



## Objects End-to-End The ODBMS Advantage

The IT landscape faces a process of continual change, now more so than ever. Business demands of time to market, reduced costs, increased complexity and functionality all drive towards the single goal of being competitive in today's markets. From a technological standpoint these pressures have driven the adoption of object-oriented technologies, principles and approaches throughout the software development lifecycle.

---

# Contents

Contents .....	2
Figures .....	3
Abstract .....	4
What's an Object Database? .....	5
But why can't I use a relational database? .....	6
So How does an object database work? .....	9
Back to basics .....	9
Object database architectures .....	10
Developing an application .....	13
Conclusion .....	18
But what's the impact on my development process? .....	19
Getting the design right .....	19
Understanding access patterns .....	19
Developing transactions .....	20
Conclusion .....	22
And how do I manage a deployed system? .....	23
The role of the ODBA .....	23
Administration Tasks .....	23
Coexistence .....	25
Conclusion .....	25
Conclusion .....	26
About Versant .....	27
Versant Overview .....	27
Versant History, <i>Innovating for Excellence</i> .....	27

---

# Figures

Figure 1 - Client-server architecture of an ODBMS . . . . .	.5
Figure 2 - Simple object model . . . . .	.6
Figure 3 - Relational schema . . . . .	.6
Figure 4 - Mapping an object to the relational database . . . . .	.7
Figure 5 - Creating an object in an object database . . . . .	.8
Figure 6 - Object Identity . . . . .	.9
Figure 7 - Client-side architecture of an object database . . . . .	.10
Figure 8 - Different database architectures . . . . .	.11
Figure 9 - Query processing . . . . .	.12
Figure 10 - UML for example . . . . .	.13
Figure 11 - Employee class . . . . .	.14
Figure 12 - Department class . . . . .	.15
Figure 13 - Creating an object . . . . .	.16
Figure 14 - A simple application . . . . .	.17
Figure 15 - Performing a query . . . . .	.17
Figure 16 - A DAG . . . . .	.21

---

# Abstract

The IT landscape faces a process of continual change, now more so than ever. Business demands of time to market, reduced costs, increased complexity and functionality all drive towards the single goal of being competitive in today's markets. From a technological standpoint these pressures have driven the adoption of object-oriented technologies, principles and approaches throughout the software development lifecycle. New acronyms like CORBA, new languages like Java and new approaches like component-based development are becoming the norm. So why would people turn to 20-year-old technology when it comes to selecting a database?

This article aims to investigate the roles for database technologies, focusing on how an Object Database Management System (ODBMS), like the one provided by Versant, can help an organisation to truly deliver end-to-end object solutions.

---

# What's An Object Database?

Object Databases appeared on the horizon around the mid 1980s. The goal then was to deliver a new breed of database, designed and optimized to store and manipulate objects. This goal was driven by the widespread adoption of object-oriented modelling techniques and languages. Design decisions made at this time differentiated object databases from existing relational databases. Primarily, instead of focusing on a data model (based on a fixed set of types) specified in some abstract, normalised form, the object database focused on the object model as defined within the O-O language.

The primary strength of an object database is its in-built ability to manage arbitrarily complex models (in terms of types) with arbitrarily complex relationships. Managing objects consisting of simple-valued attributes (integers, strings), multi-valued attributes (dynamic arrays of values) and complex structures is fundamental, but it's the ability to handle relationships that is key - not just one-to-one or one-to-many, but relationships that include semantics: like sets (uniqueness); lists (ordering); maps (associative lookup). These relationships may be complex objects in themselves, perhaps containing hashed values for efficient lookup and retrieval.

In the object world transactions involve navigating relationships and performing complex operations thereon. Object databases are optimized for this navigational access and the marshalling of objects between the database server and client. Figure 1 shows a typical ODBMS architecture.

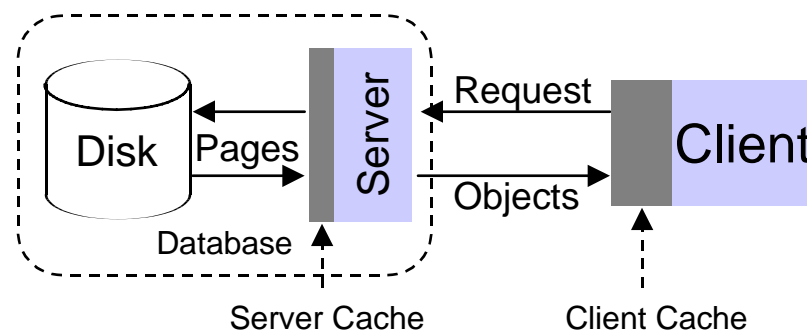


Figure 1 - Client-server architecture of an ODBMS

The server process provides concurrency and transactional control, ensuring recoverability (as with any database). The client-side cache manages the objects that have been transferred from the server, effecting transparent access through the programming language. As relationships are traversed, objects are requested from the server and instantiated in cache automatically. Once in cache they remain there until the transaction ends. When the client commits a transaction, any modified objects are transferred back to the server and the transaction ends.

So to answer the question:

*What is an object database?*

Simply put, it's a database for objects!

# But Why Can't I Use A Relational Database?

Still not convinced? Well, let's look in detail at how you might store objects in a relational database. Figure 2 shows a simple object model that we will use as an example (notice that it doesn't include any real complexity, but it should serve to prove a point).

