The Iris Architecture and Implementation

Kevin Wilkinson
Peter Lyngbæk
Waqar Hasan

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

Abstract

The Iris database management system is a research prototype being developed at Hewlett-Packard Laboratories. Its goals are to enhance database programmer productivity and to provide generalized database support for the integration of future applications. Iris is based on an object and function model. Iris objects are typed, but, unlike other object systems, Iris objects contain no state. Attribute values, relationships and behavior of objects are modeled by functions. The Iris architecture efficiently supports the evaluation of functional expressions. The goal of the architecture is to provide a database system that is powerful enough to support the definition of functions and procedures that implement the semantics of the data model. This paper provides an overview of the data model, describes the architecture in detail and discusses our implementation experience and usage of the system.

1 Introduction

Iris is an object-oriented database management system being developed at Hewlett-Packard Laboratories [Fishman,89] [Fishman,87]. One of its goals is to enhance database programmer productivity by developing an expressive data model. Another goal is to provide generalized database support for the development and integration of future applications in areas such as engineering information management, engineering test and measurement, telecommunications, office information, knowledge-based systems, and hardware and software design. These applications require a rich set of capabilities that are not supported by the current generation (i.e., relational) DBMSs.

Figure 1 illustrates the major components of the Iris system. Central to the figure is the Iris Kernel, the retrieval and update processor of the DBMS. The Iris Kernel implements the Iris data model [Lyngbaek,86] which is an object and function model. Retrievals and updates are written as functional expressions. Extensibility is provided by allowing users to define new functions. The functions may be implemented as stored tables or derived as computations. The computations may be expressed either as Iris functional expressions or as foreign functions in a general-purpose programming language, such as C.

Like most other database systems, Iris is accessible via stand-alone interactive interfaces or interfaces embedded in programming languages. All interfaces are built as clients of the Iris Kernel. A client formats a request as an Iris expression and then calls an Iris Kernel entry point that evaluates the expression and returns the result which is also formatted as an Iris expression.

Currently, two interactive interfaces are supported. One interactive interface, Object SQL (OSQL), is an object-oriented extension to SQL. The second interactive interface is the Graphical Editor. It is an X-Windows-based system that allows users to retrieve and update function values and metadata with graphical and forms-based displays.

In addition to the Kernel interface, Iris supports two other programmatic interfaces. The first, CLI (C Language Interface), is a user-friendly layer on top of the base Kernel interface. It allows programmers to access Iris in an object-oriented fashion by manipulating C variables denoting the Iris database, the Iris meta-data including types and functions, and the objects in the database. The second programmatic interface is a straightforward embedding of OSQL into various host

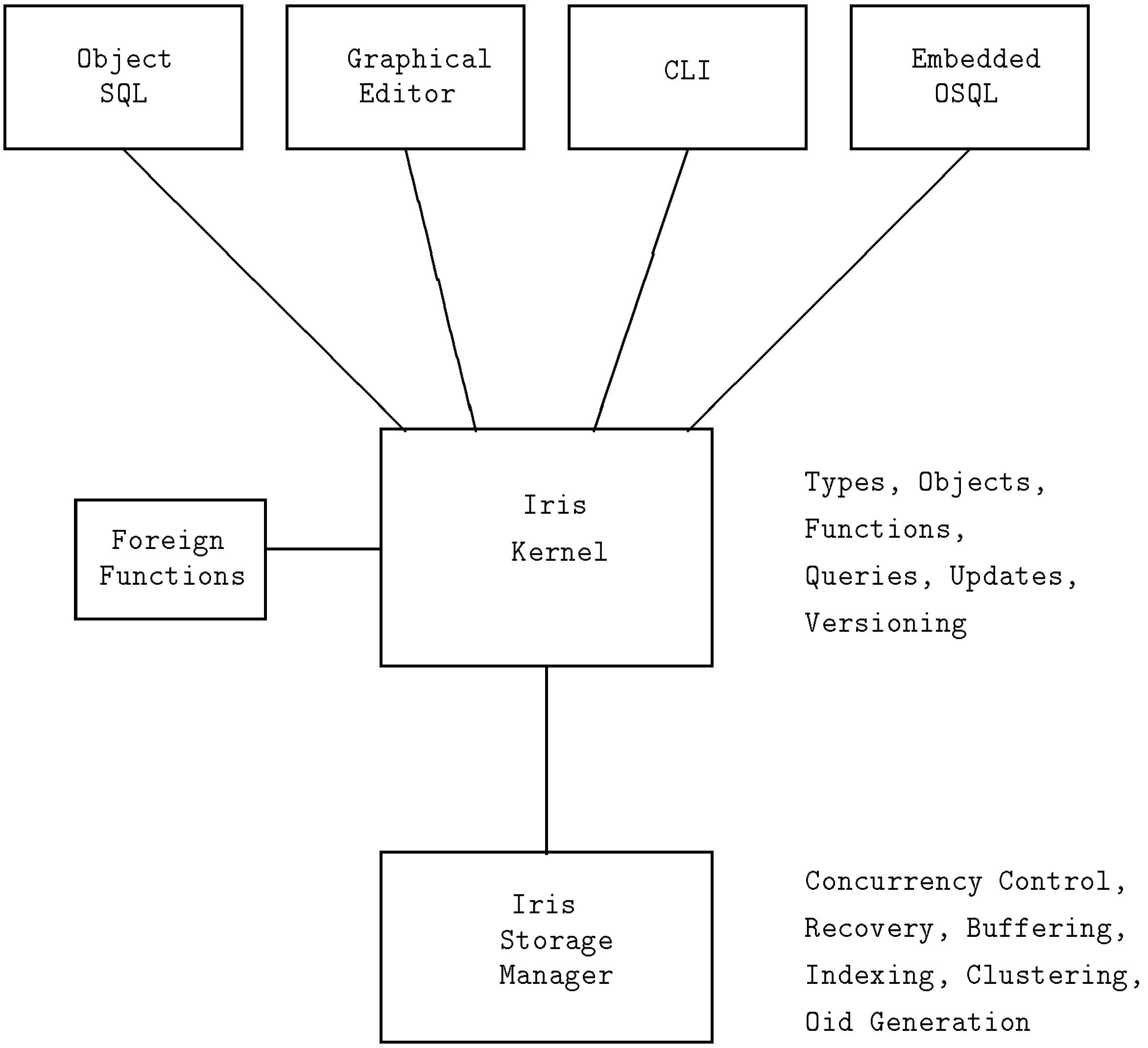

Figure 1: *Iris System Components*

languages.

One of the long-term goals of the Iris project is to be able to define and implement the Iris model in terms of its own functions [Lyngbaek,86]. This provides a conceptual simplicity with the result that the implementation of the system is easier to understand and maintain. It is also easier to prototype new operations with such a system because data model operations can be prototyped as ordinary database functions. An added advantage is that it will be possible to optimize and type check data model operations like ordinary database functions.

Since the essence of the Iris data model is function application and updating, the Iris Kernel has been architected around the single operation of invoking a function. In addition, the Kernel may call itself recursively so that data model operations may invoke other data model operations. This flexibility permits the customization of Iris operations and allows us to experiment with different semantics of, for example, multiple inheritance, versioning, and complex objects with little re-implementation effort.

The emphasis of this paper is to describe the Iris Kernel architecture. Section 2 gives a brief overview of the Iris Data Model. Section 3 describes the Iris Kernel architecture. Section 4 is a look back on our implementation experiences. Section 5 provides a summary.

