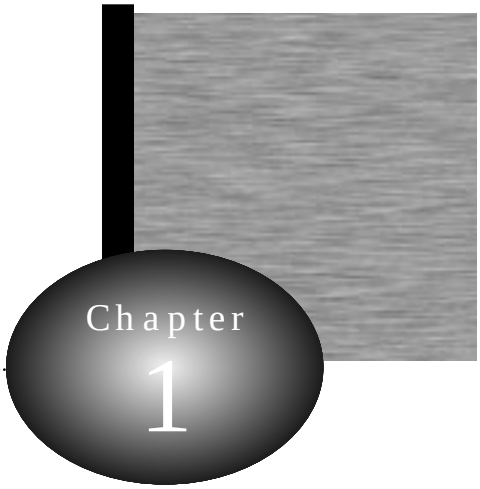

Architectural Options for Object-Relational DBMSs

by Michael R. Stonebraker

This paper describes and evaluates the architectural options for evolving a relational database management system (RDBMS) towards the object-relational database management system (ORDBMS) paradigm. The goal of this evolution is to produce a single system possessing the benefits of both relational and object-oriented technology, namely:

- The functionality of a dynamic programming language like Structured Query Language (SQL) that understands objects—user-defined types and functions—in addition to simple datatypes (integers, characters, and dates).
- The scaleability and performance that is the result of two decades of research and development by RDBMS engine vendors and the academic community.

We explore this topic first by describing the technical alternatives database vendors have for achieving this goal, and second by examining how well these alternatives perform according to both of the criteria introduced above. Finally, we survey the

commercial database (DBMS) marketplace, and show how each vendor is managing the change.

Introduction

Modern business is making increasing demands on data processing technology. First, this is because the environment within which modern industry operates is becoming increasingly complex. Today, managers need to incorporate geographic considerations into marketing, risk analysis, and supply chain management. They need to access textual reports based on the concepts that the reports discuss. They need to include digital photographs, audio data, and digitizations of items like checks and signatures. As object-oriented programming languages penetrate the mainstream, they need a DBMS technology with a philosophy closer to their client tools.

And there is no shortage of this kind of information. All of the datatypes we have described above are significantly larger than the types MIS is familiar with. One characteristic of this kind of data is that it is machine-generated. As fast as a user can type, electronic hardware can scan pages and perform optical character recognition thousands of times faster.

Yet even with all of these increasing demands there is unlikely to be a diminution of user expectations with respect to performance. This situation requires a DBMS capable of storing objects (and not simply as records) combined with support for new access methods, query processing algorithms, and storage management techniques with no degradation in the DBMS capability for traditional alphanumeric data processing.

One option is a pure object-oriented database management system (OODBMS) approach that allows developers to store and retrieve C++ and Smalltalk objects efficiently. However, OODBMSs do not, at this date, excel at processing relational fixed-form data. Conversely, a purely relational database does not make modeling complex situations easy. Any complexity cannot be handled within the system: it must be given special treatment externally. An ideal solution is one that combines the power of the object-

oriented approach with the data management flexibility of the RDBMS.

The object-relational approach does this, as adopted by Informix in Informix Dynamic Server™ with Universal Data Option™. Universal Data Option merges the best of both worlds by combining the unmatched scaleability and performance of the Informix Dynamic Scaleable Architecture (DSA) with the object-relational features. This creates a single, integrated framework enabling businesses to manage all types of information efficiently: numbers and character strings, GIS data values, images from document scans and text data from OCRing these images, digital photographs, audio, and Web pages. Indeed, any datatype.

The achievement of building Universal Data Option is quite remarkable. The relational DBMS must be drastically altered to include object-relational (OR) functionality: user-defined types and functions, inheritance, and polymorphism. Certain relational DBMS vendors are either unable or unwilling to perform the required surgery, and instead have elected to pursue inferior strategies for achieving OR functionality.

This paper assesses the architectural options available to various vendors along two crucial dimensions:

1. Functionality—The desired functionality for an object-relational DBMS includes support for base type extension, complex objects, and inheritance within the SQL query language. Some vendors, such as Oracle 8.0 and the Oracle Networking Computing Architecture (NCA), do not remotely meet this test, and should not be called object-relational DBMSs. Others, such as the IBM DB2/6000 Common Server, come closer. Hence, there are a spectrum of capabilities offered in the marketplace.
2. Performance—There are two crucial decisions to make with respect to technical architecture for an ORDBMS. First, is the server extensibility achieved by *tight integration* of user-written code within the DBMS framework (for sorting, indexing and query processing), or *loose integration* which is achieved by creating a layer of wrapper logic that hosts the extension. Second, the server can call the extensions without needing to ask the operating system to intervene in the operation *tight*

coupling or employ a heavyweight mechanism like the remote procedure call (RPC) *loose coupling.* As we shall see, the performance implications of anything except server extensibility achieved by the tight integration of user code into the DBMS leads with a tightly coupled calling mechanism to disastrous performance.

