**UNIVERSITY OF OSLO**
**Department of Informatics**

# SQL-99 and Persistent Objects

## Including a Case Study of IBM DB2 Universal Database Server

Roald Andresen

**Cand Scient Thesis**

**2002-07-01**

UNIVERSITY OF OSLO
Department of Informatics

# SQL-99 and Persistent Objects

Including a Case Study of IBM DB2 Universal Database Server

Cand. Scient. Thesis

Roald Andresen

2002-07-01

# Preface

The topic for this thesis was presented to me by my advisor and tutor Dr. Ole Jørgen Anfindsen way back in 1998. It was then 8 years since I finished my Cand.Mag. (Batchelor) degree, and I had more or less given up on getting a chance to get a Cand.Scient. (M.Sc.) degree. The topic suggested by Dr. Anfindsen just happened to coincide to a large extent with my then current job at Posten SDS AS. This made it possible to combine job and studies without creating any conflict. All the work of my Cand.Scient. degree has been done in parallel to a full-time professional career at Posten SDS AS (now Ergo Group AS) and Logica Ltd.

This has been my third attempt at taking a Cand.Scient. degree parallel to work and family. I would like to extend an apology to the two professors who spent time in vain on my two previous attempts: Prof. Khalid Azim Mughal and Prof. Trond Steihaug at the Dept. of Informatics, University of Bergen, Norway.

I am very grateful to everyone that made it possible for me to start on this endeavour, and furthermore, to actually finishing it. My thanks go primarily to my wife, Eli Sæterdal, and my three children Thomas, Ellen and Georg. They have all been far more supportive and patience than I deserve.

Naturally, I would also like to thank Dr. Anfindsen for giving me the opportunity to become Cand.Scient. and for his patience during the 4 years this project took to finish. And finally, for introducing me to Indian cuisine. The opinions expressed in this thesis are my own and are not necessarily shared by Dr. Anfindsen.

Furthermore, I would like to thank Guy Lohman at IBM Almaden Research Center, Jim Melton at Oracle Corp. and Jan Ankarstrand at IBM Norway for their input and help.

Roald Andresen
Skien, Norway, 01-July-2002

# Contents

# 1. Introduction

HOW CAN YOU DO WHAT YOU OUGHT, IF YOU DON'T KNOW WHAT YOU'VE GOT? AND IF YOU DON'T KNOW WHICH TO DO OF ALL THE THINGS IN FRONT OF YOU, THEN WHAT YOU'LL HAVE WHEN YOU ARE THROUGH IS JUST A MESS WITHOUT A CLUE

***BENJAMIN HOFF: "THE TAO OF POOH"***

## 1.1. Purpose

A recognized problem facing system developers today is that the data modelling techniques and database technology in general have been outrun by current state-of-the-art software development technology and modern programming languages. In software development environments, object-oriented modelling and programming (and even testing) is by far dominating. Most databases in use are relational and subject to both the openness and the limitations of SQL-92.

In the recent years, the database communities have tried to converge the database technology and the software development technologies. Object-oriented databases have seen the light of day, and attempts have been made to make both the relational data model and SQL more object-oriented. The emerging new data model is generally known as the object-relational database model. SQL has gone through a large revision, with many new features added. A major new version of the standard was published in 1999. This new version is commonly known as SQL-99.

This thesis will first describe what an object-relational database actually is. This description will be based on several sources (Stonebraker et.al. 1990), (Stonebraker & Moore 1996) and (Gulutzan & Pelzer 1999) in addition to the standard itself.

Next, this thesis will look at one specific commercial database system that is marketed as object-relational, namely IBM's DB2 Universal Database Server, version 7.1. The supported SQL-99 features in DB2 UDB will be described and an analysis as to whether it is justifiable to call DB2 UDB an object-relational database system or not, will be given. To some extents, the impact the object-relational aspects of DB2 have on query execution will be studied.

Finally, this thesis will describe different kinds of object persistence. The impedance mismatch problem is described and sought solved by means of object-relational modelling and a modern database programming API; explicitly JDBC 2.0. The database engine used for this is Oracle*9i*.

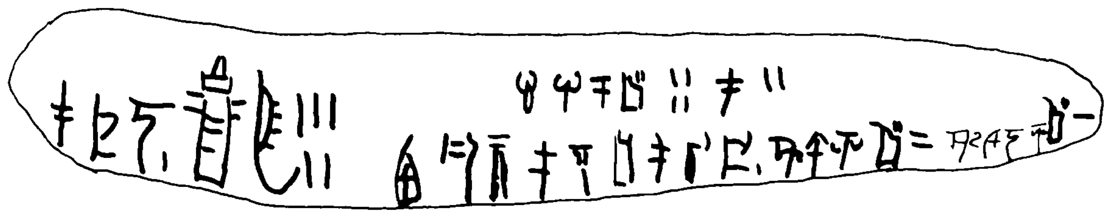
Before all this, a very brief history of database modelling will be given in the rest of this chapter.

This thesis assumes a basic knowledge of database theory, object-oriented concepts, SQL and Java.

## 1.2. Past and Present of Database Modelling

Many a thesis, paper[1] and book have started out with an outline of what have brought the database world to where it is today. This is an attempt do draw a crude sketch of a larger picture, and maybe even to peek into to the future to see what's in store.

Record keeping was one of the very first uses of any written language, alphabetic or hieroglyphic. Even the early Cro-Magnon cave paintings are a sort of recordings of everyday events, such as how many animals were killed during the latest hunt. One of the very first "databases" found in Europe was the Linear-B tablet system of the Minoan Kingdom. These 3500 years old scriptures were the book keeping system or the logistics of a relatively advanced civilisation. Even earlier, the Sumerians kept records of the royal assets and the taxes some 6000 years ago.



*Linear-B tablet showing the recordings of armour in store at the Knossos Palace*

So, throughout the centuries, non-electronic databases have existed and developed from clay tablets stored in jars to paper journals stored in files in filing cabinets.

Computers were originally thought of as advanced computing devises, so when the first commercial computer was introduced in 1951, the means for information storage was rather frail. Information could be stored on punched cards, and a modest amount of 10,000 cards could be stored on tape. The first proper database system was not to be commercially introduced until 16 years later.

In what follows, the most important data models used during the last 4 decades will be briefly introduced. To illustrate these data models, consider the need to model the following scenario:

To develop a very simple library database, the following information should be stored and maintained: *A book consists of one or more chapters. Information about the book should include title, ISBN, year of publication and a page count. Furthermore, information about the book's publisher should be maintained. Finally, a list of the book's authors should be included. Information about an author should include the author's name, nationality and date of birth and death. The publisher's name, address and phone number should also be included.*

Naturally, the history of database technology involves a variety of research and development areas: Concurrency, transaction processing, distributed databases, recovery, security and query processing, just to mention a few. These are not described in this thesis. There are excellent books covering these areas, e.g. (Papadimitriou 1986), (Gray & Reuter 1993), (Özsu & Valduriez 1999) and (Yu & Meng 1998).

[1] See e.g. (Gray 1996).

However, some questions about query processing in the object-relational features in DB2 will be discussed in a later chapter.

### 1.2.1. The Network Data Model

The formal definition of the network data model was the result of 6 years of labour by the *Conference on Data Systems Languages* committee. This committee is more commonly known as the CODASYL committee. Their work was presented in a report from the CODASYL Database Task Group in 1971.

Today, the network data model is almost without exception used on mainframes. Furthermore, the database applications using this model are, with very few exceptions[2], legacy systems.

The network database model is comprised by two data structures, namely records and sets.

Data is stored in records, which are classified into record types. In addition to having a name, each record type also includes a list of data items with names and data types. Complex record types can be defined, as a record type may include vectors and repeating groups in addition to atomic items. A vector is a data item that may have multiple values within a single record, and a repeating group allows the inclusion of a set of composite values for a data item within a single record. Vectors and repeating groups may be combined, resulting in the means to define very complex record types.