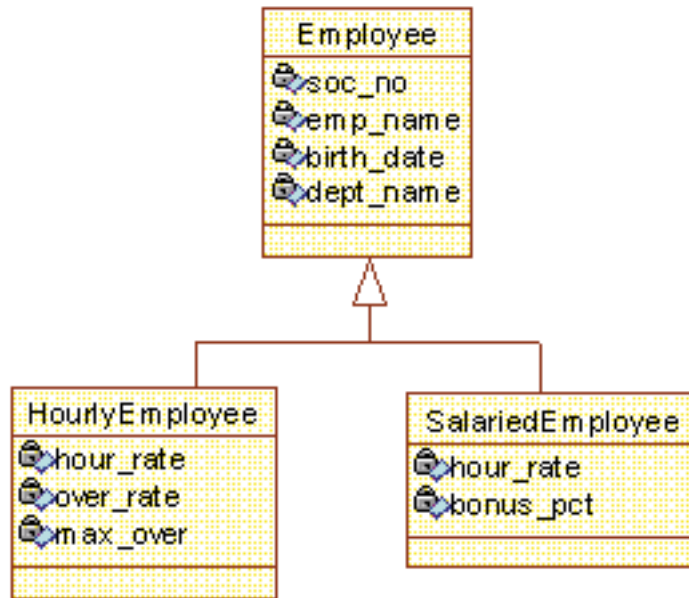


Figure 2 - Simple object model

There are a number of ways in which to map this model to a relational database. Each approach has its trade off. One way is to define a table for each class, where each column in the table represents an attribute of the class (fine for simple-valued attributes, requires more work and additional tables for multi-valued and structured attributes, and don't even think about multi-media types). Figure 3 shows the likely schema.

Employee\_Table

<u>soc_no</u>	emp_name	birth_date	dept_name
---------------	----------	------------	-----------

HourlyEmployee\_Table

<u>soc_no</u>	hour_rate	over_rate	max_over
---------------	-----------	-----------	----------

SalariedEmployee\_Table

<u>soc_no</u>	month_rate	bonus_pct	
---------------	------------	-----------	--

Figure 3 - Relational schema

---

The attributes common to all Employees are stored in the Employee table, additional attributes for HourlyEmployees are stored in the HourlyEmployee table and so on. We may refer to this method of object-to-relational mapping as *Inheritance by Join*. The other approach, analogous to the relational notion of de-normalization, is *Inheritance by Copy*. In this style, the attributes of the parent class are copied into each of its children.

Once the required schema has been defined the mapping code has to be written. This code is required to take an object, as created and manipulated in the programming language and de-construct it into the representation required by the database and conversely construct an object from the database representation so it can be used by the programming language. Figure 4 shows the SQL code required just to perform the mapping from an object to the relational schema (the same will be required to map from the relational schema to an object).

```
EXEC SQL INSERT INTO Employee_Table
(soc_no, emp_name, birth_date, dept_name)
VALUES
(:emp->soc_no, :emp->name,
:emp->birth_date, :emp->dept_name);
if (emp->type == 1)
EXEC SQL INSERT INTO HourlyEmployee_Table
(soc_no, hour_rate, over_rate, max_over)
VALUE
(:emp->soc_no, :emp->hour_rate,
:emp->over_rate, :emp->max_over);
else if (emp->type == 2)
EXEC SQL INSERT INTO SalariedEmployee_Table
(soc_no, month_rate, bonus_pct)
VALUES
(:emp->soc_no,
:emp->month_rate, :emp->bonus_pct)
EXEC SQL COMMIT WORK RELEASE;
```

Figure 4 - Mapping an object to the relational database

---

The Employee attributes are inserted into the Employee table, if the object is type 1 (some means to identify the class of the object), the additional attributes are inserted into the HourlyEmployee table and so on.

Compare this to the code in figure 5, which shows what is required by an object database to create and store an object.

```
HourlyEmployee* employee = new(db, "HourlyEmployee")
HourlyEmployee("Versant", "1/1/88", "Object Technologies");
```

Figure 5 - Creating an object in an object database

A very graphic comparison of the effort involved. You can see why it is generally recognised in the industry that anything up to 35-40% of such an application's code is involved in overcoming this mapping of objects to and from a relational database, (often referred to as overcoming the *impedance mismatch*).

And of course this is only a small part of the overall story:

- Retrieving an HourlyEmployee from the database involves joining the HourlyEmployee and Employee tables. Every time you retrieve an object of any of the child classes (e.g., HourlyEmployee), you will have to join several tables. Had we selected the *Inheritance by Copy* style, the retrieval of HourlyEmployee would avoid the inheritance joins, but that introduces other complications. Using *Inheritance by Copy*, an application must instead join every time it wishes to access objects via a parent class (in this instance, Employee).
- Which of these two approaches is least expensive for a given application depends upon data access patterns and the object model, but even mildly complex objects may require a 5-way join, across potentially large tables (as the number of objects increases).
- None of the above costs take into account mapping complex relationships (sets, lists, and maps); this can take the number of joins into double figures, and also introduce sorting. So even though the objects have been mapped into the relational database it isn't possible to get the required performance as the object model is navigated, (this of course leads to simplification of the object model and its relationships, to reduce the number of joins, thus negating the benefit of using an object approach in the first place).
- Once the mapping layer has been implemented it has to be maintained. More importantly it has to be able to evolve as the application evolves. How would you add a PartTimeEmployee class? What code would need to change? What impact would this have on other parts of the system?

As complexity increases the effort and cost to develop and maintain this mapping layer increases exponentially.

---

# So How Does An Object Database Work?

## Back to Basics

Firstly some basics, an important concept to understand are that of object identity. In the object world every object can be said to have an identity, this identity is orthogonal its state (the values of its attribute). Object identity is fundamental since it is the means by which objects are manipulated; object identity is used to build relationships between objects, and by navigation to determine which object to access next.