After a detailed discussion of these two dimensions, the characteristics of the solutions proposed or implemented by the various database management system (DBMS) vendors will be discussed.

Functionality of an ORDBMS

To illustrate the required functionality of an ORDBMS, the following example table will be used.

```
create table Emp
    (name = Scottish_name,
    age = int,
    location = point,
    picture = image);
```

Here, *name* and *age* have the obvious interpretation. However, two additional fields have been added to a traditional Emp table: the location of the employee's home address, and the employee's picture. These fields are of the datatypes "point" and "image," respectively.

Following is a typical query to this table:

```
select name
  from Emp
where name < 'b'
  and age > 50
  and contains(location, circle ('(10,10)',1.5))
  and beard(picture) > 0.7
order by name ascending;
```

This query locates all employees whose name begins with an 'a', whose age is above 50, who live within a five-mile circle surrounding the point "10,10", and who also have beards.

This query, although simple and straightforward to the average reader, is impossible in a traditional relational database management system (RDBMS). First, in many countries outside the United States, names do not sort in strict ASCII order. In such places, the normal "collation sequence" of letters is complicated by case-sensitive rules. Scotland is one such example. In the Edinburgh phone book, the following names—all variants of the clan McTavish—are sorted together:

- McTavish
- MacTavish
- M'Tavish

As a result, one cannot use the varchar datatype found in SQL-89 because it supports a '<' operator which collates in ASCII order, and thereby generates an incorrect answer for many Scottish queries. In addition, one cannot perform any single character substitution algorithm, such as supported in SQL-92, since {a, c,'} are only "silent" if preceded by an 'M'. Instead, to perform the above query requires a new datatype, Scottish_name, with a type-specific notion of '<'.

Moreover, when the user requests the result of the query in ascending order, the system must generate the answer in Scottish_name order. Hence, sorting must be accomplished using the user-defined notion of '<'.

In addition, the third clause contains a pair of user-defined functions (*contains*, *location*) and a nonstandard datatype (*circle*), while the fourth clause contains a user-defined pattern recognition function (*beard*) that looks at an image and determines whether the person in the image has a beard. The user-defined functions (*beard*, *location*, and *contains*) must be writeable in SQL, a stored procedure language such as Informix's SPL, or in a third-generation language, such as C or Java. Since it is impossible to write the *beard* function in a stored procedure language, an object-relational DBMS must support coding functions in all of the three languages noted above.

As a result, this article defines basic object-relational functionality as the ability to construct the above table, register the

above functions written in one of the three languages, and execute the query noted above.

Systems which provide this functionality include Informix Dynamic Server with Universal Data Option, and are termed fully object-relational. In contrast, other systems, such as Oracle 8.0 and Oracle NCA, have essentially no object-relational functionality. Finally, other systems, such as IBM's DB2/6000 Common Server, have some object-relational functionality but do not pass the functionality test.

As a result, systems will be classified along a functionality axis from no object-relational functionality to full object-relational functionality. In addition, Informix is collaborating on a benchmark with the University of Wisconsin which will more precisely and completely categorize object-relational functionality in the near future.

Performance

There are two critical functions that an object-relational DBMS must perform in order to offer high performance in an object-relational environmen; namely, it must tightly integrate object functionality into its engine and it must efficiently call user-written functions. These aspects will be discussed in turn.

Tight Integration

There are two options in the support of object-relational functionality. First, one can tightly integrate the functionality into an existing SQL engine. Alternately, one can implement functionality in a simulation layer above an unchanged core engine, as noted in Fig. 1–1.

To achieve high performance, a vendor must utilize the first approach, that of tightly integrating objects with its access methods, sort engine, and optimizer. The performance difference between the two choices will be explored using aspects of the previous sample query.

