarks and the ObjectLog interpreter.

References

[1] R.Ahmed, P.DeSmedt, W.Du, W.Kent, M.A.Ketabchi, W.A.Litwin, A.Rafii, M-C.Shan: “The Pegasus Heterogeneous Multidatabase System”, *IEEE Computer*, Vol. 24, No. 12, Dec. 1991.

[2] J.Annevelink: “Database Programming Languages: A Functional Approach”, *ACM SIGMOD Conf.*, pp. 318-327, 1991.

[3] S.Abiteboul, A.Bonner: “Objects and views”, *ACM SIGMOD Conf.*, pp. 238-247, 1991.

[4] M.J.Carey, D.J.DeWitt, and S.L.Vandenberg: “A data model and query language for EXODUS”, *ACM SIGMOD Conf.*, pp. 413-423, 1988.

[5] R.G.G.Cattell, J.Skeen: “Object Operations Benchmark”, *ACM Transactions on Database Systems*, Vol. 17, No. 1, pp. 1-31, March 1992.

[6] D.Chimenti, R.Gamboa, R.Krishnamurthy: “Towards an Open Architecture for $\mathcal{LDL}$”, *15th VLDB Conf.*, pp. 195-204, 1989.

[7] T.Connors, P.Lyngbaek: “Providing Uniform Access to Heterogeneous Information Bases”, In *Advances in Object-Oriented Database Systems*, K.R.Dittrich, Ed., Lecture Notes in Computer Science 334, Springer-Verlag, Sept. 1988.

[8] O.Deux: “The O_2 System”, *CACM*, Vol. 34, No. 10, October 1991.

[9] M.Guo, S.Y.W. Su, H.Lam: “An association algebra for processing object-oriented databases”, *7th Data Engineering Conf.*, pp. 23-32, 1991.

[10] D.Fishman et al.: “Overview of the IRIS DBMS”, in W.Kim, F.H.Lochovsky (eds.): *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley, 1989.

[11] *ITASCA Technical Summary*, ITASCA Systems, Inc., 1990.

[12] A.Kemper, G.Moerkotte: “Advanced query processing in object bases using access support relations”, *16th VLDB Conf.*, pp. 290-301, 1990.

[13] W.Kim, F.H.Lochovsky: *Object-Oriented Concepts, Databases, and Applications*, ACM Press, 1989.

[14] W.Kim: *Introduction to Object-Oriented Databases*, MIT Press, 1990.

[15] R.Krishnamurthy, H.Boral, C.Zaniolo: “Optimization of Nonrecursive Queries”, *12th VLDB Conf.* , pp. 128-137, 1986.

[16] R.Krishnamurthy, S.Zaniolo: "Optimization in a Logic Based Language for Knowledge and Data Intensive Applications", *Advances in Database Technology - EDBT '88*, pp. 16-33, 1988

[17] W.Leler: *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, 1988

[18] K.Li, J.F.Naughton: "Multiprocessor Main Memory Transaction Processing", *ISPDS*, IEEE CS, Austin TX, Dec., 1988

[19] W.Litwin, M-A.Neimat: *Distributed Linear Hashing*, Technical Report HPL-DTD-92-7, Database Technology Dept., HP-Laboratories, Palo Alto.

[20] Y.Lou, Z.M.Ozsoyoglu: "LLO: An object-oriented deductive language with methods and method inheritance". *ACM SIGMOD Conf.*, pp. 198-207, 1991.

[21] D.Maier, J.Stein: "Development of an Object-Oriented DBMS", *OOP-SLA Conf.*, pp. 472-482, 1986.

[22] A. Mycroft, R.A. O'Keefe: "A Polymorphic Type System for Prolog", *Artificial Intelligence* 23, pp. 295-307, 1984.

[23] S.Naqvi, S.Tsur: *A Logical Language for Data and Knowledge Bases*, Computer Science Press, 1989.

[24] J.Orenstein, S.Haradhvala, B.Margulies, D.Sakahara: "Query Processing in the ObjectStore Database System", *ACM SIGMOD Conf.*, pp. 393-402, 1992.

[25] T.Risch: "Monitoring Database Objects", *15th VLDB Conf.*, pp. 445-453, 1989.

[26] T.Risch: *WS-IRIS, a Main Memory Object-Oriented DBMS*, Technical Report HPL-DTD-92-5, Database Technology Dept., HP-Laboratories, Palo Alto.

[27] S.B.Zdonik, D.Maier: *Readings in Object-Oriented Database Systems*, Morgan-Kaufman, 1990.

[28] P.G.Selinger, M.M.Astrahan, D.D.Chamberlin, R.A.Lorie, T.G.Price: "Access Path Selection in a Relational Database Management System", *ACM SIGMOD Conf.*, pp. 23-34, 1979.