Whilst object identity is fundamental in the object world, it doesn't exist in the relational world, here data is accessed based on its value (using keys). Take a Circle object, should its colour change from *Yellow* to *Red*, its identity remains the same. In the relational world it would no longer be possible to find the row that had previously contained the value *Yellow*. This can (and indeed should) be addressed in a relational database design with unique keys not derived from any application attributes, but in a relational database these values are still potentially changeable by the application.

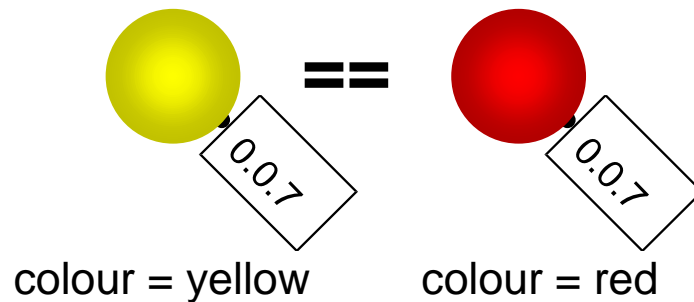


Figure 6 - Object Identity

## Object Database Architectures

Object identity is likewise fundamental to an object database. Figure 7 shows in more detail the client-side architecture of an object database.

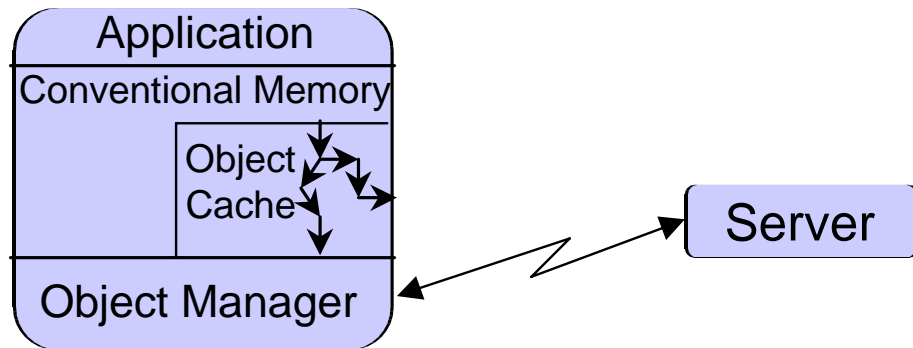


Figure 7 - Client-side architecture of an object database

The application is linked with the Object Manager (OM) provided by the database vendor, this provides transparent navigation (based on object identity) and management of persistent objects, fetching them on demand from the server into the client's object space. The application calls the OM APIs to manage the transactional boundaries; on calling commit (or rollback to discard changes) the OM sends the changed objects back to the server and ends the transaction. The database server enforces transactional integrity and isolation between multiple clients, using locking to ensure *cache-coherency* between the objects held in the client-cache and those in the server. As expected with any database, the server provides transactional recovery on failure and ensures objects held in the database are transactional consistent.

Whilst the various object databases available in the market today all provide the same basic capability they have tended to approach the problem from different perspectives. Most focused on providing persistent extensions to the language, C++ or Smalltalk, and are often categorised as persistent storage engines. Others focused on providing a database for objects. The difference can be exemplified by looking at how queries are processed.

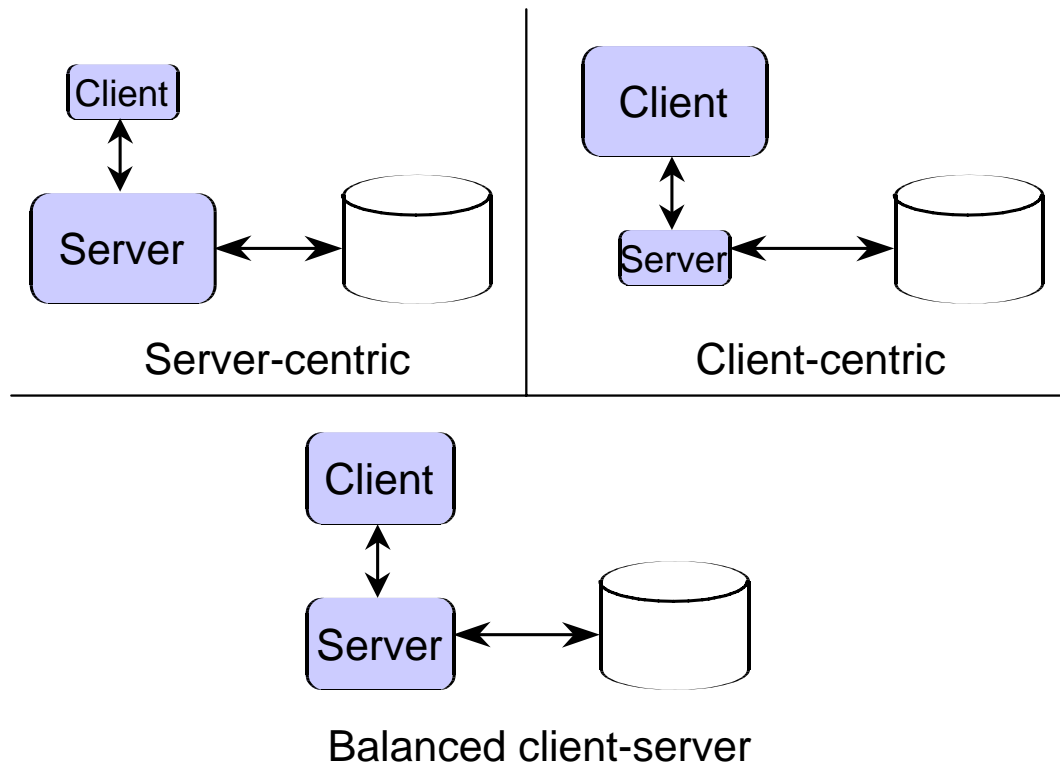


Figure 8 - Different database architectures

The persistent storage engines turned the server-centric approach of the relational database on its head, adopting instead a client-centric approach. Rather than off-loading the data processing to the database server (as when using SQL), they choose to simplify the database server so it is just responsible for managing pages (as in pages of memory). It sends these pages to clients on request (this type of database server is often referred to as a *page server*). The client then performs the required processing. To perform a query all the required objects must be transferred from the database server to the client, the client then determines which objects satisfy the specified criteria.