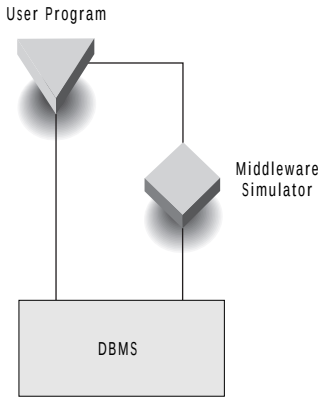

Figure 1–1 *The architecture of middleware simulators.*

Consider the clause:

```
where name < 'b'
```

which finds Scottish_names which begin with an 'a'. To provide high performance on this clause, an engine must use a b-tree index to identify the required names. Hence, b-tree code must be extended to support the indexing of new datatypes, such as Scottish_name, and must keep the indexed objects in sort order, according to a type-specific notion of '<'. In other words, the b-tree code must understand objects.

If a simulation layer is used, then the b-tree code will not understand objects and cannot be used to process the above clause; instead, a sequential scan will be required. The performance difference between an indexed scan and a sequential scan is typically one or two orders of magnitude.

Now consider the following clause:

```
where contains(location, circle ('(10,10)',1.5))
```

This is a two-dimensional search that cannot be optimized using a one-dimensional access method such as a b-tree. To perform such searches requires access methods particularized to geographic data, such as r-trees, quad trees, or grid files. As a result, an object-relational engine must be capable of supporting

additional access methods, including those written by third parties. If a simulation methodology is utilized, then there will be no fast access paths for nontraditional datatypes, and performance will gravely suffer.

Next, consider the following clause:

```
where beard (picture) > 0.7
```

It makes no sense for a DBMS to build a b-tree index on the bits in an image, because there is no inherent order that can be exploited by such an index. Instead, the above clause can be accelerated if an index is built on the result of the user-defined function, *beard*. As a result, a high-performance object-relational system must allow indexes to be defined on a *function of an attribute*. Again, a simulation layer cannot accelerate clauses of this kind.

Now, consider the following clause:

```
order by name ascending
```

If objects are tightly integrated into the sort engine in the DBMS, then the qualifying names can be efficiently sorted by the DBMS sort engine. Alternately, the sort must be performed in the simulation layer outside the core engine. The consequence of moving the sort farther from data storage is a severe performance hit.

Lastly, consider the following pair of clauses:

```
where name < 'b'
and contains(location, circle ('(10,10)',1.5))
```

In the above example, the optimizer must decide whether to use a b-tree index on the first clause or an r-tree index on the second clause to solve the query. If the selectivity of the first clause is much finer than the selectivity of the second, then the b-tree approach should be selected. Otherwise, the r-tree choice should be executed. In addition, in certain circumstances it is desirable to use both access paths. If objects are tightly integrated into the optimizer, then these decisions can be easily made. Without tight integration, incorrect query execution plans are likely, and disastrous performance can result.

These examples have all indicated the severe performance consequences of *loose integration*, that is, implementing objects in a simulation layer, as compared to *tight integration*, implementing objects in the core engine. The performance degradation often approaches orders of magnitude in size.

Degree of Coupling

Whenever a user-defined function is used in a SQL query, the vendor must support one or more protocols to execute a call to the user-written routine. The choice of the calling protocol is motivated by performance and safety considerations.

In the previous example user query, there is an operator '<' which ascertains if a given Scottish_name collates before the letter 'b'. The function which supports '<' may have an execution path length of 100 usecs and is called once for each record evaluated in the query. As such, the performance of the call is a concern. If the overhead of the call is significant relative to 100 usecs, then overall query performance is adversely affected.

In addition, one must be concerned about the effect of bugs in user-written code and the damage that bugs can cause.

Table 1–1 indicates the performance of the various options in calling a user-defined routine.

One can call a user-defined procedure in the same address space as the DBMS, thereby obtaining the highest performance. The second option is to support the *fenced* execution of local procedures, whereby the DBMS is protected from errors in the user-written routine. Of course, the performance tax of this choice is dramatic, as noted in Table 1–1. A third option is to use the standard RPC system found in essentially all operating systems.

Table 1-1 *Calling characteristics.*

Calling protocol	Typical call overhead (usecs)
Local Procedure Call (LPC)	~1
Fenced LPC	~75
Remote Procedure Call (RPC)	~1000
CORBA	~10,000

However, the tax for this feature is another big increase in overhead. The last recourse is to use a protocol such as CORBA, that includes an Object Request Broker (ORB) which can locate the required user routine is a distributed computing system. Again, performance degrades by another order of magnitude.

Most extension systems, for example, Novell NLMs, CICS applications, VB controls, and Netscape plug-ins, offer only LPC, because the performance of the other options is unacceptably bad. Moreover, essentially all existing Illustra customers use LPC, although RPC is also available. Again, the motivating issue is achieving good performance. In short, the tight coupling of user routines is required to achieve good performance. Of course, loose coupling is desirable in certain circumstances, and should be provided in any complete system.

