WHITE PAPER

A COMPARISON BETWEEN RELATIONAL AND OBJECT ORIENTED DATABASES FOR OBJECT ORIENTED APPLICATION DEVELOPMENT

BY:
JONATHAN ROBIE
DIRK BARTELS

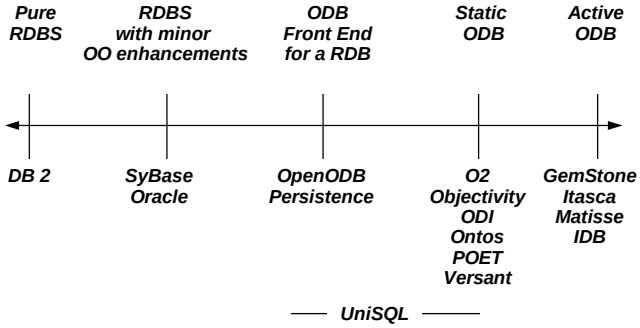
POET SOFTWARE CORPORATION
800 - 950- 8845

Introduction

This White Paper is intended to clarify the differences between relational and object oriented database systems, especially from the perspective of object oriented application development, using programming languages like C++ or Smalltalk. The document is structured into 3 parts:

- Chapter 1 explains the object oriented and the relational model. We are using C++ and SQL for basic examples.
- Chapter 2 describes the ways and shortcomings of integrating an object oriented application model with a relational database system.
- Chapter 3 gives a short overview on object oriented database systems and their potential impact on object oriented application development.

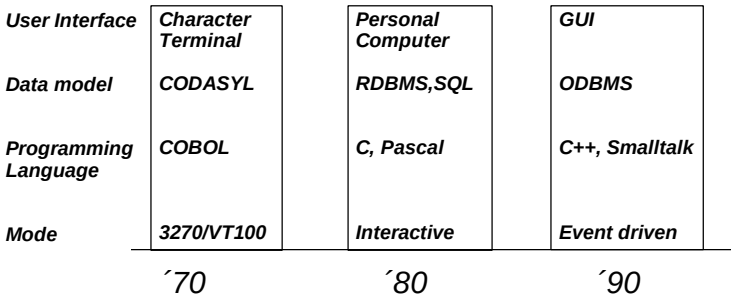
Let us start with some remarks on today's database market. There are many different kinds of database systems available, from the pure relational database systems to the pure object oriented database systems. When we talk about ODBMS in the paper, we only mean static or active ODBMSs. Relational database systems are the other systems mentioned. ODB front ends might reduce some of the shortcomings that are described later on. However they don't change the underlying data model and provide more a migration path than a new technology.



Source: Object-Oriented Strategies, August 1993
(c) Harmon Associates

Classification of database systems

A different trend can be seen in the history of the last 25 years of software development. We have seen dramatic changes in the way we are using and programming a computer. The new challenges for the software industry are leading to object technology and many analysts expect this technology to be the main stream at least until the end of this decade.



Classification of database systems

The authors hope that this White Paper helps you to make your business decisions for the upcoming years. Please let us know if you have any comments to the paper. We would greatly appreciate it.

Yours,

Dirk Bartels (dirk@poet.com) Jonathan Robie (jonathan@poet.com)

Relational Databases and object oriented Languages

Relational database systems revolutionized database management systems in the 1980s, and object oriented programming languages are revolutionizing software development in the 1990s. These systems seem to have complementary strengths. Relational database systems are good for managing large amounts of data; object oriented programming languages are good at expressing complex relationships among objects. Relational database systems are good for data retrieval but provide little support for data manipulation; object oriented programming languages are excellent at data manipulation but provide little or no support for data persistence and retrieval. Many people are trying to combine the two in order to manage large amounts of data with complex relationships.

Unfortunately, the relational model and the object model are fundamentally different, and integrating the two is not straightforward. Relational database systems are based on two-dimensional tables in which each item appears as a row. Relationships among the data are expressed by comparing the values stored in these tables. Languages like SQL allow tables to be combined on the fly to express relationships among the data. The object model is based on the tight integration of code and data, flexible data types, hierarchical relationships among data types, and references. Expressing these basic structures in the two-dimensional tables of relational database systems is not trivial, and this is only the first step. The interface between the two systems must correctly handle data manipulation and data retrieval.