2 Overview of the Iris Data Model

The Iris Database System is based on a semantic data model that supports abstract data types. Its roots can be found in previous work on Daplex [Shipman,81] and Taxis [Mylopoulos,80]. A number of recent data models, such as PDM [Manola,86] and Fugue [Heiler,88], also share many similarities with the Iris Data Model. The Iris data model contains three important constructs: *objects*, *types* and *functions*. These are briefly described below. A more complete description of the Iris Data Model and the Iris DBMS may be found in [Lyngbaek,86, Fishman,89].

2.1 Objects and Types

Objects in Iris represent entities and concepts from the application domain being modeled. Some objects such as integers, strings, and lists are self identifying. Those are called *literal objects*. Literal types have a fixed extension.

A *surrogate* object is represented by a system-generated, unique, immutable *object identifier* or *oid*. Examples of surrogate objects include system objects, such as types and functions, and user objects, such as employees and departments.

Types have unique names and are used to categorize objects into sets that are capable of participating in a specific set of functions. Objects serve as arguments to functions and may be returned as results of functions. A function may only be applied to objects that have the types required by the function.

Types are organized in an acyclic type graph that represents generalization and specialization. The type graph models inheritance in Iris. A type may be declared to be a subtype of other types (its supertypes). A function defined on a given type is also defined on all its subtypes. Objects that are instances of a type are also instances of its supertypes. The system types are shown in Figure 2 where the subtype relationship is illustrated with arrows from a supertype to its subtypes. User objects may belong to any set of user defined types. In addition, objects may gain and lose types dynamically. For example, an object representing a given person may be created as an instance of the Employee type. Later it may lose the Employee type and acquire the type Retiree. When that happens, all the functions defined on Retiree become applicable to the object and the functions on Employee become inapplicable. This feature enables us to support database evolution better than other object-oriented systems in which the types of an object may be specified only at the time the object is created.

Objects in Iris may be versioned. An object being versioned is represented by a *generic* object instance and a set of distinct *version* object instances corresponding to each version of the object. By default, objects are not versioned, i.e. they have the type `UnVersioned`. The Iris versioning mechanism is further described in [Beech,88].

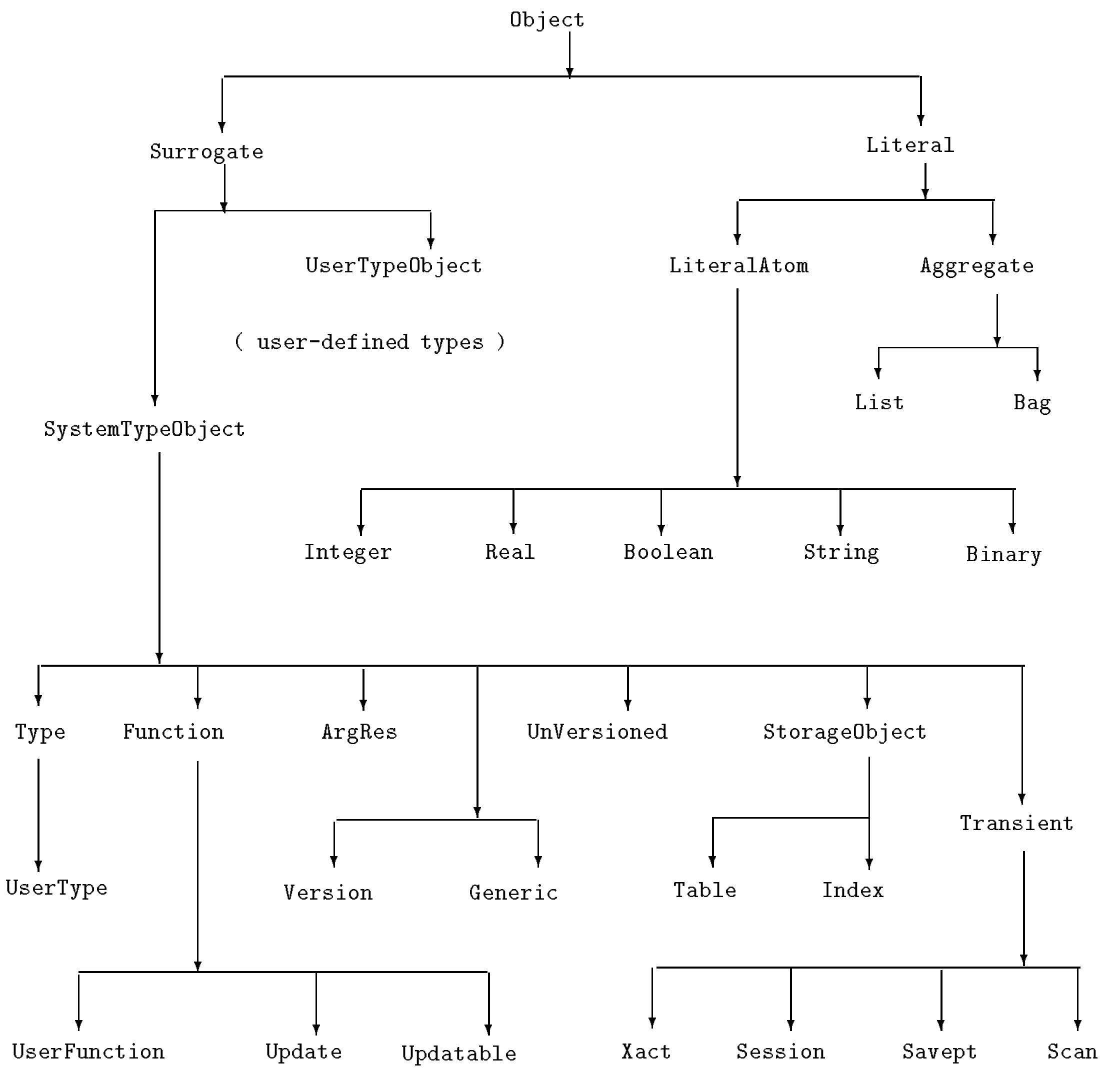

Figure 2: *Iris System Types*

2.2 Functions

Attributes of objects, relationships among objects, and computations on objects are expressed in terms of functions. Iris functions are defined over types, they may be many-valued and, unlike mathematical functions, they may have side-effects. In Iris, the declaration of a function is separated from its implementation. This provides a degree of data independence. This section discusses function declaration and Section 2.4 discusses function implementation.

In order to support stepwise refinement of functions, function names may be overloaded, i.e. functions defined on different types may be given identical names. When a function call is issued using an overloaded function name, a specific function is selected for invocation. Iris chooses the function that is defined on the most specific types of the actual arguments.

A type can be characterized by the collection of functions defined on it. The Employee type might have the following functions defined over it:

`JobTitle:` `Employee` $\rightarrow$ `String`

`EmpDept:` `Employee` $\rightarrow$ `Department`

`Manager:` `Employee` $\rightarrow$ `Employee`

`SalHist:` `Employee` $\twoheadrightarrow$ `Integer` $\times$ `Date`

`ChangeJob:` `Employee` $\times$ `String` $\times$ `Department` $\rightarrow$ `Boolean`

If Smith is working as a software engineer in the Toolkit Department reporting to Jones then the function values are as follows (references to surrogate objects are denoted by italics):

`JobTitle(`*Smith*`)` = `''Software Engineer''`

`EmpDept(`*Smith*`)` = *Toolkit*

`Manager(`*Smith*`)` = *Jones*

The SalHist function is many-valued. It is also an example of a function with multiple result types. Each result is a pair of salary and date objects, where the salary is represented by an integer. The date indicates when the salary was changed. If Smith was hired on 3/1/87 with a monthly salary of $3000 and given a raise of $300 on 3/1/88 then the salary history function has the following value:

```
SalHist(Smith) = [<3000, 3/1/87>, <3300, 3/1/88>]
```

Note that the dates are represented by surrogate object identifiers (they are in italics). Thus, there are, presumably, functions on date objects that materialize parts or all of the date, e.g. month, day and year functions.