A 1-to-n relationship between two record types is described by a set type. A set type has a name, an owner record type and a member record type. The owner record type represents the 1-side of the relationship, whereas the member record type represent the n-side of the relationship. The network data model has no construct for m-to-n relationships. This is solved by the introduction of an additional record type, usually called a linking record type.

Naturally, this is definitively not an exhaustive presentation of the network data model. To give such is not within the scope of this thesis.

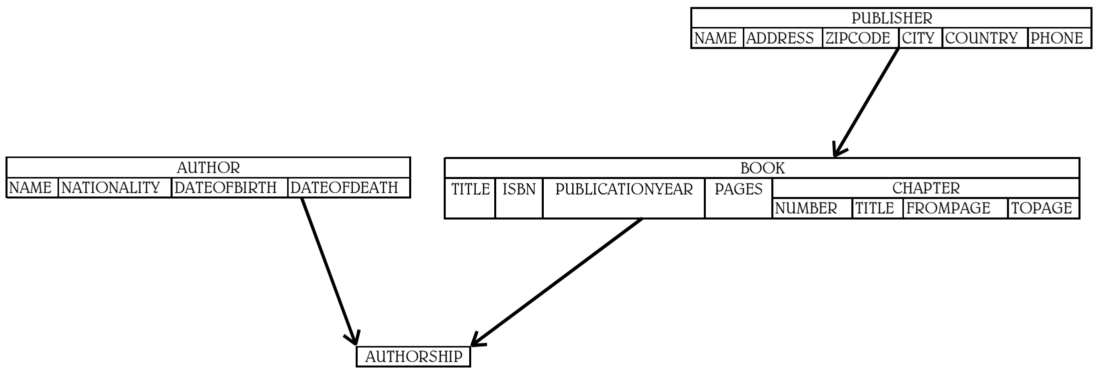
The library scenario immediately gives rise to four record types: *Book*, *Chapter*, *Publisher* and *Author*. Since it is not natural to view a chapter to be an independent entity, but as an integral part of a book, this can be modelled as a repeating group within the book record type.

Since a publisher may publish more than one book, and (in a somewhat simplified world), a book may have only one publisher, there exist a 1-to-n relationship from publisher to book.

There clearly exists an m-to-n relationship between the author record type and the book record type. A linking record type *Authorship* has to be introduced.

[2] Even as late as December 1998, a major Norwegian company posted a request for an estimate on the development of a new business critical application based on the network database model.

One possible network data model representation of the sample scenario is shown in the diagram below:



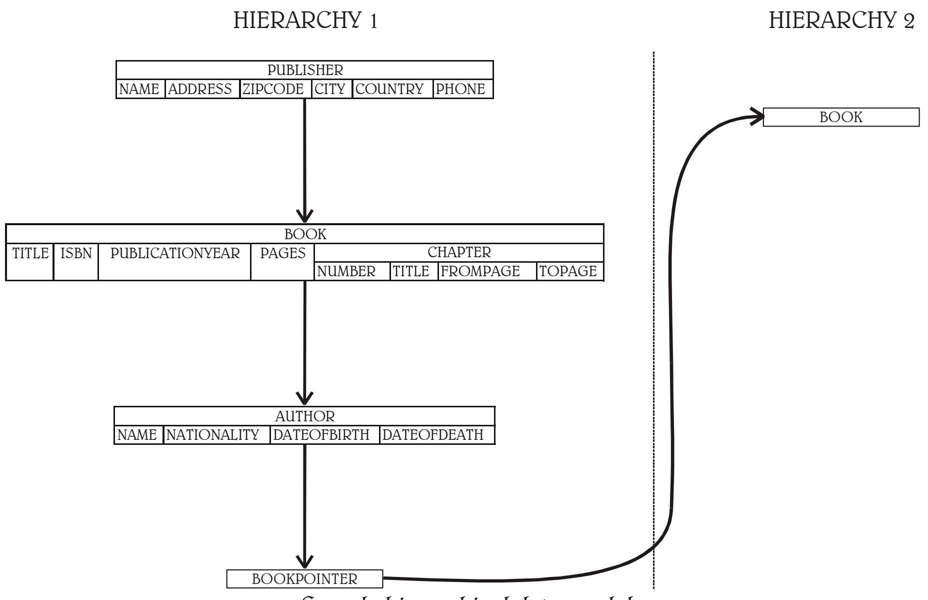
*Sample network data model*

## 1.2.2. The Hierarchical Data Model

Although there is no specific document defining the hierarchical data model, it has proved to become a powerful and important modelling paradigm when modelling the many situations in which a hierarchical structure is evident.

The hierarchical data model is very similar to the network data model. The main difference is that in the hierarchical model data is organised in tree structures rather than in general graphs.

This clearly implies that the library scenario must be presented differently. The 1-to-n relationship from publisher to book is straightforward. When trying to model the m-to-n relationship between book and author, the linking record type cannot be used, since this would result in a graph that is not a tree. To solve this, multiple hierarchies are allowed within the same schema, and a virtual record type is introduced. So, a second hierarchy must be introduced, containing a single virtual record type.



*Sample hierarchical data model*

## 1.2.3. The Relational Data Model

Inspired by software designer's struggle with a very low-level navigational programming interface, E.F.Codd offered an alternative when introducing the relational data model in 1970 (Codd 1970). This proved to be a very successful model, which led to a many very good database implementations, some of which are considered the state-of-the-art database systems today.

In the relational data model, data is stored in relations (or tables). The global schema may be seen as a relational schema **R**$(A_1,A_2,...,A_n)$, where **R** is the relations name, and the $A_i$ are attributes of the schema. An attribute may be seen as representing a data item. A relation of the schema **R** is a set of n-tuples $\{t_1,t_2,...,t_n\}$, each tuple being an ordered list of n values.

Usually **R** is split into components (or tables) such that a commonly acknowledged set of normalisation rules apply. Each component should have a clearly defined key (i.e. a set of attributes which uniquely defines each tuple).

It is possible to navigate (or to find related data) across the components by means of keys and foreign-key constraints (defining a relationship between two components). A more thorough description on the relational data model, and a description of the normalisation rules are described in most basic database theory books, see e.g. (Korth & Silberschatz 1991) or (Ullman 1988).

A 1-to-n relationship is defined by letting the component on the n-side have a foreign-key (or a pointer) to the key of the component on the 1-side of the relationship. Tuples on the n-side is subordinate to a tuple on the 1-side if the value of the foreign-key equals the value of the key on the 1-side.

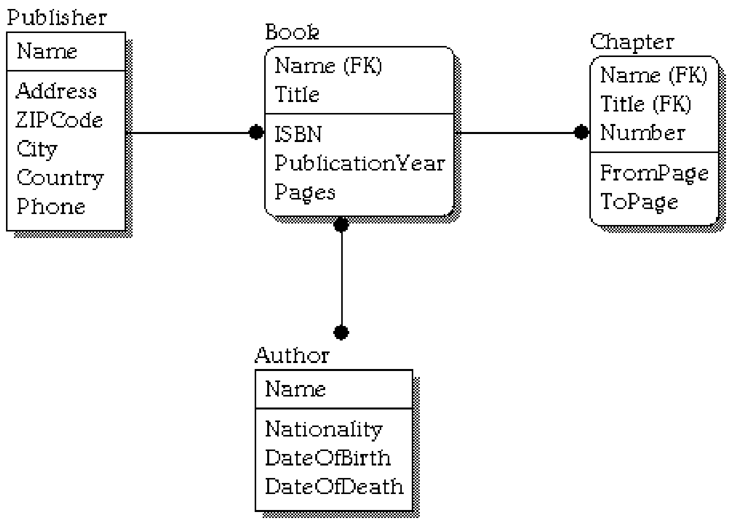
A 1-to-1 relationship is defined in somewhat the same way as a 1-to-n relationship. The only difference is that it is simply a matter of choice which component should have to foreign-key.