The semantic mismatch between object oriented programming languages and relational databases has led to the development of object oriented database systems which directly support the object model. Object oriented database systems are usually much simpler to use in object oriented programs.

This chapter focuses on the disadvantages of relational database systems in object oriented programming environments. These disadvantages make even small programs more complex and harder to understand. Large and complex applications may fall apart because of this additional complexity. In order to understand these problems we need to understand the models involved.

The Object Model

The object oriented model is based on objects, which are structures that combine related code and data. The description of an object is contained in its class declaration. The basic properties of objects are called encapsulation, inheritance and polymorphism. We will use invoices as an example to illustrate these points.

Encapsulation

Encapsulation means that code and data are packaged together to form objects, and that the implementation of these objects can be hidden from the rest of the program. For instance, a class declaration for an item might look like this:

```
class Item {
private:                    // Data members:
    PtString    name;
    float       price;
    int         number_in_stock;

public:                     // Functions
    int NumAvail();
    int Buy(int n);
    int Sell(int n);
    float TotalPrice();     // including sales tax, VAT, etc.
};
```

Packaging the code and data together helps clarify the relationships among them. If the *Buy()* function increases the number of items in stock, it is clear from the above declaration that *items_in_stock* belongs to the *Item* class. However, this detail is hidden from other classes which use *Item*. Because *number_in_stock* is *private* it can not be accessed directly except in the functions belonging to the Item

class. However, the public part of the class declaration includes functions which use *number_in_stock*. these functions may be accessed by any function or class.

Inheritance

Inheritance is a powerful mechanism which lets a new class be built using code and data declared in other classes. This allows common features of a set of classes to be expressed in a base class, and the code of related class.

Suppose we want to add a new class called *ClothingItem*. All of the code and data defined for *Item* are still relevant, but we need to add sizing information. We can do this by saying that a *ClothingItem* is an *Item* which also has a size. In C++ this might look something like this:

```
class ClothingItem : public Item {
    int size;
};
```

Because we said that *ClothingItem* is an *Item* it has all the code and data defined for *Item*.

Polymorphism

Suppose you have a variety of Items, but the rules for computing their total price differs. For instance, medical items might be exempt from sales tax in an American database system, or a German database system might compute value added tax differently for different items. You still want all Items to have a function called *TotalPrice*(), but instead of one *TotalPrice*() function we now want different functions depending on the class. In C++ this is done with virtual functions.

Here is an oversimplified example:

```
class Item {
private:
    PtString    name;
    float       price;
    int         number_in_stock;
public:
    virtual float TotalPrice();
};

class ClothingItem : public Item {
    int size;
public:
    virtual float TotalPrice()         { return 1.07 * price;}
};

class Medicialtem : public Item {
public:
    virtual float TotalPrice()         { return price; }
};
```

A function which calculates the total price for an invoice can ask each item for its total price without knowing how it is computed for that item. This code will be valid for any *Item* class--even new classes that are added later!

Object Identity, References among Objects, and Collections

Every object in an object oriented system has its own identity. This identity does not depend on the values it contains. In C++ the address of an object is used as its object identity. This allows pointer references to establish the relationships among objects.

Relationships among objects are generally established using pointers. Container classes can be created to express many to one relationships. For instance, if an invoice has a date, a customer, and a set of items we could declare it like this (the set declaration syntax is taken from the POET system):

```
class Invoice {
    PtDate                  date;
    Customer*               customer;
    cset<Item*> items;
};
```

The Relational Model

Database systems are designed for managing large amounts of data, and they provide many important features that object oriented programming languages do not, e.g.:

- fast queries
- sharing of objects among programs
- sophisticated error handling for database operations.
- permanent storage

Relational database systems (RDBS) and record oriented systems based on B-Tree or Indexed Sequential Access Method (ISAM) are the standard systems currently used in most software developments. Each requires that all data be portrayed as a series of two-dimensional tables. The relational model defines the structures, operations, and design principles to be used with these tables. Languages like SQL and Quel provide ways of combining tables to express the relationships among data which are almost completely absent in most B-Tree or ISAM programming libraries.