[29] G.M.Shaw, S.B.Zdonik: "Object-oriented queries: Equivalence and optimization", *1st Conf. on Deductive and OO Databases*, pp. 264-278, 1989.

[30] M.Stonebraker, L.Rowe: "The design of POSTGRES", *ACM SIGMOD Conf.*, pp. 340-355, 1986.

[31] D.D.Straube, M.T.Özsu: "Queries and query processing in object-oriented database systems", *ACM Transactions on Information Systems*, 8(4), pp. 387-430, October, 1990.

[32] M.Takizawa: "Distributed Database System JDDBS", *JARECT Computer Science & Technologies*, Vol 7, OHMSHA & North Holland (publ.), 262-283, 1983.

[33] S.Tsur, N.Garrison: *LDL User's Guide*, MCC Technical Report STP-LD-295-91, 1991.

[34] J.D.Ullman: *Principles of Database and Knowledge-Base Systems*, Volume I & II, Computer Science Press, 1988, 1989.

[35] D.S.Warren: "Memoing for Logic Programs", *CACM*, Vol. 35, No. 3, March 1992.

[36] G.Wiederhold: "Views, objects, and databases", *IEEE Computer*, 19(12), pp. 37-44, 1986.

[37] K.Wilkinson, P.Lyngbaek, W.Hasan: "The IRIS Architecture and Implementation", *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mars, 1990.

A Example of Cost Based Optimization

As an illustration of how our cost heuristics work, we describe how `grandsparentgrossincome` of Fig. 6 is reordered into the optimal program of Fig. 7. We assume a database of 10,000 person with 2,500 students, and that every child is expected to have 2 parents.

To reorder the program of Fig. 6 with these assumptions the optimizer will do the following calculations. Initially the only bound variable is `C`. The first rankings are calculated as:

P_1 = income^{ff}(GP,_V2)
F$_{P_1}$ = 10,000
C$_{P_1}$ = 20,000 (assumes cost 2 per tuple visited)
R$_{P_1}$ = 0.49995
P_2 = bonus^{ff}(GP,_V1)
F$_{P_2}$ = 10,000
C$_{P_2}$ = 20,000
R$_{P_2}$ = 0.49995
P_3 = plus^{fff}(_V2,_V1,GI)
R$_{P_3}$ = undefined (cannot execute here)
P_4 = typesof^{bf}(GP,typeStudent)
F$_{P_4}$ = 2,500
C$_{P_4}$ = 5,000 (assuming 2,500 students)
R$_{P_4}$ = 0.3998
P_5 = parent^{ff}(P,GP)
F$_{P_5}$ = 13,000 (size of parent-child table)
C$_{P_5}$ = 26,000
R$_{P_5}$ = 0.49996
P_6 = parent^{bf}(C,P)
F$_{P_6}$ = 2
C$_{P_6}$ = 4 (indexed)
R$_{P_6}$ = 0.25 (best rank!)

The ranking places P_6 first in the function body. Then both `C` and `P` are bound. The new rankings will be:

P_1 = income^{ff}(GP,_V2)
R$_{P_1}$ = 0.49995 (unchanged)
P_2 = bonus^{ff}(GP,_V1)
R$_{P_2}$ = 0.49995 (unchanged)
P_3 = plus^{fff}(_V2,_V1,GI)
R$_{P_3}$ = undefined (unchanged)
P_4 = typesof^{bf}(GP,typeStudent)

R$_{P_4}$ = 0.3998 (unchanged)
P_5 = parent^{bf}(P,GP)
F$_{P_5}$ = 2
C$_{P_5}$ = 4
R$_{P_5}$ = 0.25 (best rank!)

After P_5 is chosen C, P, and GP are bound. New rankings:

P_1 = income^{bf}(GP,_V2)
F$_{P_1}$ = 1 (index)
C$_{P_1}$ = 2
R$_{P_1}$ = 0 (best rank!)
P_2 = bonus^{bf}(CP,_V1)
F$_{P_2}$ = 1 (index)
C$_{P_2}$ = 2
R$_{P_2}$ = 0 (best rank!)
P_3 = plus^{fff}(_V2,_V1,GI)
R$_{P_3}$ = undefined (unchanged)
P_4 = typesof^{bb}(GP,typeStudent)
F$_{P_4}$ = 1
C$_{P_4}$ = 1
R$_{P_4}$ = 0 (best rank!)

The system now chooses P_1 and then P_2, making C, P, GP, _V1, and _V2 bound. New rankings:

P_1 = plus^{bbf}(_V2,_V1,GI)
F$_{P_1}$ = 1
C$_{P_1}$ = 1
R$_{P_1}$ = 0 (best rank!)
P_2 = typesof^{bb}(GP,typeStudent)
F$_{P_2}$ = 1
C$_{P_2}$ = 1
R$_{P_2}$ = 0 (best rank!)

After the final two choices, we get the optimal program of Fig. 7