Just as the network and the hierarchical model, the relational model has no immediate means to define an m-to-n relationship. In the relational model, this is solved by introducing an additional join component (or a join table), which consists solely of a foreign-key to the component on the m-side and a foreign-key to the component on the n-side. There may be situations where it is useful to include additional attributes in the join component. There is nothing in the relational model that prevents this.

A schema of the relational model is often expressed either in an entity-relationship diagram[3], or in an IDEF1X diagram[4]. In the latter diagram form, the example above is model is expressed like this:

[3] See (Chen 1976)

[4] See e.g. (FIPS/184 1993)



*Sample relational data model in IDEF1X syntax*

### 1.2.4. The Object-Oriented Data Model

Even though object-oriented architectures have been around since the late 1960s, a commercial and widespread use of object-oriented models and object-oriented programming languages did not transpire until the early 1990s. Both object-oriented machine architectures and operating systems have been around since the early 1980s. GemStone, the first object-oriented database system saw the light of day on 1986.

When it comes to modelling principles, an object-relational model is actually an object-oriented model realised in a relational database system.

In recent years, a de-facto standard for object-oriented modelling has been developed through collaboration between a multitude of scientists and software engineers from both commercial companies and academic research institutions. This work has been, and still is, coordinated by the *Object Management Group*[5]. The de-facto standard is known as the *"Unified Modelling Language"*, or UML. A very complete documentation of UML is found in (Rumbaught et.al 1999), and the full specification of UML can be found om OMG's web pages. The application of UML to database modelling has not been formalised, but good approaches described in (Naiburg & Maksimchuk 2001) and (Muller 1999).

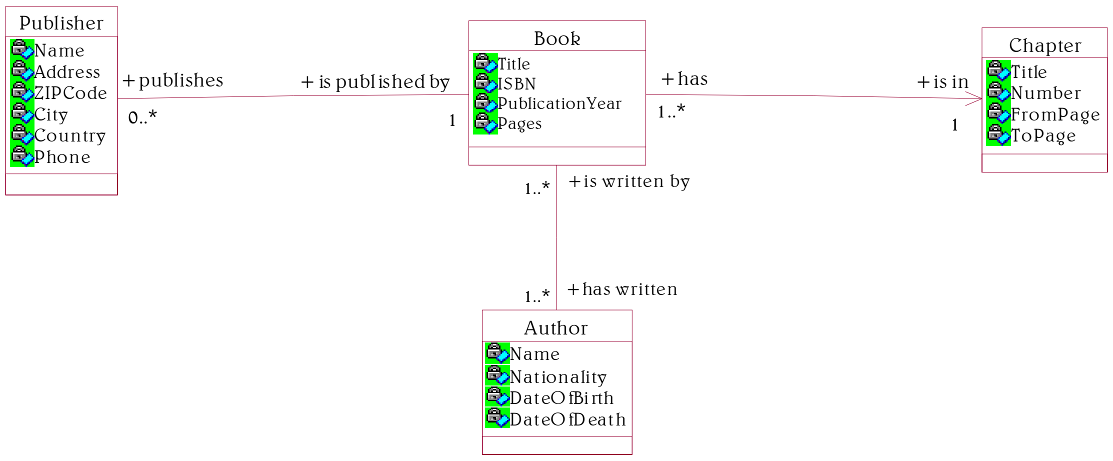
As with the models described above, it is not within the scope of this thesis to describe UML in any detail.

When used in data modelling, a UML diagram has, in the uncomplicated cases, much in common with an IDEF1X diagram. When dealing with cases with greater complexity, UML supports all object-oriented concepts such as inheritance and information hiding. An additional and potentially great reward by using UML instead of IDEF1X is that the same syntax can be used for the modelling of both the application logic and the

[5] *"The Object Management Group (OMG) is an open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications."* See http://www.omg.org

database model. This however assumes that object-orientation is supported in the database in question.

The library scenario is expressed in UML like this:



## 1.3. A New Millenium

Foretelling the future is indisputably a difficult, if not impossible, task. However, leading personae from the database community gathers at regular intervals to do a self-assessment as to where the database community is today, and which challenges lays ahead. The resulting papers from two such gatherings are found in (Silberschatz, et.al. 1996) and (Bernstein, et.al. 1998).

Some of the issues raised are:

Define data models for new data types (such as spatial, temporal and multimedia data), and integrating them with traditional database systems.

Scaling databases to allow for larger, more distributed and more heterogeneous database systems.

Supply further support for automatic data mining and data analysis.

Automate database design.

Apply databases to the Web and utilise the Web as a database.

Further unifying program logic and database systems.

In (Bernstein, et.al. 1998), a ten-year goal for the database research community is presented:

The Information Utility: *Make it easy for everyone to store, organize, access, and analyze the majority of human information online.*

This thesis will not dwell any further on neither database modelling concepts as such, nor on the future goals of the database community.  Object-oriented modelling using UML will be used wherever necessary to illustrate examples and object-relational constructs.

# 2. Persistent Objects

IF NOTHING WERE ETERNAL,
EVEN BECOMING WOULD NOT BE POSSIBLE
***ARISTOTLE, "METAPHYSICS"***

Using the thesaurus in Microsoft Word, a wealth of synonyms are presented for the word *"persistent"*:

- tenacious
- **enduring**
- immovable
- insistent
- indefatigable
- persevering
- **continuous**
- constant
- **continual**
- continued
- repeated
- steady

It is clear that there is no single meaning to the word. For the meaning usually attributed to *persistent objects*, the synonyms written in bold face above seem more appropriate.

Naturally, every object ever created in an object-oriented program has some endurance. This could be endurance within its block of declaration:

```
{
     SomeObjectType  ShortLivedObject;
     ShortLivedObject.SomeMethod();
}
```

Alternatively, the object could persist throughout the whole of program execution.

Some objects, however, is only useful if their attributes and state can persist from one execution to the next. Such an object is called a persistent object.

For how long an object persists is still dependent on the program in question. If a program keeps all its runtime parameters in an object, and this is stored upon program termination and read upon start-up, this is a very simple and short-lived persistent object. The other extreme is an object that should continue to exist for a long time, past the lifetime of both programs and machines. Naturally, it is not enough for the object to be persistent, but it must also be readily available to programs throughout its lifetime (however long it might turn out to be).

This chapter sets out to describe the infamous impedance mismatch problem and different types of persistence. Furthermore, an answer is sought to the question of whether orthogonal persistence can be achieved by accessing data in an ORDBMS through JDBC.

## 2.1. Impedance Mismatch Problem

When working with database from some interfacing programming language, one is often faced with some well-known problems. These problems often result in programmers struggling to make information flow easily between an application and the database. This could be seen as a kind of information flow inertia. The inertia is due to several fundamental differences in the logic of common programming languages and database. A set of commonly known problems has collectively been dubbed the *impedance mismatch problem* (IMP). When dealing with RDBMSs and ORDBMSs in real life, IMP is an issue in the interface between SQL and the programming language in question.

In (Melton 1998), several sides of this problem is presented:

1. Whereas an SQL SELECT statement is a set-at-a-time statement, programming languages are dealing with element-at-a-time statements.

2. SQL has inherent mechanisms for handling NULL values.

3. There is usually, if not always, some mismatch between SQL data types and programming language data types.

4. Most often, SQL has a different way of handling errors, than that of the programming language.

Although it is a general agreement the IMP is a very serious issue when developing applications that are using an RDBMS, it is still a matter of controversy whether ORDBMSs can eliminate the IMP or not. A reasonable and somewhat diplomatic claim is that ORDBMSs certainly lessen the problem.