The second approach is to design a database for objects instead of for undifferentiated blocks of data. This architecture allows three major benefits: Object-aware processing can occur at both the client and the server; objects can be transported as objects rather than as blocks of data; and Objects can be transported between differing platforms more easily.

- The first benefit means that since the database is object-aware, the server can perform processing on those objects. It can perform navigation, queries, maintain inter-object relationships, and other complex functions. To perform a query for example, the client simply specifies the criteria using an SQL-like syntax and passes this to the server. The server optimizes and executes the query, using multi-user indices if available, returning just the resultant objects to the client.

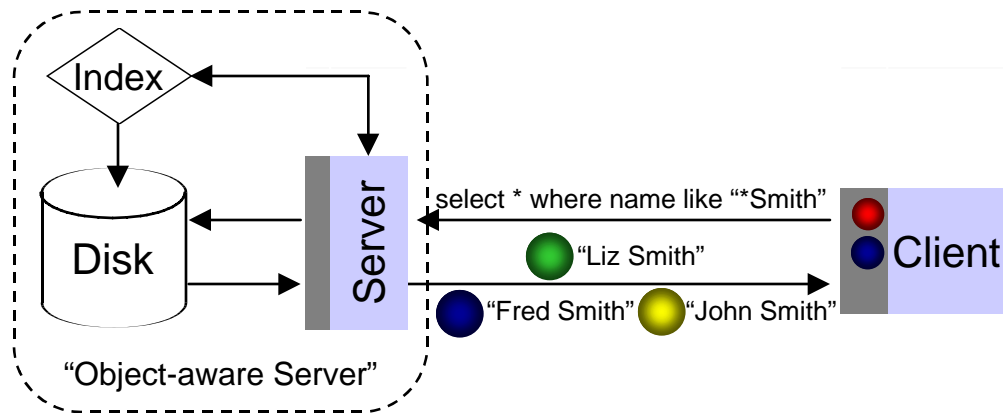


Figure 9 - Query processing

- The second benefit of a database for objects is that the server and client can exchange and cache objects as objects. This reduces overhead in both transport and caching, and allows for much faster performance.
- The third major benefit of the object-aware server is the ability to easily share objects between different platforms, compilers and languages. Because the database understands and stores objects, not pages of memory, NT clients can talk to Solaris servers, Java applications can manipulate objects created by C++ applications and vice-versa. All issues of heterogeneity are taken care of by the database, an important concept to help future-proof today's applications.

Who wants to discover they can't access their objects from a new machine/ compiler/ language because its memory layout is incompatible with that stored in the database, (32bit addressing verses 64bit addressing, for example)? Or to increase the pain of cross-platform development?

# Developing an Application

The Object Data Management Group (ODMG) has defined standard language bindings for C++, Smalltalk and Java to an object database (see <http://www.odmg.org/> for more information). The standard defines:

- How an object model is defined and captured in the database (Object Definition Language). The standard allows for a separate ODL, but generally C++, Smalltalk or Java is used as the definition language.
- How the application interacts with the Object Manager to manipulate the objects (Object Manipulation Language). This defines the language interface for C++, Smalltalk and Java.
- How the application can query the database (Object Query Language). Each language interface provides means of allowing OQL queries to be expressed and executed. Only object databases with *object-aware servers* are able pass the query to the server to be executed. Others use the client to execute the query, transferring all the objects from the database and checking for a match.

As an example lets look at a simple, ODMG C++ application. Figure 10 shows the UML model for the application.

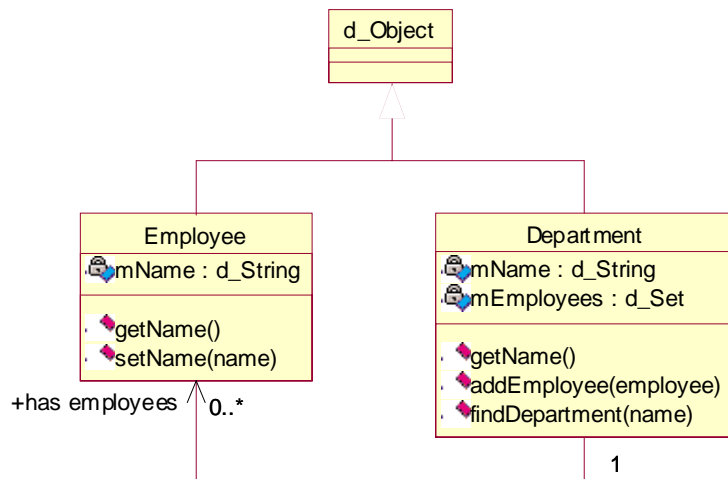


Figure 10 - UML for example

It consists of two classes. An Employee class and a Department class, each have a name. The Department class has a one-to-many relationship to Employee (denoting that a department contains many employees).