During the 1980s relational databases became so popular that almost all producers of database software claimed that their products were relational. E.F. Codd responded by releasing a set of 12 rules that should be satisfied by any system claiming to be relational. Unfortunately, these rules excluded all systems then available, even systems like SQL which were based directly on Codd's relational calculus! Since the most powerful relational systems on the market are based on SQL, most of our discussion will focus on SQL database systems. However, everything we say about relational data structure also applies to other table oriented systems.

Relational Data Structure

Table Structure

In relational databases all data is stored in two-dimensional tables. These tables were originally called relations, which led to the name relational databases. An application will generally have a series of tables to hold its data. A database is nothing more or less than a collection of two-dimensional tables.

The rows of this table are composed of fields containing types known to the database. In most systems new data types can not be added.

Primary and foreign Keys

Each row in a table corresponds to one item or record. The position of a given row may change whenever rows are added or deleted. Items stored in the tables can be identified only by their values.

In order to uniquely identify an item it must be given a field or set of fields which is guaranteed to be unique within its table. This is called the primary key for the table, and is often a number. For instance, in our invoice table we might have a field called "invoice-number".

If one object contains the primary key of another object then this allows a relationship between the two items and is called a foreign key. For instance, in our simplified invoice example we might have the following tables:

Invoice

Invoice number	Customer number	Date
1276	1232	3 / 23 / 1992
1289	4567	4 / 1 / 1992

Invoice number is the primary key of the *Invoice* table. *Customer number* is a foreign key referring to a row in the *customers* table:

Customers

Customer number	Name
1232	Eddy Murphy
1456	George Bush

An invoice may contain many rows. Since a row in the invoice table can not hold a variable number of items we need to create a table called Invoice Rows which holds the individual items for an invoice. The primary key for this table contains two fields, Invoice Number and Row Number:

Invoice Rows

Invoice number	Row number	Quantity	Item number
1234	4567	123	1001
1234	1232	1	1032
1289	4567	15	2045

In order to access the price, name, and current stocking level of an item we need to look at the Items table:

Items

Item Number	Name	Price	Number in stock
1001	Elephant Soup	22.34	1066
1032	Nuclear Dental Floss	10.04	157

Data Manipulation

New rows are added to tables using the INSERT statement. Existing rows are changed or deleted using the UPDATE and DELETE statements. The SELECT statement performs queries. UPDATE, DELETE, and SELECT statements select rows based on their values, and operate on all rows in the table.

To add a new item to our table we could use the following code:

```
INSERT
INTO ITEMS (ITEM_NUMBER, NAME, PRICE, QUANTITY)
VALUES ('10036', 'Oil of Broccoli', '5.23', '100')
```

To add one dollar to the price of each item costing less than a dollar we could use this code:

```
UPDATE ITEMS
SET PRICE = 1 + ITEMS.PRICE
WHERE S.PRICE < 1.00
```

To delete all items which cost less than ten dollars we could use this code:

```
DELETE
FROM ITEMS
WHERE P.PRICE < 10
```

Rows from different tables may be combined using joins, which are part of the SELECT statement. The result of such a join is a row, but one that is not stored directly in one table. For instance, to see all item

rows which contain Cream of Elephant Soup need to combine rows from two different tables. We could use this code:

```
SELECT INVOICE_ROWS.INVOICE_NUMBER,
               ITEMS.NAME, INVOICE_ROWS.QUANTITY
               FROM INVOICE_ROWS, ITEMS
               WHERE ITEMS.NAME = 'Cream of Elephant Soup'
```

Views

Virtual tables may be created by specifying the query that should be used to produce the table. These tables are called views, and may combine data from many tables. For instance, we may want to have a more useful form of an invoice row which contains the name and price of the item purchased, but without the inconsistencies that result if we duplicate data. We could do this with the following statement:

```
CREATE VIEW EXPANDED_INVOICE_ROW (INAME, PRICE, QUANTITY)
               AS SELECT ITEMS.NAME,  ITEMS.PRICE
               INVOICE_ROWS.QUANTITY
               FROM ITEMS, INVOICE_ROWS
```

Embedded SQL

Pure SQL is an interpreted language which is limited to database operations. Real applications are rarely written in SQL, but are written in other languages which generate SQL code and pass it to the database as ASCII text.