### 2.1.1. Set-at-a-time Versus Element-at-a-time

Retrieving data through a SELECT statement from an external programming language is not straightforward. Assume the following scenario:

A type and a table is defined in the database:

```
CREATE TYPE T_BOOK
(
     TITLE      STRING,
     WRITTEN    YEAR,
     ISBN       ISBN
);
CREATE TABLE BOOK OF TYPE T_BOOK;
```

In a C++ program, the data type `t_book` can be defined as a `class`:

```
#define year    int;
#define ISBN    string;
class t_book
{
     string     title;
     year       written;
     ISBN       isbn;
     ....
     ....
}
```

An object is defined as:

```
t_book    booktable;
```

The host program executes a SELECT statement S, which gives a result set $R_S$:

```
        SELECT * FROM BOOK WHERE WRITTEN = YEAR( 1984 );
```

If $|R_S|=1$, a one-to-one mapping from the fields in the table to the attributes of the C++ class can be made.

What if $|R_S|=0$ or $|R_S|>1$? The latter is the most usual situation.

Most database APIs[6] approach this problem in a similar manner:

The approach starts with an execution statement, which takes the SELECT statement as an input argument. This results in a cursor or an iterator, which are used to fetch data from the result set. The various fields in the cursor/iterator are bound to host variable. The actual retrieval is done through a fetching mechanism that copies a value from the current row's fields to their respective host variables.

```
        Define a SELECT statement;
        Parse and execute the statement;
        Bind host variables to the statement fields;
        While there are any more rows
             Fetch next row;
             Process the data;
        Optionally drop the cursor/iterator;
```

Surely, this algorithm does deliver all rows, no matter the size of $R_S$. Nevertheless, in SQL $R_s$ is <u>one</u> result set, and the application programmer is forced to do $|R_S|$ retrieves.

Some vendors have approached this problem by allowing the fetching function/method to fetch a predetermined number of rows into an array of host variables. Since $|R_S|$ is unknown, and most likely varies from execution to execution, this is not an optimal solution. A more appealing approach is to allow the fetching function/method to place its result into a linked list:

```
        list< t_book >  booktable;
```

However, if $|R_S|$ is very large this would prove not to be such a good idea after all. Even if `t_book` is a relatively small structure, `booktable` could still put hard-to-meet requirements on the systems resources. Consider the database of the larger National and University Libraries, or the database of Amazon.com[7] where the number of rows in the book table would easily reach several thousands.

To check whether $|R_S|=0$, APIs provide an exception, a `RowCount()` function/method that would return `0` or an `AtEOF()` function/method that would return `false` immediately after the execution of the statement.

[6] E.g. ODBC, Oracle OCI, DB2 CLI, JDBC

[7] http://www.amazon.com

### 2.1.2. Handling NULL Values

Relational databases and SQL inherently handle missing values. E.g., in addition to handle negative and positive values, and the value zero, an integer in an RDBMS/ORDBMS can be without value at all. A variable that is missing a value is said to be a NULL value. The concept of a NULL value was introduced into relational databases to allow operations and calculations over a rowset without having to eliminate missing values in advance.

Traditional programming languages do not have such a concept. To some extent, this could be simulated by letting a variable address be void when a NULL value is intended. For several reasons, this is not a good solution in the general case. One very simple reason is that many languages do not allow pointers at all[8].

In database APIs, the programmer will have to use an *indicator variable* to check if a value is NULL. To run a `SELECT` statement resulting in an n-tuple, the programmer will need 2n variables to fetch values from the database. Furthermore, after the fetch statement has been executed, and before each value is in any expression, the programmer will have to check each corresponding indicator variable to make sure that the value is not NULL.

### 2.1.3. Mismatch of Data Types

When data is to be transferred between variables in a client application and fields in a table, the ideal situation would be that this could be done without any consideration. Unfortunately, this is not the way things work.

It is tempting to assume that at least when the variables in the client application are scalar there should be no problem, but not even this is straightforward. To illustrate the problem, the following tables show the numerical data types in Java, DB2 UDB 7.1 and in Oracle*9i* respectively:

| | Java | DB2 | Oracle*9i*[9] |
|---|---|---|---|
| 8-bit integer | `byte` | `N/A` | N/A |
| 16-bit integer | `short` | `SMALLINT` | N/A |
| 32-bit integer | `int` | `INTEGER` | N/A |
| 64-bit integer | `long` | `BIGINT` | N/A |
| 32-bit floating-point | `float` | `REAL` | N/A |
| 64-bit floating-point | `double` | `DOUBLE/FLOAT` | N/A |

As can be seen, not all of the numerical Java data types have the direct counterpart in DB2 and *none* of them in Oracle*9i*[9].

[8] E.g. Java

[9] All numerical data types in Oracle are designated by specifying `NUMBER(X,Y)` where `X` is the precision and `Y` is the scale of the numerical data type. So, even if Oracle does not have the specific data types, every precision range is covered. The mapping, however, must be done by manually setting/checking `X` and `Y` for each column by the programmer/modeller.

In addition, DB2 also has numerical types where precision is controlled by a parameter in the declaration. These are called `DECIMAL` or `NUMERIC`.

To declare a field to be in the bounds of a 16-bit integer in an Oracle table, the syntax would be

```
SOMEFIELD NUMBER(5)
```

To some extent, similar problems exist with alphanumerical data types. Even trickier problems occur when dealing with date/time data types.

If the RDBMS vendor chose to include a different set of data type mappings for each language, compiler and operating system, this problem could be overcome. These mappings would typically be a set of `#define` macros for a C compiler, or a set of data type definitions for Java. Then, the application developer would be forced to use data types not native to the programming language at hand.

As if this wasn't enough, RDBMSs presents one further data type mismatch problem. If a data structure in the client application code is not comprised by scalar variables, there is no way to map this to a single database table. Consider a typical C structure:

```
struct t_author
{
     char      *name;
     char      *birthCountry;
     t_book    books[][];
};
```

This would have to be mapped to no less than three tables: `t_author`, `t_book` and a table for the many-to-many relation. Fortunately, this should be easier to handle in an ORDBMS:

```
CREATE TYPE T_AUTHOR
(
     NAME                STRING,
     BIRTHCOUNTRY        STRING,
     BOOKS               ARRAY< REF< T_BOOK > >
);
CREATE TABLE AUTHOR OF TYPE T_AUTHOR;
```

## 2.2. Different Levels of Persistence

In (Cooper 1997) any information system that has the ability to store data persistently is called a *persistent system*. A persistent system that also provides the following features is said to be *orthogonally persistent*:

- Persistently stored data has the same logical structure as it has when kept in memory.

- Any data value of any data type can be either persistent or non-persistent.

In addition to orthogonal persistence, there are at least two more levels of persistence: Session persistence and data persistence[10]. These will be explained below.

## 2.3. Session Persistence

This may be regarded as the most primitive form of persistence. Any system that allows the user to save whatever workspace the programming is maintaining in the current session, so that the same workspace may be loaded for later use, can claim to support session persistence. The persistent counterpart of the program's workspace is typically a non-shareable file.

Examples of this kind of persistence are found in word processing systems and spreadsheets. I.e. systems that have little or nothing to do with databases at all.

## 2.4. Data Persistence

Many people automatically think of object-oriented databases when they hear someone talk about persistency. They tend to forget that RDBMSs and even ordinary file systems are persistent stores.

In a traditional relational database, data is stored in tables. As described on page 17, the mismatch of data types in an RDBMS table and the class definitions of an object-oriented programming language, forces a mapping to take place.

When an object is to be stored in the database, the properties that comprise the object must be split into it's scalar parts and placed into the appropriate table columns. Even worse, it may be spread across more than one table.