---

First lets declare the Employee class:

```
class Employee : public d_Object
{
public:
// Constructors/Destructors
    Employee ( const char* name ) : mName(name) {}
    virtual ~Employee () {}

// Accessor Methods
    const char* getName ( ) const { return mName; }

// Mutator Methods
    void setName ( const char* name )
    {
        mark_modified();
        mName = name;
    }
private:
// Attributes
    d_String mName;
}; // class Employee
```

Figure 11 - Employee class

Employee inherits from the ODMG persistence base-class, `d_Object`. This means the class is now persistent capable (transient or persistent instances of `Employee` can be created). The class has a single attribute, `mName`, which uses the ODMG `d_String` string class rather than a `const char*` (since memory pointers cannot be stored in the database). The mutator method, used to set an employees name, first calls `mark_modified()` to tell the Object Manager that the object is going to be changed (depending on the database, this may result in a write lock being granted to ensure no one else tries to modify the same object).

---

If we now look at the Department class:

```
class Department : public d_Object
{
public:
    // Constructors/Destructors
    Department ( const char* name ) : mName(name) {}
    virtual ~Department ( ) {}

    // Accessor Methods
        const char* getName ( ) { return mName; }

    // Mutator Methods
    void addEmployee ( Employee* employee )
    {
        mark_modified();
        mEmployees.add(employee);
    }

    // Static Methods
    static d_Ref<Department> findDepartment ( const char* name);

private:
    // Attributes
        d_String mName;
        d_Set<d_Ref<Employee> > mEmployees;
}; // class Department
```

Figure 12 - Department class

---

Again the Department class inherits from d\_object and has a name attribute. Each Department has a set of Employees, this relationship is defined using an ODMG d\_Set<> (d\_Set<> is an unordered collection class) of ODMG references, d\_Ref<>, to Employee. The mutator method to add an employee to a department calls mark\_modified() to signify its intention to update itself and adds the given employee to its set of employees. The static method findDepartment() returns a reference to a Department. The d\_Ref<> class replaces the usual use of C++ pointers. C++ pointers are not safe when manipulating persistent objects, a pointer to an object is only valid while it remains in memory. Also pointers are limited by the addressing model of the operating system. The implementation of findDepartment() will be looked at later.

Once the database schema has been captured from the class definitions and loaded into the database, persistent objects can be created using the new operator:

```
Employee* emp1 =  
    new Employee("Mickey"); // Transient instance  
  
Employee* emp2 =  
    new(db, "Employee") Employee("Donald"); // Persistent instance
```

Figure 13 - Creating an object

The ODMG overloaded new operator takes two additional parameters: the database in which the object is to be created; and the class of the object. Otherwise it functions as the standard new operator. To create a transient instance of a persistent capable class just call the standard new operator.

Figure 14 shows a simple application that opens a database, starts a transaction, creates a department and some employees and commits:

```
main()  
{  
    d_Database db;  
    d_Transaction txn;  
  
    db.open("Staff.db"); // Open a database connection  
    txn.begin(); // Start a transaction  
  
    // Create a persistent Department and some Employees
```

```

Department* dep = new(db, "Department") Department("RD");
Employee* emp1 = new(db, "Employee") Employee("Mickey");
Employee* emp2 = new(db, "Employee") Employee("Donald");

// Add the Employees to the Department
dep->addEmployee(emp1);
dep->addEmployee(emp2);

txn.commit(); // Commit the transaction
db.close(); // Close the database connection
}

```

Figure 14 - A simple application

The ODMG `d_Database` and `d_Transaction` classes interface to the Object Manager and are used to control database connections and transaction boundaries. The rest is standard C++ code.

Finally, taking a look at the implementation of `findDepartment()` demonstrates the use of queries:

```

d_Ref<Department> Department::findDepartment ( const char *name )
{
    d_Set<d_Ref<Department> > results; // Results of query
    d_VQL_Query query // Query object
    ("select SelfOID from Department where name like $1");

    query << name; // Bind name to $1
    d_oql_execute(query, results); // Execute the query
    if (results.cardinality() > 1)
    {
        // Oops, found more than one department
        throw InvalidDepartmentException(name);
    }
}

```

---

```
else if (results.cardinality() == 1)
{
    // Found a single match, return the first element in set
    return *results.begin();
}
else
{
    // No match found, return NULL
    return NULL;
}
}
```

Figure 15 - Performing a query

The query object is built using the SQL-like select syntax, SelfOID denotes that the result is a collection of object references of matching objects (rather than a projection of attributes from the matched objects). `d_oql_execute()` executes the query.

## Conclusion

As can be seen, building a C++ application that interfaces to an object database is straightforward. The ODMG language binding hides all the complexity of storing and retrieving objects.

Eliminating the complex mapping required to use a relational database results in less code to design, write, debug, test and maintain. This can lead to significantly shorter development time scales, one of the fabled promises of object-orientation.

By using the same object representation in the database as that manipulated by the application, providing direct support for complex relationships and navigation, an object database can deliver significant performance gains over a relational database.

Using an object database that has a *balanced client-server* architecture ensures maximum flexibility in application design, allowing complex navigation to be performed by the client and ad-hoc queries by the server, delivering the best performance in both cases, and supporting cross-platform communication.

---

# But What's The Impact On My Development Process?

## Getting the Design Right

The use of an object database is more pervasive in the development process than a relational database. Thought must be given to issues of concurrency, performance and scalability during analysis and design (rather than during deployment, by relying on the DBA to optimise the data model and define appropriate indexing strategies). This is not because the object database itself requires it, but because concurrency, performance and scalability are real-world requirements that need to be reflected throughout analysis, design and into implementation. The use of a database (object or otherwise) is often driven by requirements related to concurrency, performance and scalability.