In Iris, we use the term *procedure* to refer to a function whose implementation has side-effects. The function ChangeJob is an example of a function with multiple argument types. Let us assume that its implementation will be used to assign a given employee to a given department and to change the job title as specified. In this case, ChangeJob is a procedure. The promotion of Smith to Project Manger in the Applications Department can be reflected in the database by the following procedure invocation:

```
ChangeJob(Smith, ``Project Manager'', Applications)
```

In Iris, a new function is *declared* by specifying its name, the types of its argument and result parameters and, optionally, names for the arguments and results[1].

create function `Manager( Employee ) → supervisor Employee;`

Before a function may be invoked, an *implementation* must be specified. This process is described in Section 2.4.

2.3 Database Updates and Retrievals

Properties of objects can be modified by changing the values of functions. For example, the following operations will cause the JobTitle function to return the value "MTS" in a future invocation with the parameter Smith and add another salary and date pair to Smith's salary history:

[1]For readability, subsequent examples will be expressed in OSQL [Fishman,89]. Keywords of this language appear in bold font. Keep in mind, however, that OSQL is parsing each statement into a functional expression that invokes an Iris system or user function.

```
set JobTitle(Smith) = ''MTS'';

add Salhist(Smith) = <3800, 1/1/89>;
```

In addition to setting and adding function values, one or more values may be removed from the value-set of a many-valued function. The ability to update a function's values depends on its implementation. In general, functions whose values are stored as a table can always be updated. However, functions whose values are computed may or may not be updatable (see Section 2.4).

The database can be queried by using the OSQL `select` statement and specifying a list of results, a list of existentially quantified variables, and a predicate expression. The result list contains variables and function invocations. The predicate may use variables, constants (object identifiers or literals), nested function applications, and comparison operators. Query execution causes the existential variables to be instantiated. The result list is then used to compute a result value for each tuple of the instantiated existential variables. The collection of all result values is returned as a bag.

The following statement retrieves all the dates on which Smith's salary was modified:

```
select d
for each Date d, Integer s
where SalHist( Smith ) = <s, d>;
```

and the statement:

```
select Manager( Smith );
```

returns Smith's manager.

Retrievals must be side-effect free, i.e. they may not invoke procedures.

2.4 Function Implementation

So far, we have discussed the declaration of functions and their use in retrievals and updates. An important additional characteristic of a function is its *implementation* or *body*, that is, the specification of its behavior. The implementation of the function is compiled and optimized into an internal format that is then stored in the system catalog. When the function is later invoked, the compiled representation is retrieved and interpreted.

Iris supports three methods of function implementation: Stored, Derived, and Foreign. A stored implementation explicitly maintains the extension of the function as a stored table in the database. Derived and foreign implementations are alternative methods for computing function values.

Stored Functions

The extension of a function may be explicitly maintained by storing the mapping of corresponding argument and result values as tuples in a database table. Since our storage manager does not support nested structures, the result of a many-valued function is stored as several tuples with identical argument values.

To improve performance, functions with the same argument types may be clustered in a single table by extending the tuple width to include result columns for each clustered function. As illustrated in Section 2.3, stored functions may be updated by using the OSQL `set` and `add` statements. There is also a `remove` statement that is not shown. A formal treatment of the mapping of Iris functions to relational tables may be found in [Lyngbaek,87].

Derived Functions

Derived functions are functions that are computed by evaluating an Iris expression. The expression may represent a retrieval or an update. As an example of a retrieval function, the `select` statement in Section 2.3 could represent the body of a derived function with zero arguments that retrieves the dates on which Smith's salary was modified.

A function may also be derived as a sequence of updates, which defines a procedure. For example the first two updates in Section 2.3 may be encapsulated as a procedure that updates Smith's personnel data. Procedures themselves may not be updated.

A derived function without side-effects may be thought of as a view of the stored data. The semantics of updates to such a function are not always well-defined. For example, if the derivation expression of a given function requires joining several tables, the function cannot be directly updated. However, in those cases where Iris can solve the "view update" problem, the update actions are automatically inferred by Iris. For example, functions that are derived as inverses of stored functions are updatable.

Foreign Functions

A foreign function is implemented as a subroutine written in some general-purpose programming language and compiled outside of Iris. The implementation of the foreign function must adhere to certain interface conventions. Beyond that, the implementation is a black box with respect to the rest of the system.

This has three consequences. First, it is impossible to determine at compile time whether the implementation has side-effects. For this reason, users must specify whether or not their foreign function has side-effects. Second, foreign functions cannot be updated. Third, the implementation of foreign functions cannot be optimized by Iris. However, their usage can, potentially, be optimized. For example, given a foreign function that computes simple arithmetic over two numbers, rules could be added to evaluate the result at compile time if the operands are constants.

Foreign functions provide flexibility and extensibility. Since the Iris database language is not computationally complete, there are certain computations that cannot be expressed as derived functions. Foreign functions provide a mechanism for incorporating such computations into the system. Furthermore, an existing program can be integrated with Iris through the foreign function mechanism either by modifying the program to adhere to the foreign function calling conventions or by writing new foreign functions to provide an interface to the existing program. Foreign functions and the mechanisms with which they are supported in Iris are described in detail in [Connors,88].

2.5 Iris System Objects

In Iris, types and functions are also objects. They are instances of the system types, `Type` and `Function`, respectively (see Figure 2). Like user-defined types, system types have functions defined on them. The collection of system types and functions model the Iris metadata and the Iris data model operations.

System functions are used to retrieve and update metadata and user data. Examples of retrieval functions include, `FunctionArgcount`, that returns the number of arguments of a function, `SubTypes`, that returns the subtypes of a type, and `FunctionBody`, that retrieves the compiled representation of a function. *System procedures* correspond to the operations of the data model are used to update metadata and user data. Examples of system procedures include `ObjectCreate`, to create a new object, `FunctionDelete`, to delete a function and `IndexCreate`, to create an index.

`Select` and `Update` are two important system functions that may be used to access metadata or user data. `Select` is the system function that corresponds to the OSQL `select` statement illustrated in Section 2.3. `Update` is used to modify function values. It corresponds to the OSQL `set`, `add` and `remove` statements.

As with user functions, system functions (and procedures) may have either stored, derived or foreign implementations. Currently, there are a number of *system foreign functions*. These exist either because their functionality cannot be expressed as Iris functional expressions or they are more efficiently implemented as foreign functions. Most system procedures are implemented as foreign functions. System foreign functions are also used to implement transaction support, e.g., `XactCommit` and `XactRollback`, and facilities for source code tracing and timing.

In order to compile and execute functions, the Iris Kernel needs access to metadata (a system catalog) that describes the database schema. The Iris system catalog is maintained as a collection of stored system functions. Since certain system functions are frequently accessed together, most of the functions for a particular type of object are clustered on the same table. For example, the *function* table is a clustering of the `FunctionName`, `FunctionArgcount`, and `FunctionResultcount` functions, among others. The current system catalog consists of approximately 100 functions clustered in 15 tables.

3 Iris Kernel Architecture

3.1 Overview

The Iris Kernel is a program that implements the Iris data model. The Kernel architecture shares many similarities with an architecture described in [Buneman,82]. This section describes the component modules of the Iris Kernel in more detail and concludes with an example that illustrates the flow of execution for a sample request.

The Iris Kernel is invoked via a subroutine entry point that serves as a function call evaluator. Iris requests are formatted as Iris expressions. Each node in an Iris expression is self-identifying and consists of a header and some data fields.