Likewise, when the values are to be read from the database into some object, all the parts have to be collected and glued together again.

This very clearly illustrates that the data is stored persistently, whereas there are little or no correlation between the class hierarchy in the applications using the data and the table schema in the database.

## 2.5. Indicating orthogonal persistency

Among all the classes and objects defined in an application system, most likely not all should be made persistent. So, how should the application developers indicate which classes and/or objects are to be persistent objects? In persistent systems that do not deliver orthogonal persistence, the answer is obvious: The developer uses different syntax when dealing handling whatever is to be persistent. Specific functions handle file I/O, and access of RDBMS data is handled through either a special API or maybe some kind of embedded SQL. Due to the very definition of "orthogonal persistence", the answer is somewhat subtler.

[10] In (Cooper 1997), data persistence is referred to as *file persistence.*

Usually, the answer is dictated by the architecture chosen by the database vendor. At least seven different approaches can be identified (Cooper 1997):

*Persistent classes:*
The persistence of an object is determined in the class declaration.

*Persistent shadow class:*
For each class, a persistent version of the class is also created. The persistence of an object is determined by declaring it to be an instance of the persistent version of the class.

*Persistent root class:*
A class C is declared as persistent. Any object that is an instance of C or of any class derived from C, is inherently persistent.

*System provided persistent roots:*
The database system provides one or more predefined persistent root classes. Any persistent object must be derived from one (or more) of these root classes.

*Persistence declared at object creation:*
The persistence of an object is indicated when the object is declared, or when its constructor is called.

*Persistence by explicit storage:*
Any object is explicitly stored in the database through the means of a specific *store* command.

*Named root objects:*
This is sometimes called persistence through reachability. One or more objects are declared to be persistent, any object that can be reached through references from one or more of these persistent objects are also persistent.

# 3. Object-Relational Databases

THINGS IN THE COSMOS ARE NOT DISTINCTLY
SEPARATE FROM ANOTHER, NOR ARE THEY CHOPPED
ASUNDER BY AN AXE
***ANAXAGORAS***

This chapter will describe the concept of an object-relational database system. A new ANSI standard intended to extend the SQL-92 standard is under development. This new, not yet fully defined, SQL standard has been given the name SQL-99. SQL-99 is a multipart standard:

| Part # | Part name: | Published[11]: | |
|---|---|---|---|
| 1 | Framework[12] | Yes | |
| 2 | Foundation[13] | Yes | |
| 3 | CLI (Call Level Interface)[14] | Yes | |
| 4 | PSM (Persistent Stored Modules)[15] | Yes | |
| 5 | Bindings [16] | Yes | |
| 6 | (Obsolete) | | **Expected publ.:** |
| 7 | Temporal | | post 2002 |
| 8 | (Obsolete) | | |
| 9 | MED (Management of External Data) | Yes | |
| 10 | OLB (Object Language Binding) | Yes | |
| 11 | Schemata | | post 2002 |
| 12 | Replication | | post 2002 |

[11] As of summer 2001

[12] See (ISO/IEC 9075-1 1999).

[13] See (ISO/IEC 9075-2 1999).

[14] See (ISO/IEC 9075-3 1999).

[15] See (ISO/IEC 9075-4 1999).

[16] See (ISO/IEC 9075-5 1999).

As the chart above shows, parts 1 to 5 and 9 and 10 have already been published. Parts 7, 11 and 12 on the other hand are planned and/or proposed parts that are to be included in a future revision of SQL This future revision is expected to start at the end of 2002.[17]

New extensions to SQL are proposed more or less continuously. Which proposals are likely to enter the SQL standards in the future is not easy, if at all possible, to say. Here is a few proposals found in resent articles:

SchemaSQL (Lakshmanan et.al 2001)
This extension offers the capability of uniform manipulation of data and schema in relational multidatabase systems.

SQL/MM (Melton & Eisenberg 2001)
SQL/MM intends to standardize class libraries for science and engineering, full-text and document processing, and methods for the management of multimedia objects such as image, sound, animation, music, and video.

OSQL (Ng 2001)
This extension is intended to provide the users with the capability of capturing the semantics of ordered data in relational databases.

Different commercial software vendors are involved in the development of this new standard. This has proved to be a two-way relationship: On the one hand the vendors have pushed for their already existing extensions of SQL, to be included in the standard, and on the other hand they have to various degrees extended their supported SQL syntax towards the SQL-99 standard specifications. This has resulted in slightly diverging SQL syntaxes, and extracts from some of these diverging syntaxes[18] will be used throughout this chapter when illustrating the different aspects and possibilities in an ORDBMS.

## 3.1. What is an object-relational database?

A complete and proper definition of what an object-relational database actually is has never really been given. The concept was first introduced, or at least formalised in (Stonebraker et.al. 1990), this being the first of three so-called manifestos, which all aims at defining the future directions for database systems. A further elaboration on ORDBMS functionality is found in (Stonebraker & Moore 1996). The second of the fore-mentioned manifestos (Atkinson, et.al. 1990) gives a description of what are actually considered to be the principles behind an object-oriented database. The third manifesto (Date & Darwen 1998) describes an alternative set of principles for an object-relational database. The approach described by Date and Darwen is by most professionals considered a sound theoretical work, but it will most likely never have much impact on the systems that are dominating the commercial market. Despite this apparent indecisiveness both the SQL standards committee and the leading DBMS vendors in the market has converged towards SQL-99. This thesis will thus regard SQL-99 as the ORDBMS standard. This

[17] According to (Melton 2000a)

[18] Mainly the SQL implementations in DB2 UDB v.7.x and Oracle*9i* have been used.

chapter aims at defining and describing the services and options that such an ORDBMS should offer, and that which is covered in part 1 and 2 in the SQL-99 standard.

In (Stonebraker et.al. 1990), three creeds and thirteen propositions are presented as guidelines to the development of a 3rd generation DBMS, or an ORDBMS. The general opinion on what an ORDBMS should be, is to a somewhat varying degree based on these creeds and propositions.

Creed 1: 3rd generation DBMSs will provide support for richer object structures and rules.

Creed 2: 3rd generation DBMSs must incorporate 2nd generation DBMSs ideas and structures.

Creed 3: 3rd generation DBMBs must be open to other subsystems.

As said, this is further elaborated in thirteen propositions, these are grouped according to the main creeds:

Group 1:

1. A 3rd generation DBMS must have a rich type system: A list of desirables follows this proposition:

   Abstract data types.

   Several type constructors (array, sequence, record, set and union)

   Functions as types

   Recursive combinations of the above constructors.

2. Multiple inheritance of types.

3. Functions, database procedures, methods and encapsulation.

4. Unique identifiers for records should automatically be assigned only in those cases where a primary key is not defined.[19]

5. Rules will become a major feature in future systems.[20]

Group 2:

1. Essentially all programmatic access to a database should be through a nonprocedural, high-level access language.

2. There should be at least two ways to specify collections, one using enumeration of members, and one using the query language to specify membership.

19 This is opposed to the OIDs in the ODMG proposed standard for OODBMSs.

20 Some commercially available RDBMSs already support the use of rules.

3. Updateable views are essential.

4. Performance indicators should not be a part of the data models.

Group 3:

1. Support for persistency through a variety of languages.

2. SQL is intergalactic *"dataspeak"*.

3. Queries and their resulting answers should be the lowest level of communication between a client and a server.

The rest of this chapter will describe how the SQL-99 standard has answered these creeds and propositions. A very good documentation of this new standard is found in (Gulutzan & Pelzer 1999); this book will be used extensively as a reference throughout this chapter.

## 3.2. Type extensions

The SQL-92 standard provides only a limited set of data types.

| SQL-92 Data Types: |
|---|
| Integers |
| Floating point numbers |
| Character strings |
| Date, time and time intervals |

Different RDBMS vendors provide different variations over these data types. These variations could be fixed and variable length character strings, or single-byte and multiple-byte integers.