Since part of the strength of an object database comes from its ability to support complex relationships, allowing clients to navigate and perform complex manipulations, it is essential to ensure that the object model includes the navigational paths dictated by the system's transactions. This is often seen as a drawback of using an object database, the emphasis on getting the class model right. But is it any less true if using a relational database? It may be argued that a DBA can tune a data model; de-normalise tables, define indexes, build join tables, and so on to achieve performance and concurrency requirements. And in a purely data driven application this may be true, but few applications are purely data-driven. It was shown earlier that the impact on the mapping code when changing the database schema is potentially catastrophic and indexes or join tables can only be created where common keys already exist. If a particular access path wasn't envisaged then neither relational or object database can help you. There is no exception to the rule:

*There is no substitute for good design*

Due in large part to the much greater cost of fixing problems in the implementation or maintenance phases, today's object-oriented methodologies, like the Rational Objectory Process\* , place a great importance on getting the design right. They place emphasis on the consideration of *use cases* or *scenarios* that help identify interactions (transactions) within a system and bring to the fore, thoughts of performance and concurrency. Object databases are a natural part of this approach. The role of the ODBA should be more proactive. Instead of becoming involved only at the end of the project to tune and manage the data, the ODBA has more of an architectural role, and works as part of the analysis, design and implementation teams. The ODBA helps to consider issues of concurrency, performance and scalability; ensuring that transactions are designed making best use the features of the chosen technology. And since the ODBA is taking a more proactive role within the project, why not introduce the concept of a class librarian, facilitating reuse by providing a repository for class libraries and components.

## Understanding Access Patterns

Access patterns are the ways in which a system accesses and manipulates its objects. Consideration of access patterns helps to identify the navigational paths required by a system's transactions. The performance requirements of these transactions will in turn sway implementation decisions, ensuring that required relationships are present or supporting classes available.

\* The Objectory Software Development Process. Addison-Wesley, Jacobson, Booch, Rumbaugh.

---

A persistent storage engine places great importance on defining relationships, since navigation is the only means of access. If an object can't be reached via navigation then it is garbage (which is why some technologies advocate garbage collection within the database). This client-centric approach places a great emphasis on maintaining collections of objects to support the required access patterns. Taking a look at the earlier example, to support the `findDepartment()` method all Department objects would have to be inserted into a collection of departments (the collection would likely be a map, associating a department name to a department very efficiently).

This is an obvious requirement, but its impact on concurrency may not be. Each time a department is created it must be inserted into the collection of departments. Not a problem for an application using an in-memory model, but it can act as a potential concurrency *hot-spot* otherwise; since only one transaction can create a department at any time (due to the need to update the department collection). This may not be evident during initial prototyping and isn't pleasant to discover during deployment as the number of users, levels of concurrency or transaction rates increase. And whilst the access path is available to navigate from a department to its employees, what for a latter requirement to find all employees with a given name? This would have to be performed by retrieving each department to the client and navigating to each employee, checking for a match, one by one; unless this was foreseen during design and a collection of employees added (in which case a query over the collection would suffice). But how would either approach scale to manage millions of objects.

Compare this approach to that of using an object database with an *object-aware server*. The class declarations remain the same, but since the server automatically maintains a collection of all objects in a given class, referred to as a class extent, it is not necessary to maintain a department collection. `findDepartment()` would be implemented using the query capability of the database. Likewise for a latter requirement to find an employee, rather than navigating to every employee via a department, a query could be used (with an index if available) over the Employee class extent. Of course finding the employees of a department is still a simple means of navigation, unlike a relational database that would require a query.

With this approach it is possible to balance concurrency, performance and scalability requirements. Navigation has the potential to deliver the best performance at the expense of having to maintain the required relationships. Querying, whilst not as fast, offers a greater degree of concurrency (no need for collections) and is more scalable, being able to manage millions of objects and deliver near constant performance through the use of indexes.

The elimination of potential concurrency *hot-spots* when using a balanced client-server approach makes it much more suited to multi-user, highly concurrent database applications. And since every object is reachable via a query, there is no need for garbage collection to reclaim space.

## Developing Transactions

Every database client needs to execute one or more transactions. For those not familiar with developing database applications deciding on what constitutes a transaction can be one of the toughest problem. A transaction should consist of a logical unit of work that is either done in its entirety or not done at all. An object transaction will typically start by creating an object, navigating from an ODMG root object (simply an object which has been given a name) or performing a query. Thereafter anything can happen. A transaction finishes when the application issues a commit or rollback.

When developing transactions a few simple rules apply:

- Keep transactions short.

Thought should be given to the impact of holding locks for extended periods of time in multi-user systems. For GUI applications, an optimistic locking model may be more appropriate (where locks aren't held, instead timestamps ensure multiple updates don't occur).

- Use navigation.

Navigation should be the primary means of interaction with the database.

- Judiciously use queries.

For object databases with *object-aware* servers this can help maintain near-constant performance as numbers of objects increase by minimising the objects transferred to the client and eliminating *hot-spots*.

The ability to perform queries adds a fourth dimension to the way an object application would otherwise be developed. As an example, figure 16 shows a directed **acyclic** graph of circles.