The header defines the node type[2]. The possible node types are: object identifier, variable, function call and one node type for each Iris literal type (i.e. *integer*, *real*, *boolean*, etc.). Iris provides a subroutine library to create and manipulate nodes and expressions. Once the request is properly formatted, it may be passed to the Iris Kernel for evaluation. The returned results are also formatted as Iris expressions.

All user and system operations are invoked via function calls. This represents a range of capabilities from low-level operations, such as comparison or equality checking, up to high-level operations, such as function or object creation.

3.1.1 Kernel Modules

The Kernel is organized as a collection of software modules. They are layered as illustrated in Figure 3. The top-level module, the Executive (EX), implements the Kernel entry points and manages the client-Kernel interaction. For each request, it calls the Query Translator, QT, to produce a relational algebra tree for the request. EX then passes this tree to the Query Interpreter, QI, which produces the result expression.

The Object Manager, OM, is a set of system procedures and functions that are implemented as foreign functions (denoted `ff` in Figure 3). The Cache Manager, CM, is an intermediate layer

[2]This is a distinct notion from an object type. Node types merely identify interface data structures.

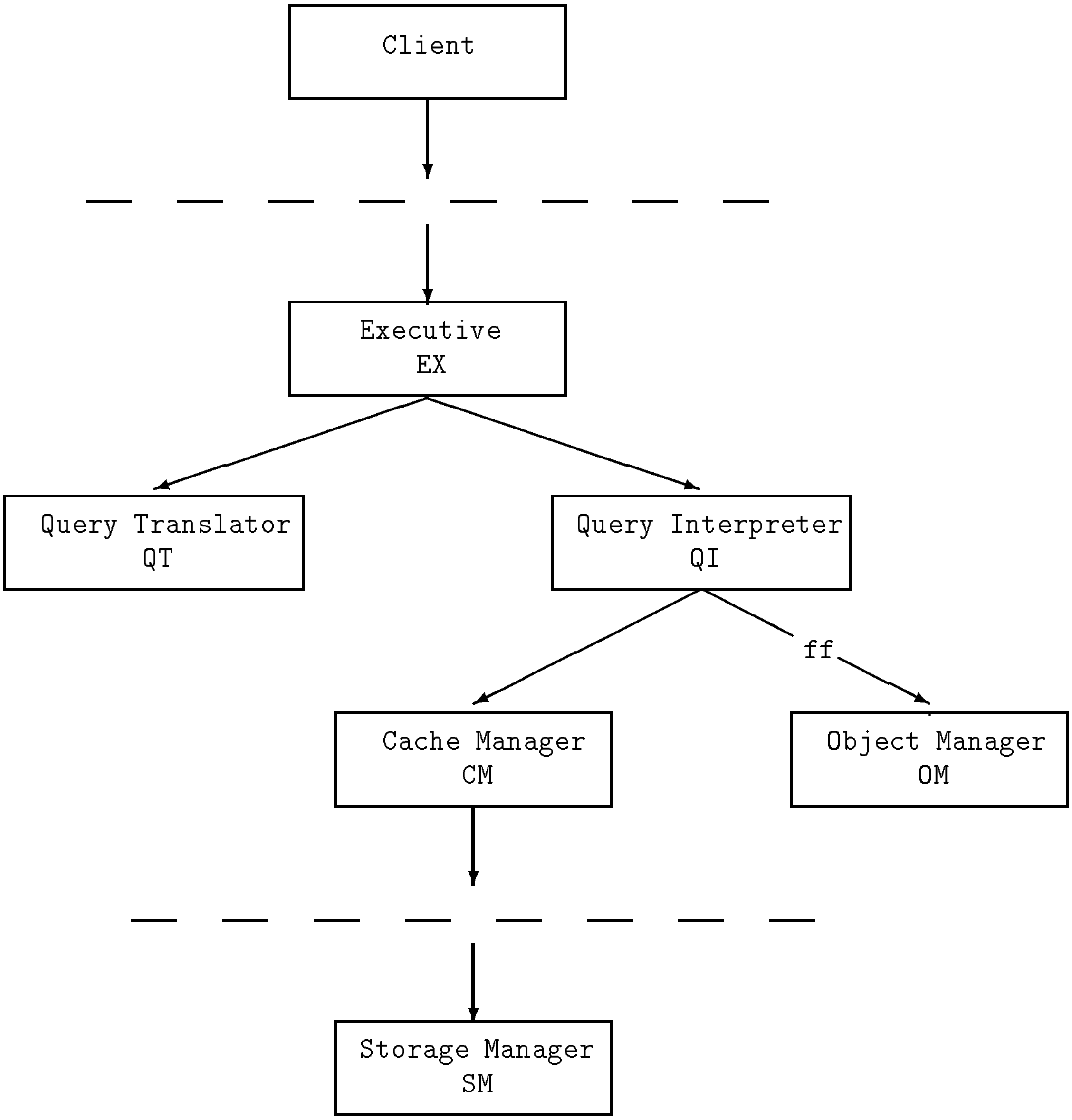

Figure 3: *Iris Kernel Architecture*

between the Iris Kernel and the Storage Manager, SM. It provides prefetching and cache management for data retrieval and data updates between the Kernel and SM. The Storage Manager provides data sharing, transaction management and access to stored tables.

The Kernel may be called recursively through EX. The Query Translator makes recursive calls to invoke system functions that retrieve metadata. Some high-level system procedures in the Object Manager make recursive calls to invoke lower level system procedures and functions.

The Iris Kernel is single-threaded and each client runs its own copy of the Kernel. The Kernel may execute as a server in a separate process and communicate with the client via messages. Alternatively, the client and Kernel may be tightly coupled in the same process and communicate via subroutine calls. In either case, the configuration is transparent to the source code of the client. The Storage Manager always executes in the same address space as the Kernel. The multiple instances of the Storage Manager use a shared memory buffer for caching data, concurrency control and transaction logging.

3.2 Iris Executive

The Executive module, EX, manages interaction between the Iris Kernel and its clients and implements the Kernel entry points. A request consists of a functional expression, a result buffer and an error buffer. The result buffer is filled with result objects produced by evaluating the expression tree. The error buffer is filled with any error messages generated during the processing of the request.

Request processing consists of two steps: compile and interpret. The compilation step, done by QT, converts the functional expression into an extended relational algebra tree. The interpreter, QI, then traverses the tree and produces the result objects for the request. The structure of the result depends on the request. Invoking a many-valued function returns a bag of objects while invoking a single-valued function returns a single object. If the result buffer is too small to contain the entire result object, as much as will fit in the buffer is returned and an error message is generated.

To ensure that no results are lost, the client may open a scan over the result object. This may be done by calling a system function that takes a bag and returns a scan object. Alternatively, as a convenience since opening a scan is a frequent operation, EX provides a separate entry point that always opens a scan on the results of a request.

Iris supports two sets of EX entry points: one for clients, the other for internal Kernel calls. This was done to decrease the internal path length and improve performance for recursive Kernel calls. This is possible because the internal calls are considered trustworthy, so it is safe to skip some of the type and sanity checks that are done for ordinary requests. The internal entry points also provide a form of security. Certain system functions may only be invoked through the internal entry points and, therefore, they can be hidden from the client.

3.3 Query Translator

The Query Translator, QT [Derrett,89], compiles an Iris functional expression into an execution tree. The functional expression, formatted as a tree, is known as an *F-tree*. The nodes of an F-tree include function calls, variables, and literal nodes. The execution tree is an extended relational algebra tree referred to as an *R-tree*.[3]

Currently, the Query Translator is limited in that every request expression must be rooted by a function call node and the call arguments must be constants. However, QT permits specific arguments to some functions to be arbitrary expressions. For example the predicate argument to the Iris system function, `Select`, may be an expression containing variables and nested function calls. Currently, these functions are treated as special-cases by QT. We plan to extend QT by allowing any function call argument to be an expression that will be evaluated before invoking the function. However, the semantics of such expressions remain to be defined. Conventional *eval-apply* algorithms may not be consistent with the semantics of existing Iris functions, such as `Select`.