Most RDBMSs also supply an extended base type set. A very common example of such extension is the supplementation of a *currency* type[21]. These extensions have become a part of the different RDBMSs gradually, and more or less on a need-to-provide basis. Existing RDBMSs thus provide divergent base types sets.

This need, that has pushed RDBMS vendors to gradually extend the base type set, is a clear signal that developers and users are dealing with more complex data compared to what have been the situation in the past. A need to handle large objects, both textual and binary, has emerged from development in areas such as CAD systems and systems for handling multimedia. A fairly simple example of this need (or at least craving) for storing the picture of an employee in a personnel database:

```
create table emp
(
     name       varchar(50),
     age        integer,
     salary     currency,
     photo      image
);
```

[21] E.g. Microsoft Access

A similar example could be a dental database where all x-ray images are stored. Other large objects could be sounds, videos, complete novels, etc.

The `emp` table above contains an attribute of the type `currency` and an attribute of type `image`. These are not types defined in SQL-92. However, if developers and data modellers were provided with a means to define data types as needed, solutions to real-world problems would be easier to implement. This could very well result in data models and programs with better performance, since developers then could concentrate on the efficiency of code, rather than struggling to squeeze the world they are trying to model into too narrow bounds.

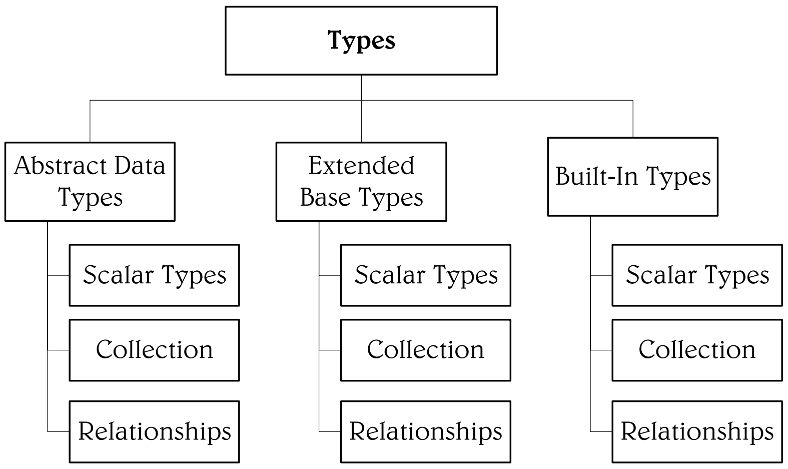
Current RDBMSs are not strongly typed. This means you can easily combine values of distinct but similar types with one another. For example: Assume the `emp` table above is defined.

Since both age and salary are numerical types, there is nothing to prevent a user to run the query:

```
select
     name,
     age*salary absurdity
from emp;
```

Naturally, it makes no sense to multiply an age with a salary, so this defies most people's logic, and most likely is not in accordance with any application's semantics. With strong typing, this would not be allowed. There could however be situations where this kind of type mixing is actually wanted, but in these situations, a deliberately chosen type casting should be the only way to get a result.

There are several ways to approach the need for richer object structures. New data types can be organised as shown here:



## 3.2.1. Extended Base Type Set

When it comes to providing the users with larger flexibility through more data types, a comparatively easy way out for vendors is to statically implement an extended base type set. As mentioned, this is something that vendors have been doing for some time already.

For instance, all relational database management systems provided by IBM have had large object types. Until DB2 Common Server, version 2.1 these LOBs[22] were limited to a maximum length of 32 kilobytes, nevertheless, they were present.

Likewise, Oracle version 7.3 has a LONG data type handling variable-length character data containing up to two gigabytes of information. It also has a RAW and a LONG RAW data type provided to handle binary data.

Microsoft Access-97 includes a multitude of base types: text, memo, number, date/time, currency, autonumber, yes/no, OLE object, and hyperlink.

The base data type set defined in SQL-99 is:

| | |
|---|---|
| Numbers | `Integer` |
| | `smallint` |
| | `numeric [ ( precision [,scale] ) ]` |
| | `decimal [ ( precision [,scale] ) ]` |
| | `float [ (precision) ]` |
| | `double precision` |
| Bit strings | `bit [ (length) ]` |
| | `bit varying (length)` |
| Binary Large Objects | `binary large object (length) 9` |
| Character Strings | `character [ (length) ] [ character set < character set name >] [ collate <collation name`[23]`> ]` |
| | `national character [ (length) ] [ collate <collation name> ]` |
| | `character varying (length) [character set <character set name>] [collate < collation name> ]` |
| | `national character varying (length) [ collate <collation name> ]` |
| | `character large object [ (length) ] [ character set <character set name>] [ collate <collation name> ]` |

[22] **L**arge **Ob**ject

[23] A collation is defined in SQL-99 to be a set of rules that determines the result when character strings are compared.

| | |
|---|---|
| | `national character large object [ (length) ] [ collate <collation name> ]` |
| Temporal | `date` |
| | `time [ (fractional seconds precision) ] [ with without time zone ]` |
| | `timestamp [ (fractional seconds precision) ] [ with\|without time zone ]` |
| | `interval (interval qualifier)` |
| Boolean | `boolean` |

Some of these data types have also an alternative short name, e.g. `national character large object` is also called `nclob`. The various parameters indicated in the type definitions will not be described any further in this thesis.

### 3.2.2. Domain Types

An extended base type set does not make an ORDBMS. An object-relational database system must provide facilities for the *user* to extend the type set when needed.

Allowing the user to create domain types could do this. A domain is a data type that is based directly on existing base data types. This was never really suggested as a separate option in either (Stonebraker et.al. 1990) or (Stonebraker & Moore 1996).

When a domain type has been created, it will be handled as an entirely separate data type. The SQL-99 syntax for the creation of domains is:

```
create domain <domain name> [ as ] <base data type>
     [ default <default value> ]
     [ <domain constraint list> ]
     [ collate <collation type> ]
```

Assume two data types `ohm` and `ampere` have been defined, both as domain types based on `float`:

```
create domain ohm as float
     constraint non_zero
     check ( value >= 0 );

create domain amphere as float
     constraint non_zero
     check ( value >= 0 );
```

Assume further that we have defined this table:

```
create table electricity
(
     id          integer,
     resistance  ohm,
     current     ampere
);
```

Even if both the attributes `resistance` and `current` are represented internally in the database as `floats`, they are treated as utterly incompatible data types.

Imagine we would like to run the following query:

```
        select resistance.current voltage
             from electricity
```

Since an ORDBMS is to be strongly typed, attributes of different data types cannot be combined in expressions, and thus the query is illegal.

For every domain type, there should be casting functions that allow conversion from the domain type to the type it is based on, and visa versa. If we assume further that, a data type `volt` is also defined as a domain type based on `float`:

```
        create domain volt as float
             constraint non_zero
             check ( value >= 0 );
```

Using type casting this query then should be legal:

```
        select
             cast
             (
                  (
                       cast( resistance as float ) *
                       cast( current as float )
                  )
                  as volt
             ) voltage
        from electricity;
```

This query first casts `resistance` and `current` to `floats`, multiplies the results of this, and finally casts the result of the multiplication to `volt`.

### 3.2.3. Abstract Data Types

One of the limitations of the traditional RDBMSs is that every kind of data is forced into tables of atomic data values of basic data types. There is no natural way to model more complex situations. To provide further complexity, and to provide data encapsulation, an ORDBMS should accommodate the user with abstract data types.