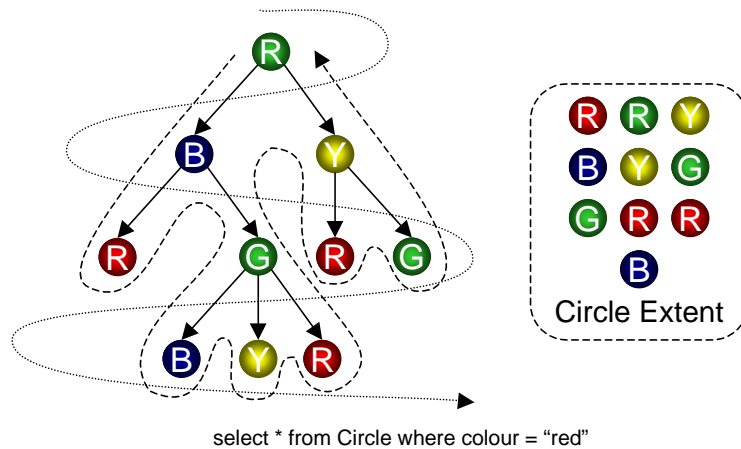


Figure 16 - A DAG

---

Most transactions would navigate the tree-structure, depth or breadth first, making the most of the object database's ability to handle complex structures. And where a tree walk is required navigation is the optimal approach, offering performance far superior to that achievable with a comparable relational solution. But for searches, navigation may not prove the most efficient means of solving the problem, especially as the graph grows into the millions (as the number of objects doubles so does the time to navigate them). Instead a query over all instances of Circle (its class extent) to identify those matching the specified criteria will return in near-constant time, irrespective of the number of objects (when used in conjunction with an index).

## Conclusion

The use of an object database provides a less segregated approach to object-oriented development than using a relational database. The emphasis of current methodologies on understanding the real-world requirements of a system dovetails neatly with the use of an object database. The lesson from objects, as with other technologies is that it's important to get the design right, no technology can overcome poor design. The argument for keeping the database (and data definition) segregated from the application to ensure that it can be optimized, independent of the application, is a fallacy. Mapping objects to tables results in convoluted relational schemas and large amounts of mapping code.

Key issues to be addressed revolve around concurrency, performance and scalability requirements. These will invariably impact class design and implementation, with consideration being given to the capabilities offered by the database itself. Persistent storage engines may prove sufficient for single-user applications that require persistence (embedded systems, CAD/CAM packages). But only an object database with the capability and flexibility to be a database for objects allows multiple applications, with different transaction types and access patterns, to each optimise their use of the database to deliver maximum performance and concurrency.

---

# And How Do I Manage a Deployed System?

Deploying an object database involves similar considerations to deploying any database. It consists of a number of data volumes stored on disk, that need to be backed up and managed. Hardware considerations like disk speed, number of CPUs and speed, I/O bandwidth will all affect the behaviour of the database. Understanding the database architecture helps in extrapolating likely cause and effects. Availability and recoverability requirements dictate how a database is managed and which policies and procedures are put in place. Therefore the skills normally equated with the DBA are still a necessary part of deploying an object database, but the focus of the DBA does change.

## The Role of the ODBA

The role of the database administrator in the object database world is different than it is in the relational world. The task of optimization happens earlier in the process, with the ODBA providing architectural input to help get the design right, instead of after the fact. Therefore some tasks disappear (for example, building join tables). The class model identifies access patterns by virtue of its relationships; therefore much of the optimization is implicit in this approach. The development team, including the ODBA in an expanded role, has optimized the transactions to make the best use of available features to ensure the performance, scalability and concurrency requirements are met. There is little design repair work for the ODBA to do here after the fact.

Day-to-day management of the database is, of course, still necessary. Backup strategies need to be defined and implemented, additional volumes need to be added, high availability solutions need to be put in place, and as the applications evolve the database schema needs to evolve with it. These are all tasks that still fall into the realm of the DBA. This pure administrative role is perhaps 20% of the workload associated with a typically RDBMS DBA. This shift in workload presents an opportunity for the DBA to become more active within the overall development process, participating in class and transaction design, sharing knowledge regarding the impact of using a database.

## Administration Tasks

Each object database offers different levels of administrative support depending upon the underlying capabilities of their architecture. One aspect to consider is whether the tasks can be performed on-line (particularly important if backing up, adding extra data volumes or evolving schema). If tasks require the database to be shutdown it will significantly effect how the database can be deployed. The following list identifies typical administrative tasks:

- Creating and deleting a database

In a development environment developers usually undertake this. In a deployment environment databases should be created and managed by the ODBA.

- 
- Starting and stopping a database

It is useful if the database starts automatically upon first connection, undergoing automatic recovery if required, to help reduce administrative intervention. An option to stop the database after a specified time of inactivity also helps.

- Backing up and recovering a database

Backup should be an on-line task; the database should be available during a backup. Support for incremental backups helps reduce backup times particularly for large databases. The option to archive transaction records since the last backup is useful in facilitating full recovery. A backup is restored and the archived transaction records replayed.

- Adding additional data volumes to a database

Adding additional space to a database should be an on-line task; where a database consists of many data volumes, each potentially on different physical devices for reasons of performance or reliability.

- Defining clustering strategies

It should be possible to create logical partitions, each consisting of several physical volumes, and be able to specify which classes are stored in which partitions to support physical optimization of storage if required.

- Reorganising a database

A database should support automatic, on-line space reclamation and reuse to ensure optimal performance over time. An option to undergo off-line reorganisation is useful to optimise internal structures periodically.

- Adding and removing indexes

Adding and removing indexes should be an on-line task performed at any time after database creation (for those databases that support server-side querying).

- Evolving the schema of a database

Evolving the database schema should be an on-line task allowing the addition of a new class or removal of an existing one, addition, renaming or removal of attributes of a class. Support for *lazy* schema evolution allows the schema to be evolved without changing the actual objects themselves, especially important for large databases. Objects are evolved into the new representation as accessed, over time. Schema evolution, by virtue of its potential to break applications, should be controlled by the ODBA.