Thus, in most cases, the task of QT is straightforward. It checks that the actual arguments of the function call are the same types or subtypes of the corresponding formal arguments and resolves function name overloading. Then, it retrieves the previously compiled R-tree for the

[3] Not to be confused with spatial R-trees.

function, substitutes the actual arguments and returns the resulting R-tree. This is known as the QT *fastpath* since it avoids the full compilation process described below.

For those system functions, such as `Select`, that have expressions as arguments, QT uses the full request translation process to generate the R-tree. The process consists of three main steps. First, the F-tree is converted to a canonical form, in which, for example, nested function calls are unnested by introducing auxiliary variables. Type checking is also performed and the names of overloaded functions are resolved.

The second step converts the canonical F-tree to an unoptimized R-tree. This is a mechanical process in which function calls are replaced by their stored implementations which are, themselves, R-trees. The resulting R-tree consists of nodes for the relational algebra project, filter[4] and cross-product operators, and *table* nodes which represent scans over stored tables. Joins are specified by placing a filter node above a cross-product node to compare the columns of the underlying cross-product.[5] To increase the functionality of the Query Interpreter, there are some additional nodes. A *temp-table* node creates and, optionally, sorts a temporary table. An *update* node modifies an existing table. A *sequence* node executes each of its subtrees in turn. A *foreign function* node invokes the executable code that is the implementation of a foreign function. The leaves of the R-tree are either table nodes or foreign function nodes. Note that a table node may have an associated predicate and projection list to reduce the size of the scan.

The initial R-tree is then checked to ensure that all declared variables are bound to a column of their declared type. If a variable is not bound, the R-tree is joined with the extension of the type of the variable. Of course, this is only done for types with finite extensions, e.g. unbound integer variables are not allowed. Note that foreign functions require that their input arguments be bound to a value before invocation since the subroutine that implements them expects to be passed a value. However, stored functions are not so restricted since the underlying table that implements the function can bind the arguments as well as the results. Thus, the inverse of stored functions may be computed without deriving a new function by merely invoking the function with

[4] We use the term *filter* for the relational algebra select operator to avoid confusion with the Iris Select function.

[5] Of course, joins are rarely executed this way because the filter predicate is typically pushed down into a table node below the cross-product to produce a nested-loops join.

bindings for the result values and leaving the arguments unbound.

The final and most complex step is to optimize the R-tree. The optimizer is rule-based. Each rule consists of a test predicate and a transformation routine. If the predicate evaluates to true, the transformation routine is invoked. Both the predicate and transformation routines are written as C subroutines that take an R-tree node as an argument. As in [Graefe,87], the system must be recompiled whenever the rules are modified.

Rules are organized into *rule sets* such that each rule set accomplishes a specific task. For example, one rule set contains all rules concerned with simplifying constant expressions (e.g. constant propagation and folding). Another rule set reorders the tables in a cross-product to take advantage of indexes and to ensure that input arguments are bound where necessary. Optimization is accomplished by applying the rule sets in a specific order over the R-tree tree. Within a rule set, QT traverses the tree and applies the rules in order of their declaration. However, the rule writer may modify the evaluation order in certain ways by setting flags that, for example, inhibit future firing of a rule on a specific node or reevaluate all rules on the entire R-tree.

3.4 Query Interpreter

The Query Interpreter, QI, evaluates an R-tree which yields a collection of tuples that become the result objects for the Iris request expression. It uses the conventional technique of piping data between parent and child nodes in the execution tree. Each node in the R-tree is treated as a scan object and must implement three operations: *open*, *next*, and *close*. These operations may call QI recursively to evaluate a subtree.

An *open* operation on a table node creates a Storage Manager scan. An *open* operation on a foreign function node may perform some data-dependent initializations. Note that, due to its potentially large size, the object code for a foreign function is stored separately from an R-tree that invokes it. QI uses a dynamic loader to load and link the object code at run time. However, the object code for system foreign functions is memory resident because they are frequently accessed.

A *next* operation produces the next tuple in the scan produced by an R-tree node. A *next* operation on an update node will update the database. But, if the subtree of an update node references the stored table that is being updated, the update tuples are spooled into a temporary table. This prevents cycles in the data pipeline.

One consequence of using a scan paradigm for the interpreter is that a stream of objects is always produced. Normally, the Query Interpreter converts the stream into a bag. However, this causes a problem when invoking a single-valued function since the caller expects a single object to be returned rather than a bag. To prevent unexpected bagging, QT returns a boolean flag as part of compilation that indicates if the R-tree is single-valued. If the flag is true, QI will not bag the result.

3.5 Object Manager

The Object Manager, OM, is a set of system foreign functions. These functions provide services that are essential to Iris but whose implementations either cannot be expressed as stored or derived functions or are more efficiently written as foreign functions.

In the current version of Iris, most of the system procedures are implemented as foreign functions. One problem with this is that QT cannot optimize calls to these procedures. Thus, we plan to reimplement many of the Iris system procedures as extended relational algebra expressions using *update* and *sequence* nodes. Then, a complex system procedure could be implemented as a derived function using a sequence of updates to individual stored system functions. This work is currently under investigation. It may require the addition of new relational algebra nodes, e.g. a branching node.

3.6 Cache Manager

The Cache Manager, CM, implements a general-purpose caching facility between the Iris Kernel and the Storage Manager. An important point is that, since the Query Interpreter operates on relational algebra trees, CM caches tables not functions. The Cache Manager maintains two types of table caches: a *tuple cache* and a *predicate cache*. The tuple cache is used to cache tuples from individual tables. A table may have at most one tuple cache. A tuple cache is accessed via a column of the table and that column must be declared as either uniquely-valued or *many-valued.*

If a column is declared as many-valued, the cache ensures that whenever a given value of that column occurs in the cache, all tuples of the table with the same column value will also occur in the cache. This guarantees that when a cache hit on a many-valued column occurs, the scan can be entirely satisfied from the cache without having to invoke the Storage Manager. This enables the effective caching of many-valued functions.

A table may have many predicate caches. A predicate cache may be considered the materialization of a table node in an R-tree. It contains tuples from a table that satisfy a particular predicate. Thus, a predicate cache has an associated predicate and projection list. A predicate cache is useful in caching intermediate results during R-tree interpretation. Tuple caches are primarily intended to support caching of system tables. However, user tables may also be cached in this way. Information in the cache is always kept consistent with the Storage Manager. If an update request is too complicated to preserve cache consistency, the table will be automatically uncached.

3.7 Storage Manager

The Iris Storage Manager is a conventional relational storage subsystem, namely that of HP-SQL [HP]. HP-SQL's Storage Manager is very similar to System R's RSS [Blasgen,77]. It was extended to support the generation of unique OID's. In general, it operates over a single table per request. Joins and aggregate operations are done outside the Storage Manager.

Tables can be created and dropped at any time. The system supports transactions with savepoints and restores to savepoints, concurrency control, logging and recovery, archiving, indexing and buffer management. It provides tuple-at-a-time processing with commands to retrieve, update, insert and delete tuples. Indexes allow users to access the tuples of a table in a predefined order. Additionally, a predicate over column values can be defined to qualify tuples during retrieval. The Storage Manager also provides a scan operation which permits associative access to a table and returns multiple tuples per request.