An abstract data type in an ORDBMS is presented by a type declaration:

```
        create type t_book
        (
             title       string,   // being derived from varchar
             written     year,     // being derived from integer
             isbn        ISBN      // being derived from varchar
        );
```

This type can then be used for instance in a table definition:

```
        create table book
             of type t_book;
```

If such a type definition is considered the equivalent of a `C++` class, then a row in the book table can be regarded as the equivalent with a `C++` object

So, what is so great about that? Couldn't this simply have been defined as a table looking like this?

```
        create table book
        (
             title       string,   // being derived from varchar
             written     year,     // being derived from integer
             isbn        ISBN      // being derived from varchar
        );
```

Well, the *type* approach has several advantages over the traditional table approach:

A consistent representation of a logical data unit (or object) that may be used throughout the model may be achieved.

References (i.e. pointers) to objects can be declared.

Objects can be used as parameters to functions.

Type inheritance can be provided.

Data that are inherently part of a relation can be presented as an attribute even if it doesn't fit into the predefined atomic data types.

The introduction of abstract data types gives rise to a far more consistent representation of attributes. Assume a librarian needs to implement a database of the library's books. It is easy to imagine that she needs a table for loaners, and a table for publishers:

```
        create type t_location
        (
             street     string,
             zip        string,
             city       string,
             phone      string
        );
        create table loaners
        (
             name       string,
             id         integer,
             address    t_location
        );
        create table publishers
        (
             name       string,
             address    t_location
        );
```

Both `loaners` and `publishers` have an address of type `t_location`. In a traditional RDBMS the attributes of `t_location` would have to be defined either locally in both the `loaners` table and the `publishers` table, or in a separate table `addresses` to which both the `loaners` table and the `publishers` table would have to have a foreign key.

The examples given above encompass the bare basics of abstract data types, or user-defined types (UDTs) as they are called in the standard, of SQL-99. Not surprisingly, the complete standards syntax is much more elaborate than this. Some of the options available will be covered below in the section on inheritance.

A database type is now becoming suspiciously similar to a class in the object-oriented sense of the word. To further narrow the gap between database types and object-oriented classes, methods could be included as parts of the type definition. Except for a feeble request for inheritance in (Stonebraker et.al. 1990), this has not been regarded as something an ORDBMS needs to have. Nevertheless, some vendors[24] have already seen this as a natural part of an ORDBMS. SQL-99 certainly does include methods as a part of user-defined types.

Still using the library example above, assume there is a need to keep track of when a book was loaned, and when it is due for return. The checkout date is simply a good, old traditional date attribute, but since the library allows loans for 4 weeks only, it should not be necessary to store a return date in the type. We can therefore imagine defining the `t_book` type like this:

```
create type t_book
(
     title      string,
     written    year,
     isbn       ISBN,
     check_out  date,
     method Due_Date()
          returns date
);
create table book
     of type t_book;
```

Ignoring the actual implementation of the `Due_Date()` method for now, it is easy to imagine a query like this:

```
select title, Due_Date()
     from book
     where title = 'Phaedo';
```

The actual method is declared in a separate statement, which includes the implementation of the method:

```
create method Due_Date()
     return date
     for t_book
begin
     return ( checkout + 28 );
end;
```

When a create type statement is executed to create type `T`, SQL-99 expects the ORDBMS to create a constructor function for `T`. Following the example above, a statement creating a constructor for `t_book` should be executed:

```
create function t_book()
     return t_book
declare
     v    t_book;
begin
     ....  ;
     return v;
end;
```

[24] E.g. Oracle*9i*.

where `v` is a value, of type `t_book`, with all attributes set to their default values. In addition to the constructor function, the ORDBMS is also expected to create *observer* and *mutator* functions for each of the types attributes. These functions are the `get` and `set` functions of object-oriented classes.

It is tempting to ask whether to provide methods and constructor, observer and mutator functions is a way to bestow the ORDBMS with data encapsulation. To achieve encapsulation there are mainly two options:

1. Automatically enforcing encapsulation of all attributes, and thereby demanding access functions to be provided for all attributes. These could then be provided as default functions by the ORDBMS, with the option for the user to override them when needed, or optionally dropped for those attributes that are intended for internal use only.

2. Providing a means for the user to further qualify whether each attribute should be encapsulated or publicly available[25]. The management of access functions for those encapsulated attributes could then follow the guidelines given in option 1 above.

Option 1 is partly provided for in the SQL-99 standard, but it is still legal to access the attributes directly. Option 2 is not supported at all. Therefore, SQL-99 does not cover encapsulation. It can be argued whether encapsulation is an actual need in an ORDBMS. There are mainly two reasons for encapsulation in object-oriented modelling and programming. These are information hiding and validity checking.

The purpose of information hiding is to hide potential changes in the internal storage of an attribute, and to prevent users from directly accessing an attribute. However, on the one hand, a database type is not likely to change after the database system has been set into production. The primary need a database user has is to access the tables, the records and the attributes of a data model. This raises the question whether encapsulation is actually needed or not. If rules (ref. proposition 5 in group 1 above) is heeded, there is no need for validation through access functions, and therefore no need for encapsulation to provide validity checking. On the other hand, for an object-oriented language, encapsulation is essential and "the right way to do it".

Most access methods can be generalised into the following patterns:

```
method SetAttribX( typeX Value )
begin
     Value := funcA(..., Value, ...  );
     if ( proposition ) then
     begin
          Value := funcB(..., Value, ...  );
          AttribX := Value;
     end;
end;
```

[25] Maybe also provide some form of "friend" availability should be included?

```
typeX method GetAttribX()
     result typeX;
begin
     result := funcC(...,AttribX, ...  );
     return result;
end;
```

In `SetAttribX(...  )`, the parameter value is pre-processed in a function `funcA( ... )`, checked against some `proposition`, and if the `proposition` is satisfied, the `Value` is further processed in a function `funcB( ...   )`, and finally the attribute `AttribX` assigned its `Value`. Naturally, some or all of `funcA`, `proposition` and `funcB` could very well be void.

Assume then that a validation rule has been made to trigger when a new record is inserted or an update of a record is employed. This could then capture all of the functionality of `SetAttribX(...  )`:

```
trigger Validation
     on insert, update of tableX
begin
     Value := funcA(..., new.AttribX, ...  );
     if not ( proposition ) then
          cancel event;
     else
     begin
          Value := funcB( ..., new.AttribX, ...  );
          new.AttribX := Value;
     end;
end;
```

In `GetAttribX()` the function `funcC ( ...   )` could potentially do some pre-processing of the requested attribute before it is passed as a result of the method.

### 3.2.4. Collection Constructors

In (Stonebraker & Moore 1996), the abstract data type discussed above is presented as one of three basic building blocks for creating complex types in an ORDBMS. Another of these basic building blocks is the collection type constructor. This might come in several flavours:

- Set
- Bag
- List
- Stack
- Queue
- Array

It should not be necessary to describe all of these in detail, so focus will be on the *set* and the *array* constructors.

Formally, if `T` is any type, then `set<T>` must also be a data type, namely a set of items of type `T`.

If supporting a set type, an ORDBMS should also support the most common set theoretic operations. Assuming $S_1$ and $S_2$ are sets of type `T`, and `x` is an item of type `T`, these operations should include:

`union(`$S_1$` , `$S_2$`)` resulting in a set $S = S_1 \cup S_2$.

`intersection (S1 , S2)` resulting in a set $S = S_1 \cap S_2$.

`difference (S1 , S2)` resulting in a set $S = S_1 - S_2$.

`inSet(x , S1)` resulting in **true** if $x \in S_1$, and in **false** if not.

`add (x , S1)` adding `x` to the set `S`, such that $S = S_1 \cup \{x\}$.

`remove(x, S1)` removing `x` from the set `S`, such that $S = S_1 - \{x\}$.

Likewise, if `T` is any type, then `T[n]` must also be a data type, namely an array of length `n` of items of type `T`. Recursively this also implies multidimensional arrays.