This article now focuses on three points about the safety of user-written functions. First, a bug in any program causes the address space in which it runs to fail. The functionality utilized is then unavailable until recovery is accomplished. Any program that is used as a plug-in to any framework will have this characteristic. As a result, good software practice includes a serious quality assurance program, and released code, either from a DBMS vendor or a third-party supplier of extension routines, must be tested thoroughly. The adequate testing of any software routines inserted into a production system is standard practice in the industry.

Second, either fenced execution or RPC is available at a huge performance penalty if a user does not feel confident in their routine.

Third, safety concerns are different for an interpreted language, such as Java, than for a compiled language, such as C or C++. In an interpreted language, the execution system will catch errors, and safety concerns are not an issue.

In this section we have indicated that the tight integration of object-relational capabilities into the underlying engine, as well as tight coupling to user-written routines, is required to achieve good object-relational performance. As such, the products of the various vendors offer performance that ranges from bad to good, depending on their choices in these two areas. The next section of this article discusses a collection of architectures that have been proposed and compares them on the basis of functionality and performance considerations.

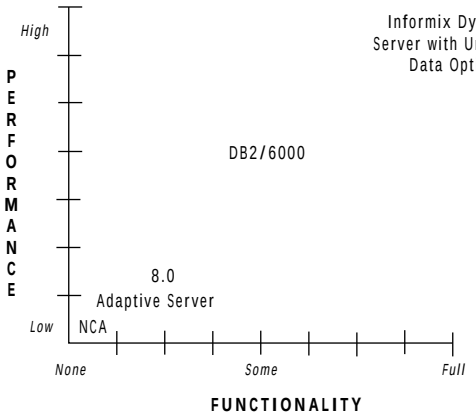

Figure 1-2 *Functionality versus performance for servers.*

Marketplace Architectures

In Fig. 1–2, five systems, Sybase's Adaptive Server, Oracle NCA, Oracle 8.0, IBM's DB/2 version 5, and Informix Dynamic Server with Universal Data Option, and Common Server, are placed on the functionality versus performance chart. Along the horizontal axis is the object-relational functionality provided, ranging from none to full. On the vertical axis is the degree of integration and degree of coupling provided, which determines performance, ranging from low to high.

Sybase Adaptive Server

Sybase were late-comers to the object-relational party. In fact, it is unclear today whether Sybase even intends to produce an ORDBMS at all or instead focus on marketing a middleware and connectivity product. Sybase's Adaptive Server provides a set of interfaces allowing developers to integrate data sources into a single query-able system: a technology superficially similar to Informix's Virtual Table Interface. In fact, Sybase has built a sophisticated wrapper layer around both Sybase's core SQL-92

RDBMS and whatever extensions developers add. The DBMS server itself is almost unchanged.

At the time of writing, users could integrate code written in the Java language or integrate data sources with 'C'. This means that it is not possible to pose our query to the Adaptive Server. Instead, each datatype must be integrated separately, and the code to run functions like `beard()` is hosted outside the DBMS, in the layer of wrapper code.

This makes Sybase a loosely coupled and loosely integrated ORDBMS. So far as their architecture is concerned, this implies mediocre performance (at best) for operations involving any one extension, and disastrous performance when they are combined. For example, in our sample query, each data source must be handled separately and the result of each operation "joined" in a final step. If the proportion of employees living in the specific circle is very high, but the number over fifty is relatively low, the system will still compute the `containd()` function for every emp and join the result set in the middle-ware.

Since so much effort is required to "extend" a relational engine to make it "object aware," some vendors use application middleware approaches to mimic this functionality. This is possible due to many application frameworks, such as the following:

- VB controls;
- Netscape plug-ins;
- Oracle cartridges;
- Novell Network Loadable Modules (NLMs); and
- The class library interface in most 4GLs.

These frameworks support the execution of user-written routines in a non-SQL context. Such frameworks allow one user program to make a function call to a second user program, thereby providing application-to-application connectivity.

However, this architecture has nothing to do with DBMS-type extension because it is not integrated with SQL. For example, Oracle NCA cannot run the example query unless a user writes an application program that implements this functionality. Middleware plug-ins do not constitute a true object-relational DBMS architecture at all; there is no server-managed integrity, no

common query language, no single table view, and numerous other disadvantages.

As such, Oracle NCA is at the far left-hand side of the diagram of Fig. 1–2. In addition, NCA uses CORBA as its connection protocol, the slowest of the options. Hence, it is also at the bottom of the chart. NCA should never be confused with an object-relational DBMS.