3.8 Example: Type Creation

To provide a better understanding of the Iris architecture, we describe the Kernel processing in creating a new type. The creation of a new type is the task of the system procedure, `TypeCreate`, which is implemented as an OM foreign function. To invoke this procedure, a request expression is built containing a single call node with a function identifier of "TypeCreate" and an argument list of two elements: the name of the new type and its super-types. The super-type argument is itself a list containing the names or object identifiers of the super-types of the new type.

The request expression is then passed to an Iris entry point in the Executive. The Executive immediately passes the request expression to the Query Translator for compilation. QT checks the argument types, retrieves the R-tree implementation for "TypeCreate" and substitutes the actual arguments for the formal arguments.

Since this request was a simple function call with constant arguments, QT uses its fastpath and no further optimization of the R-tree is required. In this case, the R-tree is a single foreign function node that calls the OM foreign function that implements `TypeCreate`.

QT returns the R-tree to the Executive which then passes the tree to the Query Interpreter to produce the result values. QI simply invokes the foreign function and returns the result that is the object identifier of the newly created type. Thus, the real work of type creation is done in the foreign function.

The `TypeCreate` foreign function performs the following actions. First, it checks that the type name is unique and that the supertypes exist. Then, it updates the system functions for the type metadata. Finally, it creates a *typing function* for the new type that maintains the extension of the type.

Function creation is performed by the Iris system procedure, `FunctionCreate`, that is invoked through a recursive Kernel call. If the function is successfully created, the `TypeCreate` foreign function returns the object identifier of the new type object. Otherwise, the type object is deleted and an error added to the error buffer.

4 Implementation and Usage Experience

In this section, we describe our experiences in implementing and using Iris. We concentrate on what we view are the most novel aspects of the system.

4.1 Data Model

A major advantage of the Iris data model is that it provides a good separation among the concepts of object, type and function. This is reflected in three ways. First, objects may acquire and lose types, dynamically. Second, functions over a set of types may be created and destroyed at any time. Third, objects of a given type are not required to participate in every function defined on that type.

An important consequence of this orthogonality is that an instance of an Iris database, including its schema, may evolve without affecting existing applications. This has wide appeal among Iris users. For example, the addition of a new function on an existing type is transparent to objects of that type and to other functions on that type. Similarly, deleting a type only affects applications that use functions defined on the type.

As another example, an object retains its identity across type changes. Changing a person object to a mouse object does not require the object to be deleted and reinserted in the database. References to that object are still valid after the type change.[6]

Another novel feature of Iris is the use of functions to unify the notions of attribute, relationship and operation. This makes the data model conceptually simpler and simplifies the Kernel by reducing the number of constructs that must be implemented. In addition, by separating the declaration of a function from its implementation, Iris provides data independence.

For example, a function that returns the age of a person might change from a stored implementation to a derived implementation, e.g. based on birthdate, without affecting any application programs. This is another feature that appeals to Iris users since, again, it permits schema evolution. Of course, the same effect could, optionally, be achieved in other systems by defining views.

[6]However, references to the object as a person are automatically removed as part of the type change.

However, in Iris, data independence exists for all functions as a consequence of the model. As an aside, we note that it is not always true that applications are immune to changes in the implementation of a function. This is because different function implementations have subtly different semantics. For example, all stored functions may be updated whereas no foreign functions can be updated. Thus, an application that updates the age function might be broken by the previously mentioned implementation change.

Finally, we note that modeling the Iris metadata in terms of the Iris data model was positively received by users. Since there is a common language for accessing user and system data, the writing of user interfaces is simplified because there is no need to special-case metadata access. Of course, many database systems have this feature but it was important to retain it in the Iris model.

The principal extension we plan to make to the Iris data model is to add support for complex objects. However, complex objects are an interesting problem for the Iris model because, unlike other object systems, Iris objects have no explicit state; they only have function values. Thus, an Iris complex object must be identified by a subset of all the functions that are defined on a particular object type. The major issue, then, is how to identify the functions that comprise the complex object.

Another problem with the current implementation is the lack of orthogonality for bags and tuples. For example, tuples may not contain bags or tuples as elements and bags may not contain bags as elements. This is related to the complex object problem and we expect that solutions there will apply to this problem.

4.2 Interfaces

Several groups within HP are experimenting with Iris. In general, their experiences have been quite positive. The OSQL interface has been a valuable tool in introducing users to Iris. It is a fairly natural interface for those who have been exposed to SQL. Typically, a new Iris user will experiment with an OSQL schema and then browse the database using the Graphical Editor to see what was created. The ability to display the type hierarchy and functions associated with each type was deemed especially helpful.

A good test of the usability of Iris and OSQL is how easily they can be taught to new users. We noted an interesting learning curve for Iris novices. First-time users seem to have some initial trouble accepting the model. We speculate that this is due to previous experience in which the notions of data and type were combined (as in many programming languages and traditional database systems). Then, things seem to click and new users are rapidly able to develop schemas and applications.

But later, things become more difficult. We believe this is due to the flexibility of the model. Users get confused as they become aware of more modeling choices. Given that objects may gain and lose types, novice data modelers are, sometimes, unsure whether to model something as a type or a function. For example, rather than use a function to return the sex of a person, one could define two types, male and female, and make all persons instances of one of those types. However, modeling problems occur in developing schemas in any database system. Once a certain level of expertise is reached with the primitives of the data model, the real difficulty shifts from the model to identifying the object types and relationships in the application.

For new users, we found that OSQL simplified the use of Iris in two ways. First, OSQL statements can implement higher level operations than those provided by the Iris system functions and procedures. For example, OSQL provides a single statement that bundles together the creation of a type and several functions on that type. This was easier for new users to comprehend and more convenient, in general, than writing separate statements. Second, OSQL can reduce the complexity in using Iris by hiding some of the nuances of the data model and limiting the number of options available to users.

The CLI and Kernel interfaces were not used by most novice and casual users. These interfaces seemed relatively hard for new users, perhaps because they expose too many choices of the data model. Embedded OSQL was the preferred programmatic interface. It is interesting to note that some users did use CLI as a base layer to define their own Iris interface, in effect, implementing their own data model.

We also received feedback from more experienced OSQL users who developed relatively large OSQL applications. One group reported that, for their application, the OSQL schema was one third smaller than the equivalent SQL schema. They felt that the primary reason for this reduction was function inheritance. Inheritance reduced the amount of redundancy that was present in the relational schema, e.g. a function could be inherited rather than repeating a foreign key in a table. Also, the OSQL schema and queries were easier to read than their relational equivalents because OSQL was able to hide some join expressions. For example, some joins that, in a relational system would be stated explicitly, could be specified in Iris through function composition and through the type hierarchy.

4.3 Usage and Performance

To illustrate the functional extensibility of Iris, we wrote a transitive closure function, `tc`. The following OSQL statement defines `tc` as an Iris foreign function whose implementation is to be found in the file `om_tc.o`.

```
create function tc( Function f, Object root, Integer maxDepth ) ->
      <Object obj, Integer depth> as link om_tc;
```

The implementation of the function required about 200 lines of C code. The `tc` function takes as its first argument another function that is required to be a unary function with the same argument and result type. Starting at the root object it returns all lists $< obj, depth >$ where depth is the smallest integer less than maxDepth such that $obj = f^{depth}(root)$. The data may be cyclic and duplicates are eliminated. A maxDepth of -1 returns all reachable objects and their shortest distance from the root.