The SELECT statement returns a table of results, but most programming languages can not manage tables. Cursors are a method for accessing a table sequentially. A cursor is always positioned on a given row. It may be used to retrieve the current row or positioned to another row.

Combining the Models

Relational database systems and object oriented programming systems have very different programming models. They are not very compatible. Moreover, C++ does not have any standard method for accessing data type information at run time, so your interface to the database is likely to be application specific.

Basic Strategies

There are a few general strategies that can be used. You can try to model the database in your object oriented program, you can try to model your application in the database, or you can simplify your database access to the point that the problems become less severe.

Model the Database in your Application Classes

One method is to build your object oriented application around the relational model. Each object needs to know how to store and retrieve its data from the database. This requires extra programming in every class, and does not result in a clean program design since details of the database implementation must be reflected in every class. Since the data structures of object oriented languages are not closely related to the tables of relational databases this code is often non-trivial.

Model your Application in the Database

The second method is to reflect the object model in the relational database. This tends to be much harder than modeling the database in application classes since relational databases have limited data types, and it requires significant coding in the object oriented application because many aspects of objects can not be expressed directly in relational databases (e.g. inheritance, pointer, polymorphism, collections).

Row-oriented Database Interface

If your application only needs simple data types which do not have many relationships then you may be able to limit your interface to row-oriented access. This will greatly simplify your development. In many databases it may be useful to define views which present data to the program in a convenient way.

Some applications need to store complex data but exchange only simple data with a relational database. For such applications it may make sense to use an object oriented database for most data and use simple tables and views to share data with the relational database.

In the rest of this presentation we will show the difficulties involved in supporting an object oriented application with a relational database. We will focus on the problems encountered in data definition, data manipulation, and queries.

Database Schema Definition for Objects

The database schema refers to the structure of the database; in this case, the tables that should be created in the database to support our object oriented application. We will organize our discussion by the basic attributes of objects.

Encapsulation

The ability to combine code and data to form objects is basic to the organization of object oriented programs. Since relational databases do not support this you must write functions for each class to extract data and store it in the database. Similarly, you must also write functions to build objects and retrieve their data from the database.

Relational databases do not support implementation hiding except through password protection, but your whole program will generally run at one privilege level. Everything your program needs becomes public for your program. You can protect yourself somewhat by creating public and private views for the data members of a class, but this only helps if you remember which view to use in each part of your program. Instead of the protection your object oriented programming language guarantees you, you have to settle for conventions that will minimize the risk.

Inheritance

Since relational database systems can not store functions you must ensure that the function hierarchy is restored when you read objects from the database. This usually requires type information to be stored in the database to enable your application to build objects appropriately before loading their data.

The data hierarchy can be represented by creating a separate table for each point in the hierarchy. For instance, if your hierarchy includes an Item, a ClothingItem, and a LingerieItem then you need three tables, one for each class. Each LingerieItem must be stored three times to make sure is present in each of these tables.

Every time you process a LingerieItem you must ensure that all three tables are handled appropriately.

Polymorphism

Relational databases offer no support for polymorphism. You must be careful to store enough information about your data types to enable you to reconstruct objects properly before loading their data.

Object Identity

You can ensure that no two rows in a table have the same primary key by defining the field to be unique. This lets you use the primary key as a form of database identity. However, this does not prevent you from loading multiple copies of an object, which can cause inconsistent updates. In other words, the object identity in your database is lost as soon as the object is loaded into memory.

References among Objects

References among objects are expressed by foreign keys. In general, any pointer field in your objects will be expressed as a foreign key in your database table. The code you use to store your objects must follow pointer references and ensure that the proper foreign key is stored for each referenced object. If you intend to follow pointer references in objects that have been retrieved then you must carefully follow all foreign key references, load the appropriate objects, and correctly initialize the pointer variables.

Data Manipulation for Objects

In the previous section we have seen that relational databases can store only the data from objects and offers no support for many basic aspects of the object model. We have also seen that single objects generally must be stored as a series of tables to express the data hierarchy, and that the relationships among objects must be established using primary and foreign keys.