Oracle 8.0

Oracle 8 was released in early 1997. According to the analysts, it was primarily a scaleability and performance release that included very few ORDBMS features. Such features, as it did have, were poorly integrated with the rest of the system—indicating that they were hasty, last minute additions to the product plan. Specifically, it contained an extension to the stored procedure facility of Oracle 7.3, whereby a stored procedure can appear in the WHERE clause of an SQL query. This is added functionality relative to Oracle 7.3, where stored procedures can only be executed. The arguments to an Oracle 8.0 stored procedure are the fields in a row in a table.

This capability is not helpful in generating the schema for the example data, which requires points, images, and Scottish_names. In addition, Oracle 8.0 functions can only be coded in PL/SQL. As noted earlier, none of the functions in our example query are easily written in PL/SQL.

In addition, Oracle 8.0 offers a limited form of inheritance, whereby tables can inherit data from other tables and can inherit functions in PL/SQL written for those tables. However, Oracle 8.0 does not support the overloading of function names, *polymorphism*, so it is impossible for a user to redefine the behavior of a subobject.

As such, Oracle 8.0 has minimal object capabilities and does not support user-defined base datatypes or sets, and does not support functions in a third-generation language.

In addition, user-defined functions are not integrated with the access methods, sort engine, or optimizer. Because Oracle 8.0 only supports loose integration, it is also near the lower left-hand corner of Fig. 1– 2.

Oracle 8.1 and 8.2

Oracle marketing talks extensively of the future notion that cartridges in the Oracle NCA can be plugged into either a middleware application layer or the DBMS engine. In fact, cartridges are not part of Oracle 8.0, and they are unlikely to be supported until Oracle 8.1 or even 8.2 (due in 1999).

Since this capability has yet to be designed, it is impossible to place Oracle 8.1 or 8.2 in our diagram, because functionality and degree of integration are not yet known.

IBM DB2/6000 Common Server

IBM DB2/6000 Common Server supports base-type extension and can construct the schema indicated earlier. In addition, it can almost run the example query; unfortunately, it is impossible to have a type-specific notion of '<'. Hence, the following clause:

```
where name < 'b'
```

must be executed in user code. Also, DB2/6000 Common Server does not support inheritance or complex objects. As such, it is to the right of Oracle 8.0, but does not support full functionality.

In addition, the DB2/6000 optimizer is aware of objects, but the access methods and sort engine are not; they work only on SQL-89 datatypes. Furthermore, user-defined functions can be written in C and are called using either LPC or RPC. Hence, the IBM system offers better performance than the Oracle systems, but there is still room for improvement.

Informix Dynamic Server with Universal Data Option

Informix Dynamic Server with Universal Data Option is the result of a code merge of Informix's Dynamic Scalable Architecture (DSA) with the Illustra Server. It preserves the scalability and alphanumeric performance of Informix DSA, while also providing the Illustra data model and query language. As such, it provides user-defined types, user-defined functions, complex objects, and inheritance. Moreover, it allows a user to overload all operators and functions. It is the only system which will

run the example query, and therefore, it is on the right-hand side of Fig. 1–2.

Moreover, it is a single engine with an object-aware optimizer, sort engine, and b-tree package. Lastly, it supports LPC. As such, it is the only system with both tight integration and tight coupling. As a result, it is positioned highest on the vertical axis.

Summary of Architectural Options

This article has discussed the six systems in the previous section. This section briefly reviews the characteristics of these systems.

Sybase Adaptive Server

This system is a loosely coupled and loosely integrated ORDBMS. Its architecture can at best provide mediocre performance for operations involving any single extension, and this performance worsens when extensions are combined.

Oracle NCA

This is not an object-relational engine, but rather an application middleware system. The possibility of inserting “cartridges” in either middleware or the server will not be available until 1998 or 1999. Informix has this capability.

Oracle 8.0

This system can be described as Oracle 7.3 with a limited notion of inheritance and extensions to stored procedures. Basically, it is a scalability release with minimal object capabilities.

Oracle 8.1 and 8.2

Better object support is expected in version 8.1 or 8.2 with the support of cartridges.

IBM DB2/6000 Common Server

This is an engine with support for base-type extension, but not inheritance or complex objects. Object support is tightly integrated into the engine and extensions are tightly coupled, thereby offering good performance.

Informix Dynamic Server with Universal Data Option

This is an engine with support for all object-relational concepts, offering both tight integration and tight coupling. It is considered the best of the current breed of products.