The following OSQL schema uses `tc` to implement the All_subParts function which retrieves all sub-parts of a given part.

```
create function IsPartOf( Object o ) -> Object p as stored;
create function All_subParts( Object p ) -> Object sp as
      select sp for each Object sp, Function f, Integer d
      where tc(f,p, -1) = <sp, d> and NameOfFunction(f) = ''IsPartOf'';
```

We note that the return type of these functions is declared as Object rather than Part as might be expected. This is necessary to satisfy the Iris type checker because function `tc` is declared with a return type of Object. Since the type checker is invoked at compile time and it does not support late binding (see Section 4.4), the type checker only knows that objects returned by `tc` have the declared return type, in this case, Object. Thus, a useful new feature in the type checker would be support for type coercion.

At this time, there is no commonly accepted benchmark for object oriented database systems. Relational benchmarks such as the Wisconsin benchmark [Bitton,83] are unsuitable for evaluating object oriented database systems. For example, Iris typically clusters functions on the object identifier for a type. Therefore, it is not possible to build clustered indexes on user-supplied primary keys as is done in relational systems.

However, since the Wisconsin benchmark is well known we ported the benchmark to OSQL. Response time was acceptable given the lack of tuning and limited optimization in the current prototype. For example, the benchmark query that performs a selection over a non-clustered index with a selectivity factor of 10% on a 10,000 tuple relation had an average elapsed wall clock time of 7.36 seconds over 100 executions on an HP 9000/350 (a 68020-CPU machine).

4.4 Rule-based Query Translator

As discussed in Section 3.3, the Iris Query Translator contains a rule-based optimizer. The promise of rules and rule-sets is that the query optimizer is easy to maintain and simple to modify because the rules can be manipulated independently. In our experience, that was generally, true, i.e. when the rules in a rule set were independent, adding or modifying rules was relatively simple. For example, it was easy to modify a rule set containing rules for simplifying algebraic expressions (e.g. constant propagation and folding). In this rule set, most rules could be fired independently of other rules.

However, most rule sets were difficult to modify because of the inter-dependencies among the rules. In addition, the rules within a rule set can be applied in almost any order so it is hard to understand the effect of their inter-dependencies. In practice, it was often easier to add a new rule set rather than modify an existing rule set. This is because the dependencies among the rule sets were well understood since the rule sets were applied sequentially.

A second difficulty in modifying the rule base occurred whenever a new operator or data node type was added to the set of R-tree nodes. Since Iris is an evolving system, the operators and data types of the R-tree occasionally change. However, many of the rules were written with case statements based on the node type. Thus, adding or removing an operator or leaf node required modifying the case statements in all such rules. This was not difficult but it was tedious.

The performance of the optimizer was adequate for our purposes. Clearly, it would have run faster written as straight-line code. But, the flexibility was much more important than incrementally better performance. Notice, we do not claim that the operation of rule-based optimizers is easy to understand nor that they are easy to debug. For debugging, the usual tactic was to trace the flow of the optimizer through each rule and rule set.

It is interesting to note that most of the rules were concerned with algebraic and topological transformations of the tree, e.g. removing unnecessary projections, pushing down projects and filters. The logic for doing the real work of optimization, such as choosing alternative access paths or ordering the operands of a cross-product, was concentrated in a small number of rules. Thus, we were not able to take full advantage of the rule system to experiment with alternative optimization strategies. Typically what we did was, in effect, replace one optimization algorithm encapsulated in a rule by another rule that encapsulated a different algorithm.

But overall, the use of a rule-based optimizer was a good choice for Iris. It was flexible enough to support the addition of new operators and nodes which was a primary consideration in our prototype system. It was disappointing to discover that, as with conventional optimizers, modification of a rule-based optimizer does require a guru.

Finally, we note that the current version of Iris does not support late binding for functions and object types. On the one hand, this is one of the strengths of Iris since it means we can generate pipelined relational algebra expressions for our execution trees. Thus, with tuning and an equivalent optimizer, the performance of our execution engine should be as good as a relational system. However, there are situations where late binding is necessary. Thus, a future area of investigation is how to support late binding while retaining the efficient pipelined execution tree.

4.5 Foreign Functions

Foreign functions have been very successful in extending the capabilities of Iris beyond that offered by its interface. They have been used to implement the arithmetic operators, to call SQL and other database systems remotely and to implement aggregate operators. They are also used in the Object Manager to implement the procedures of the data model, e.g. type and function creation, object deletion. Foreign functions permit Iris to serve as an integrator of data and applications by providing access to external services, applications and databases. Users see this as a major benefit.

However, there are a few problems with the current implementation of foreign functions. First, they are a security risk. Second, they are difficult to debug. Third, to the naive or casual user, they may seem difficult to write. Foreign functions pose a security problem because they execute without protection in the same address space as the Iris Kernel. There is no good solution to this problem without operating system or hardware support.

The debugging problem exists because the symbolic debugger cannot be used on code that is dynamically loaded. We have experimented with executing foreign functions in a separate process. This permits use of the symbolic debugger and also solves the security problem. This is adequate for development, testing and integration. However, it is not a general solution due to the overhead of the process creation and remote procedure calls.

The difficulty in writing foreign functions is a consequence of the writer having to adhere to the interface conventions and being exposed to Iris internal data structures. These problems can be partially alleviated through the use of utility subroutines to hide the internal details and the ability to use the embedded OSQL or CLI interfaces in foreign functions.

4.6 Recursive Kernel Calls

The ability of the Iris Kernel to call itself is an unusual and powerful aspect of the implementation. This feature is heavily used in both the Query Translator and Object Manager modules. It also provides a degree of *function independence* in that we may change the implementation of system procedures and functions without affecting the callers. Thus, the change of a system procedure, such as `TypeCreate`, from a foreign function to a derived function would be transparent to the rest of the Kernel.

There are two questions raised by this recursive architecture. The first issue is performance. This is addressed in the next section. The second issue is the bootstrapping problem, i.e. if Iris is implemented in terms of itself, how does it get started? Rather than describe the solution in detail, suffice it to say that certain system functions are treated as special cases by the Query Translator and the recursive call is avoided. For example, the system function that retrieves the R-tree implementation for a function must be special-cased to avoid infinite recursion. The number of these special cases is on the order of a dozen and is not expected to grow significantly. So, we do not feel it is an indictment of the architecture.

4.7 Iris Cache

The performance of the current architecture is very dependent on its cache. In this architecture, metadata is accessed by invoking system functions through recursive Kernel calls. Typically, the system function is implemented as a table node that accesses one of the stored tables containing the metadata. Most system tables are cached by CM which prevents a (high overhead) Storage Manager call. This differs from the first Iris prototype which contained a special-purpose metadata cache. That cache was accessed through direct subroutine calls.

At first glance, it appears we have accomplished little except to significantly increase the path length for metadata access. However, the special-purpose cache in the first prototype was limited in that it could not cache many-valued system functions. More importantly, the implementation of that cache was tailored to the structure of the metadata. Thus, any change to the structure of the metadata required rewriting portions of the cache code. Finally, there were two paths to the metadata, one for the Kernel and another for clients. This complicated the coding of system procedures and meant that client requests could not take advantage of the cache.

The cache in the new architecture is general-purpose and makes cache access transparent to the execution tree. Thus, although the path length is longer for compile-time metadata access, we have a net gain in performance because the cache has wider applicability. In addition, it is much easier to modify the metadata structure (e.g. to add new columns or tables) since the cache code is unaffected. And, in a sense, the compilation time for a request is not a critical factor so long as it is within reason. Execution time is much more important. We expect the general-purpose cache to be a benefit there since it is accessible at run time for both user and system functions. For example, we can cache the inner tables of nested-loops joins.

Finally, note that retrieval performance could be significantly improved if Iris cached *function* values rather than *tables* since a function call could be directly evaluated without compiling it into a relational algebra tree. However, updates to stored tables pose a challenge since a single tuple in a stored table may contain values for many functions (e.g. when functions are clustered together in one table; see [Lyngbaek,87]). The individual function caches would need to be located and updated in an efficient manner. This problem is currently under investigation.