It comes as no surprise that this data model complicates data manipulation. We would like to illustrate this by describing the steps needed to store, retrieve, update, and delete objects in the database.

Storing Objects

Decide which Tables are used to store the Object

Most objects can not be stored in a single table, but need a series of tables to store the object hierarchy, arrays, variable length data, etc.

The documentation for each class should generally contain detailed information on how it is stored in the database, since this can be rather complex and is not always obvious.

Create Primary Keys for the Tables

Objects that are stored in relational databases should generally contain all primary keys as data members to make it easier to test whether keys have been assigned. Each object will need a primary key for each row in each table used to store it.

Since the database structure is not closely related to the class structure this tends to complicate your classes significantly, but you generally need some way to determine if and how an object has been stored in the database.

Update or create foreign Keys for referenced Objects

You need to convert each pointer or reference in your object to a foreign key since pointers have no meaning in the database.

Extract Data and store it

Do this within a transaction to ensure that the whole object is stored if the transaction succeeds. Since only the object itself knows how to find its private members the object should extract its own data, and it may make sense to have the object store its data. Another option is to create a friend class which knows how to store the data for the class.

Store referenced Objects, if necessary

Depending on the semantic relationships among your objects you may want to store all referenced objects to ensure a consistent state. If so, this may be in the same transaction or a different transaction than the one used to store the original object.

If your objects contain a dirty flag then you can avoid unnecessary database operations when storing referenced objects.

Associate Memory Address with Keys

You probably want to avoid multiple copies of the same object in memory, which violates the semantics of the object model and can lead to a variety of update consistency problems. This means that you need a way to determine if an object in the database has already been loaded. The best way to do this is to keep a table in memory which associates the primary keys with the memory addresses of existing objects. You have to develop a complete object management system for your memory.

Retrieving Objects

Read Type Information to see how Object is built

You need to know the entire class hierarchy of an object to build it correctly. If a ClothingItem is stored then you can read it as an Item, but it will not will not have a size and will not know how to compute sales tax correctly to determine its total price.

Determine if the Object is already in memory

If you have a table in memory which associates primary keys with the memory addresses of existing objects then you can use this to determine if the object is in memory.

If the object is in memory then you will generally want to return a pointer to the existing object rather than creating a second copy. Since this is the program's working copy you probably don't want to overwrite its values with those stored in the database.

Build a new Object, if necessary

If you have stored the complete type information for an object then you can call the appropriate constructor to create a vanilla object that can be stuffed with data. This object will have the right methods, so polymorphism will be preserved.

Fill the new Object with Data from the Database

If your object contains private members, it will have to do this itself or have a friend class do it, since this requires knowledge of the object's memory layout.

Load referenced Objects, if necessary

If your object contains pointers or references then you will generally want to load the referenced objects and initialize pointer fields to the addresses of the corresponding objects. This prevents your program from crashing when it tries to follow a null pointer or an uninitialized pointer.

Changing Objects

Changing objects is similar to storing objects, but you will need the primary key of each table to allow you to specify the UPDATE operations for your database.

Deleting Objects

When deleting objects make sure that the object is correctly deleted in each of its tables. Do this in a transaction to make sure that the entire operation succeeds or fails as a unit.

Queries

Specifying the criteria for a query is somewhat tricky because you have to use the semantics of the data in the object and the structure of your tables to create SQL code for your database. To do this you will need to know which table is used to store each data member and how the primary and foreign keys are used to build the relationships among the object's data.

This becomes even trickier if you want to do a query that involves values in a referenced object. Now two objects need to cooperate to build the query since neither object has knowledge of the database structure of the other object. Alternatively, one query class could be created and declared a friend class of all classes stored in the database.

Since the result set of a query is a full table you need to find some way to bring results into your query. You will need to use cursors to step through the results sequentially. In some cases you may want to return key values for the objects which have been found, in others you may want to load the objects and return pointers to them. Note that you have to avoid creating multiple copies of objects in memory when you perform queries!

Limitations of These Solutions

We have tried to show how to support objects with a relational database system. None of the operations we have described are particularly complex; these are meat-and-potatoes operations that will be used in most programs. Although the operations are not complex, the interface needed to support them is reasonably complex, and it has several significant shortcomings:

Two semantic Spaces, Two separate Designs

Relational databases and object oriented programming languages have fundamentally different programming models, and you need to create two different data models for your application. After you create your class hierarchy you need to decide how each class should be stored in the database.