- Granting access to a database

Granting access rights to a database should be on-line. As a minimum it should be possible to specify user-level on a read or read/write basis.

- 
- Browsing and querying a database

It should be possible to browse the database schema and objects and perform ad-hoc queries (where supported). An option to insert, update, delete via a browser is useful but can break the encapsulation of the class model since an object's attributes are manipulated directly (however as a developer/DBA tool its invaluable to help fix potential problem areas).

- Monitoring and tuning a database

It should be possible to monitor the health of the database as well as the overall performance to aid tuning.

- Configuring fault tolerance

Not all object databases offer a fault tolerant solution, but for those that do it should be the responsibility of the DBA to configure and manage. Fault tolerance should be transparent to the application; not needing changes to the application code.

## Coexistence

An object database deployed in isolation or embedded as part of an application has little or no requirement to coexist with other technologies. Often however the reality is that the object database must be part of an overall infrastructure. Particularly if this infrastructure consists of existing reporting or data access tools. It is important that these tools can access the information stored within the object database to satisfy the user's needs.

Most object databases vendors can provide an ODBC interface to their database. This should automatically map the class model to a relational model and present this through an ODBC interface allowing ODBC-compliant tools to create, update and delete information in the database, thus preserving the investment in these tools.

## Conclusion

Deploying an object database is like deploying any database, it requires consideration of the hardware and database capabilities, along with the requirements for availability and recoverability. Typically object databases simplify the deployment process by automating many of the administrative tasks and allowing most to be performed on-line.

---

# Conclusion

Object databases are an integral part of an object end-to-end approach. They address the shortfalls of existing database technologies by managing objects as objects, with all their inherent complexities, allowing the developer to concentrate on developing the application and business logic rather than focusing on overcoming the *impedance mismatch* between object and relational models.

As part of an encompassing O-O approach, with the emphasis on getting the design right, they can address issues of:

- Time to market

- Ease of use

- Performance

- Flexibility

- Scalability

- Concurrency

But consideration must be given to choosing the most appropriate product for the job.

Object database's have different architectures and different capabilities. A database with an *object-aware server* offers the highest level of concurrency, scalability and flexibility, all pre-requisites for it to be considered part of an infrastructure supporting a suite of evolving applications. Importantly the object database should be a database first and foremost.

So what are the advantages of an ODBMS? Well that's up to you!

---

# About Versant

## Versant Overview

As a leading provider of object-oriented middleware infrastructure, Versant offers the **Versant Developer Suite** and **Versant enJin**. Versant has demonstrated leadership in offering object-oriented solutions for some of the most demanding and complex applications in the world. Today, Versant solutions support more than 650 customers including AT&T, Alcatel, BNP/Paribas, British Airways, Chicago Stock Exchange, Department of Defense, Ericsson, ING Barings, MCI/Worldcom, Neustar, SIAC, Siemens, TRW, Telstra, among others. The Versant Developer Suite, an object database, helps large-scale enterprise-level customers build and manage highly distributed and complex C++ or Java applications. Its newest e-business product suite, Versant enJin, works with the leading application servers to accelerate Internet transactions.

## Versant History, *Innovating for Excellence*

In 1988, Versant's visionaries began building solutions based on a highly scalable and distributed object-oriented architecture and a patented caching algorithm that proved to be prescient. Versant's initial flagship product, the Versant Object Database Management System (ODBMS), was viewed by the industry as the one true enterprise-scalable object database. Leading telecommunications, financial services, defense and transportation companies have all depended on Versant to solve some of the most complex data management applications in the world. Applications such as fraud detection, risk analysis, yield management and real-time data collection and analysis have benefited from Versant's unique object-oriented architecture.

## **VERSANT Corporation**

### **Worldwide Headquarters**

6539 Dumbarton Circle  
Fremont, CA 94555 USA  
Main: +1 510 789 1500  
Fax: +1 510 789 1515

## **VERSANT Ltd.**

### **European Headquarters**

Unit 4.2 Intec Business Park  
Wade Road, Basingstoke  
Hampshire, RG24 8NE UK  
Main: +44 (0) 1256 366500  
Fax: +44 (0) 1256 366555

## **VERSANT GmbH**

### **Germany**

Arabellastrasse 4  
D-81925 Munich, Germany  
Main: +49-89-920078-0  
Fax: +49-89-920078-44

## **VERSANT S.A.R.L.**

### **France**

10 rue Troyon  
F-92316 Sèvres Cedex, France  
Main: +33 1 450767 00  
Fax: +33 1 450767 01

## **VERSANT Italia S.r.l.**

### **Italy**

Via C. Colombo, 163  
00147 Roma, Italy  
Main: +39 06 5185 0800  
Fax: +39 06 5185 0855

## **VERSANT Israel Ltd.**

### **Israel**

Haouman St., 9  
POB 52210  
91521 Jerusalem, Israel  
Main: +972 2 679 38 30  
Fax: +972 2 679 61 49

## **VERSANT India Pvt Ltd.**

### **India**

1240-A, Subhadra Bhavan,  
Apte Road, Shivajinagar  
Pune 411 004, India  
Main: +91 20 553 9909  
Fax: +91 20 553 9908



**1-800-VERSANT**  
**[www.versant.com](http://www.versant.com)**

WP\_000101r3

© Versant Corporation 2000

Reprinted in 2001

All products are trademarks or registered trademarks of their respective companies in the United States and other countries.

The information contained in this document is a summary only.

For more information about Versant Corporation and its products and services, please contact Versant Worldwide or European Headquarters.