4.8 Storage Manager

An issue in the early implementation phases of Iris was whether to use a low level or high level storage manager. We define a low level storage manager as an RSS-like layer, i.e. one providing intra-table operations, and a high level storage manager as an RDS-like layer, i.e. one providing inter-table operations, such as joins. The decision to go with the low level storage manager was for performance reasons. In this way, Iris gained more control over storage structures was able to implement its own optimization strategies. It was also able to take advantage of storage manager features that might be hidden by higher layers, such as tuple identifiers and links between tuples. In addition, it meant there shorter pathlengths and fewer translations between Iris and the stored database.

The performance of Iris could be improved through better integration with the Storage Manager. Iris uses an off-the-shelf Storage Manager that was not specifically tailored to Iris. This results in duplicated services such as caching and forces translations between the two different data storage formats. A related problem is that Iris metadata receives no special treatment from the Storage Manager. Since metadata is often a database hotspot, concurrency control problems may occur when one Iris client updates the metadata. This effectively locks the metadata and other clients are forced to wait for the updater to finish.

Iris would benefit from some added functionality in the Storage Manager. For example, the Storage Manager does not support the complete set of Iris base data types. It provides little control over data placement (e.g. for locality of reference) and only B-tree access methods are supported. Also, the Storage Manager only supports flat (first-normal form) tuples and there is a limit of 4 kilobytes on the tuple size.

However, use of the Storage Manager was an excellent choice for the Iris prototype. The performance was adequate, in general, and it allowed us to concentrate our efforts on the data model and interfaces. In fact, rather than investigate tighter coupling with the Storage Manager, a research issue in Iris is Storage Manager independence. We would like to define an abstract model of a storage manager interface and then efficiently map that model onto different real storage managers. The value of this is twofold. First, it simplifies the porting of Iris onto different storage subsystems. Second, and more importantly, it facilitates remote database access in that Iris would be able to use two different storage managers, simultaneously. An initial step along the lines of an abstract storage manager interface is provided by the Cache Manager interface.

4.9 Statistics and Development Environment

The Iris system is a research effort involving approximately twelve people over a period of four years. Roughly two thirds of the effort was on the Iris data model and Kernel, the remaining third was on the Iris interfaces. Two prototypes have been built, both in C. The second (and current) prototype was built to increase the flexibility of the Iris Kernel and to permit recursive calls.

The Iris Kernel (excluding the Storage Manager) consists of approximately 85K lines of C code. The operating system, HP-UX, is HP's version of Unix System V with extensions from Berkeley Unix. Iris runs on two hardware platforms: the HP Series 300 workstation (Motorola 68000 CPU) and the HP Series 800 workstation (HP Precision Architecture - RISC).

To coordinate the activities of the many people working on Iris, RCS [Tichy,82] was used to record the change history for individual source files. A validation test suite was automatically executed twice daily on the most recently checked-in version of Iris. This automated testing system worked very well. In fact, it worked almost too well in that we tended to rely on it as our correctness criterion. Thus, a broken feature might go undetected for weeks if the feature was not tested by the validation suite.

5 Conclusions and Future Work

The Iris System facilitates the prototyping of new model semantics and functionalities. We expect a lot of experimentation in such areas as authorization, complex objects, versioning, overloading, late binding, monitors and triggers. In addition, two main research directions have been identified. One is to extend and generalize the Iris model and language to allow most application code to be written in OSQL. That way a large amount of code sharing and reuse can be obtained. Some of the interesting research topics include optimization of database programs with side-effects, optimization of database requests against multiple, possibly different storage managers, and using the language to support declarative integrity constraints.

The other research direction is to increase the functionality and power of the Iris client interface, i.e. the parts of Iris running on a client machine. In order to better utilize local resources, the Iris interpreter must dynamically decide whether to interpret a given request locally (using locally cached data and possibly extending the cache in the process), send the entire request to the server for interpretation, or possibly split the request into several parts some of which are evaluated locally and others remotely. Some of the interesting research topics include copy management (for example by using database monitors [Risch,89]), checkin/checkout mechanisms, and data clustering techniques.

Acknowledgements

Marie-Anne Neimat designed and implemented the general-purpose cache manager. Jurgen Annevelink and Jim Davis were instrumental in converting Iris to the new architecture. Ming-Chien Shan and Tim Connors also helped in the conversion.

Many members of the Database Technology Department at HP Laboratories contributed to and influenced the Iris project: Jurgen Annevelink, Tim Connors, Jim Davis, Dan Fishman, Charles Hoch, Bill Kent, Marie-Anne Neimat, Tore Risch, and Ming-Chien Shan. Nigel Derrett, Tom Ryan, David Beech and Brom Mahbod also made substantial contributions to the first Iris prototype.

The authors wish to thank Jurgen Annevelink, Dan Fishman, Stefan Gower, Marie-Anne Neimat, Emmanuel Onuegbe, Katie Rotzell and the reviewers for helpful comments on this paper.

References

[Blasgen,77] M. W. Blasgen and K. P. Eswaran. Storage and Access in Relation Databases. *IBM Systems Journal*, 16(4):363–377, 1977.

[Beech,88] D. Beech and B. Mahbod. Generalized Version Control in an Object-Oriented Database. In *Proceedings of IEEE Data Engineering Conference*, February 1988.

[Bitton,83] D. Bitton, D. J. DeWitt and C. Turbyfill. Benchmarking Database Systems - A Systematic Approach. In *Proceedings of the 1983 VLDB Conference*, October 1983.

[Buneman,82] P. Buneman, R. E. Frankel and R. Nikhil. An Implementation Technique for Database Query Languages *ACM Transactions on Database Systems*, 7(2), June 1982.

[Connors,88] T. Connors and P. Lyngbaek. Providing Uninform Access to Heterogeneous Information Bases. In Klaus Dittrich, editor, *Lecture Notes in Computer Science 334, Advances in Object-Oriented Database Systems*. Springer-Verlag, September 1988.

[Derrett,89] N. Derrett and M. C. Shan. Rule-Based Query Optimization in Iris. In *Proceedings of ACM Annual Computer Science Conference*, Louisville, Kentucky, February 1989.

[Fishman,87] D. H. Fishman et al. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[Fishman,89] D. H. Fishman et al. Overview of the Iris DBMS. In W. Kim, F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* ACM Press, New York, N.Y., 1989.

[Graefe,87] G. Graefe and D. J. Dewitt. The EXODUS Optimizer Generator. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 160–172, 1987.

[HP] Hewlett-Packard Company. *HP-SQL Reference Manual.* Part Number 36217-90001.

[Heiler,88] S. Heiler and S. Zdonik. Views, Data Abstraction, and Inheritance in the FUGUE Data Model. In Klaus Dittrich, editor, *Lecture Notes in Computer Science 334, Advances in Object-Oriented Database Systems.* Springer-Verlag, September 1988.

[Lyngbaek,86] P. Lyngbaek and W. Kent. A Data Modeling Methodology for the Design and Implementation of Information Systems. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[Lyngbaek,87] P. Lyngbaek and V. Vianu. Mapping a Semantic Data Model to the Relational Model. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, San Francisco, California, May 1987.

[Mylopoulos,80] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2), June 1980.

[Manola,86] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[Risch,89] T. Risch. Monitoring Database Objects. In *Proceedings of the 1989 VLDB Conference*, Amsterdam, The Netherlands, August 1989.

[Shipman,81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), September 1981.

[Tichy,82] W. F. Tichy. Revision Control System. In *Proceedings of the IEEE 6th International Conference on Software Engineering.*