Complex Interface between the two Designs

Because the relational model does not support the most basic aspects of the object model the interface to the database has become complex. The reason is simple: relational databases were never designed to store objects, and objects were never designed to be stored in two dimensional tables.

This Interface is reflected in each Object

A complex interface is bad enough, but when this interface has to be reflected in every single object then it becomes much more difficult to maintain. The result is that your whole program tends to revolve around the fact that you intend to store and retrieve these objects in a database.

All real Object Operations must be Implemented in your Program, not the Database

Your program must build objects when reading their data from the database. It must store all type information and create appropriate constructors for each class that can be retrieved from the database, and it should generally maintain tables to tell it which objects have been loaded and at which addresses. Each object must know how to store itself, and building queries becomes complex.

Object Oriented Databases

Because many people have become frustrated when storing objects in relational databases, a new kind of database has been developed which supports object oriented programs directly.

Support for Objects

There are some databases which only pretend to be object oriented, but real object oriented databases provide full support for objects.

Encapsulation

An object which is read from the database has the same code and data that it had when it was stored. Objects may have private and protected parts, and they are managed appropriately.

Inheritance

Classes which are derived from other classes can be stored with one operation. The database system must know the class hierarchy and manage object storage accordingly.

Polymorphism

When an object is read from the database it is given all the code and data members that it had when it was originally read. This is true even if you read it without knowing its complete type information. For instance, if you store a rocking chair, you might read it back when you look at all the chairs in your database. If you do so, the chair you just read can rock.

Object Identity

Object oriented database systems integrate the object identity in the database with the identity of objects in memory. If you store an object then it knows if it corresponds to an object in the database, and when you retrieve an object it knows if the object has already been loaded into program memory. There is no need for the programmer to maintain the relationship between database objects and objects in memory.

References among Objects

True object oriented database systems can automatically resolve pointer references in your program's objects and represent them in the database. The better systems use pre-compilers to automatically provide full type information to the database.

Basic Database Operations

We would like to show you how basic database operations can be performed in object oriented database systems. In order to be concrete we will provide examples using POET, a commercial object oriented database system which runs on DOS, Windows, and UNIX systems. This is not an overview of POET, which has many more features than we can cover in a few pages, but simply a way of illustrating that basic database operations are much simpler with an object oriented database system.

Storing Objects

If an object has been assigned to a database then you can store it in POET by calling its *Store*() member function:

```
MyObject->Store();
```

Changing Objects

Objects are changed by changing them in memory and calling their Store() member function. Because POET knows the identity of all stored objects it can tell whether the object exists in the database; if so, it updates the existing object instead of creating a new one.

Deleting Objects

You delete an object by telling it to delete itself from the database. This looks like this:

```
MyObject->Delete();
```

Queries

Since queries support all the semantics of your objects and objects to which they refer many forms can be used for queries. Here is a simple example which searches for all people in the database whose names start with the letter 'M'.

To understand this example you need to know that a PersonSet is a set of pointers to objects whose type is Person, and is used to hold the result set. PersonAllSet is the set of all Persons that have been stored in the database. A PersonQuery is an object which holds the query specification.

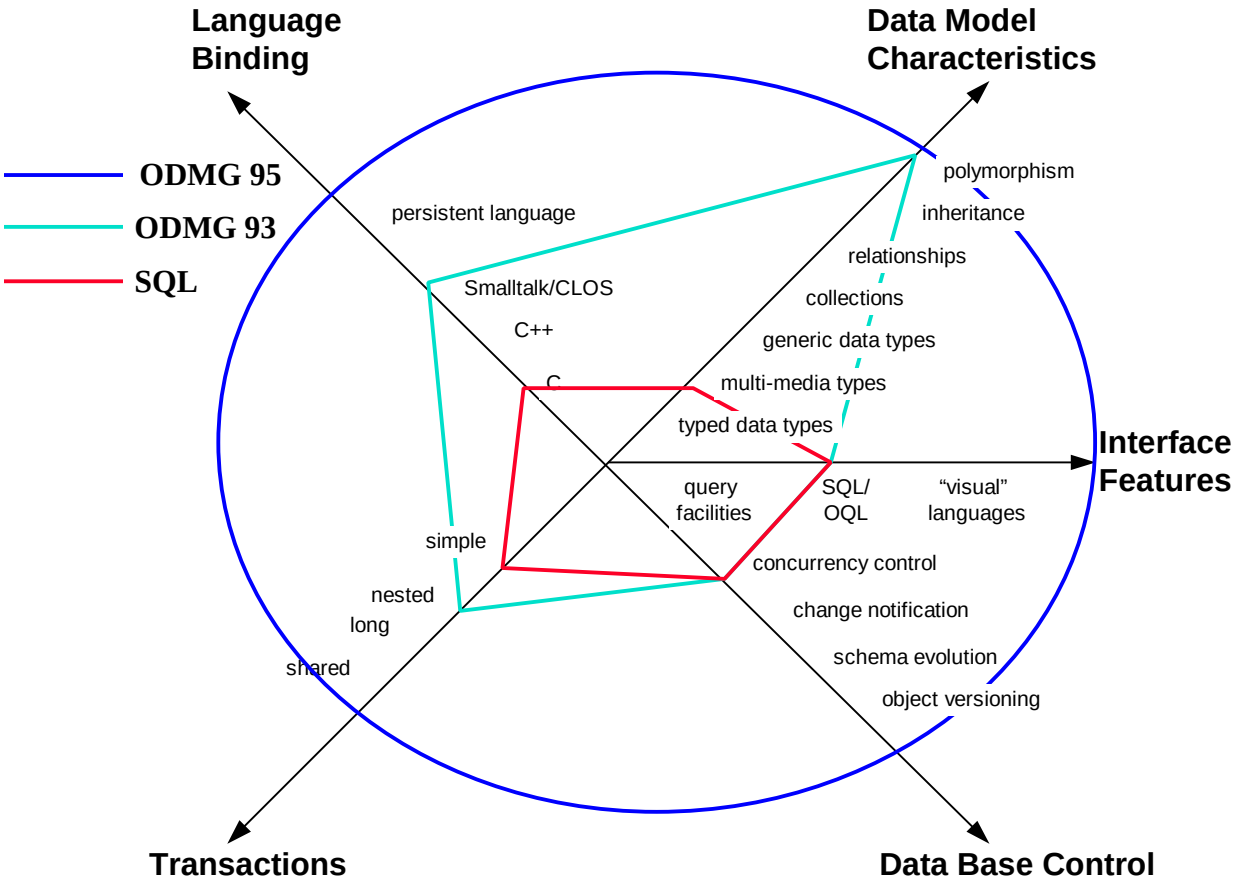
```
typedef lset<Person*>       PersonSet;

    PersonAllSet            *all = new PersonAllSet (objbase);
    PersonSet               *result = new PersonSet;
    PersonQuery q;

    q.SetName ( "M*", PtEQ );          // PtString supports
                                       // wildcard comparisons
    all->Query ( &q, result );
```

Summary

The relational model is simple and elegant, but it is fundamentally different from the object model. Relational databases are not designed for storing objects, and programming an interface for storing objects in a relational database is not trivial.

We have shown that relational databases are poorly adapted to express the basic attributes of objects like encapsulation, inheritance, polymorphism, object identity, references among objects, and user defined data types. The ramifications of this have been illustrated by showing what your program must do to correctly handle simple operations like store, read, delete, change, and search for objects. The strategies we illustrate may be useful for those who need to interface object oriented applications to relational databases.

Finally, we have shown that object oriented databases greatly simplify object database management because they support the object model directly. We have illustrated this by using some simple code examples programmed for POET, an object oriented database for C++.



Comparison of SQL and ODMG standards

Object oriented database system vendors have started to define a standard for this type of technology. They are organized in the Object Database Management Group (ODMG) and the first version of this standard has been published in September 1993. The ODMG-93 standard covers the essentials for object database programming, especially the OO model and the OOP language bindings. This standard has already many advantages compared to extended relational database systems that do not comply to any standard. You can expect a complete standard from the ODMG in less than 2 years from now.