Like the set type, the array type and all other collection types need their set of supporting functions. For the array type, this should at least include element access by index:

`x := a[k]` assign `x` the value of the $k^{th}$ element of the array.

`a[k] := x` set the $k^{th}$ element of the array to the value of `x`.

If we allow the array type to be a dynamic array type, the set of supporting functions should include:

`a.Append( x )` append `x` to the array.

`a.Insert( x , k )` insert an element into the array so that `x` become the $k^{th}$ element, adjusting the array so that:

`a[ m + 1 ] := a[ m ]` $\forall$ `m` $\geq$ `k.`

`a.Delete( k )` delete the kth element of the array, adjusting the array so that:

`a[ m ] := a[ m + 1 ]` $\forall$ `m` $\geq$ `k.`

This is assuming that the type of `x` is the same type as the array `a` is defined over.

One can easily imagine several other useful functions for addition, deletion and insertion of elements, searching the array for a specific value, merging two arrays, etc.

The SQL-99 standard, however, only specifies one collection type, i.e. **`array`**. Several of the other collection types described above would be very useful. So, hopefully, the future will see both an expansion of the standard and beyond-standard implementation in ORDBMS products.

The declaration of an array attribute is made with the syntax:

```
<data type> ARRAY[ unsigned integer ]
```

### 3.2.5. Reference types

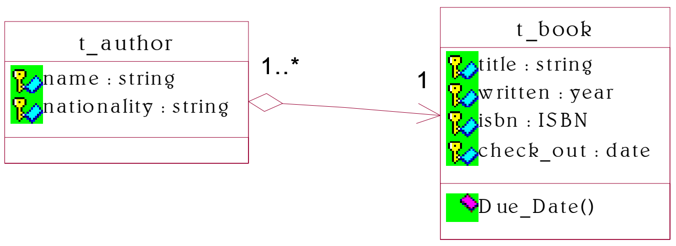
The last of the three basic building blocks for an ORDBMS is the reference type. The reference type is very much similar to an address pointer in traditional programming languages, e.g. `C/C++`.

Formally, if `T` is any abstract data type, then `ref<T>` must be a data type, namely a reference to an object or item of abstract data type `T`.

The declaration of an array attribute is made with the syntax:

```
<data type> REF(<abstract data type>)
     [SCOPE <table name>
          [REFERENCES ARE [NOT] CHECKED
               [ON DELETE
                    {CASCADE |
                     SET NULL |
                     SET DEFAULT |
                     RESTRICT |
                     NO ACTION
                    }
               ]
          ]
     ]
```

Using references can easily be an alternative to traditional foreign keys. Assume the following scenario:



In this scenario, an author may have written one or more books, whereas a book (somewhat artificially) can have been written by only one author. This can then be expressed as:

```
create type t_book
(
     title                   string,
     written                 year,
     isbn                    ISBN,
     check_out               date,
     function DueDate()      return date
);
create table book of type t_book;

create type t_author
(
     name              string,
     nationality       string,
     books             ref(t_book) array[x]26
);
create table author of type t_author;
```

The `books` attribute is then an attribute that is an array of references to objects of type `t_book`.

[26] This limits the number of books written by a specific author to x.

## 3.3. Functions

It is strongly recommended (in (Stonebraker et.al. 1990)) that an ORDBMS must provide the user with the ability to register functions that accept arguments and return results that are anything from scalars to sets of composites. If a function `f` returns a value/object of type `t`, The ORDBMS should allow `f` to be used anywhere a value/object of type `t` is expected. This notion is also covered in the SQL-99 standard.

One can easily see at least three breeds of functions:

1. Functions that simply return the result of an ordinary SQL select statement:

```
create function AuthorsBooks( p_name string )
     return array< ref< t_books > > as
     select books from author where name = p_name;
```

2. Functions implemented in some ORDBMS internal structured programming language[27]:

```
create function AuthorsBooks( p_name string )
     return array< ref< t_books > > as

     tmp-books array< ref < t_books > >;
     cursor tmp_author is
          select * from author;

begin
     // use some kind of constructor to initialise
     // the tmp_books variable
     tmp books := NewArray( ref < tbooks > );

     for author_rec in tmp_author loop
          if ( author_rec.name == p_name ) then
               tmp_books.Append( author_rec.books );
          end if;
     end loop;

     return trap-books;
end;
```

3. Functions implemented in some external structural programming language[28]:

```
create function AuthorsBooks( p_name string )
     return array< ref< t_books > > as
          external name 'e:\cproj\bibl\f-AuthBook'
          language C;
```

Every function should have zero or one return value, and should be able to handle an arbitrary number of parameters. Both parameters and return value should be a scalar value, an object, an object reference, a row set or a collection.

**Example:**

[27] E.g. Oracle's PL/SQL.

[28] This could e.g. be C+ +, Java, SmallTalk, etc.

Consider the situation described on page 12. Assume that a new book is purchased by a library and that this book has to be inserted into the database. This could of course be done like this:

```
INSERT INTO BOOK
      ( TITLE , WRITTEN , ISBN )
      VALUES
      ( '3001 - The Final Odyssey' , 1997 , '0-246-12689-2' );
UPDATE AUTHOR
      BOOKS.APPEND(   SELECT REF(BOOK) FROM BOOK
                      WHERE ISBN='0-246-12689-2' )
      WHERE NAME='Clarke, Arthur C.';
```

This will do the job, but it is awkward. A somewhat better solution would be to define a function `AddBook` on the `t_author` type that takes the title, the written year and the ISBN as input parameters. `AddBook` would then more or less do the same job as the two statements above, but the operation to do the actual addition would then simply be:

```
UPDATE AUTHOR
      AddBook(  '3001 - The Final Odyssey' ,
                1997 ,
                '0-246-12689-2' )
      WHERE NAME='Clarke, Arthur C.';
```

## 3.4. External Functions and Procedures

An alternative to persistently stored procedures and functions is to link the database to an externally defined function or procedure. An externally defined function or procedure is a function or procedure that is implemented and stored outside of the database. It is theoretically irrelevant for a database system what programming language such a function or procedure are to be implemented in. In real-life systems however, the options are usually limited to one or more of the languages C, C++ and Java.

### 3.4.1. Implementation Options of Functions and Procedures

When implementing an object-relational (or an SQL-99 adherent) database management system, designers and developers are faced with a tough challenge. In which context should external functions and procedures run? In (Melton 1998), three different approaches are presented. These approaches are to place the execution context in:

the same context of the database server itself,

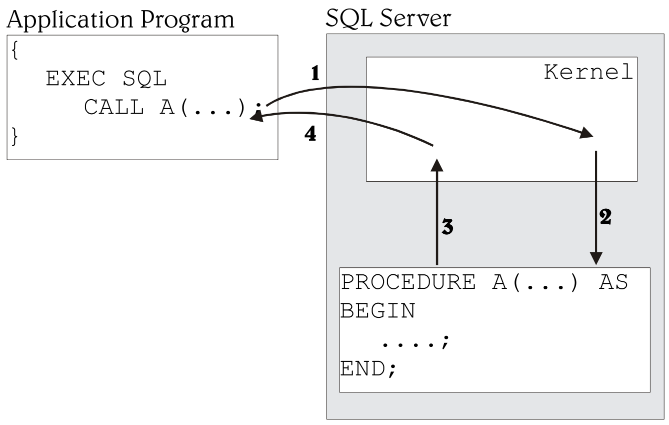
the same context as the actual application program,

or the context outside both the database server and the application program.

All of these have both advantages and disadvantages. In the discussion that follows, only external *procedures* are discussed. The same argument holds for functions.

## In the database server

In this approach, the procedure is executed in the same memory space as the kernel of the database server.



```
{
   EXEC SQL
      CALL A(...);
}
```

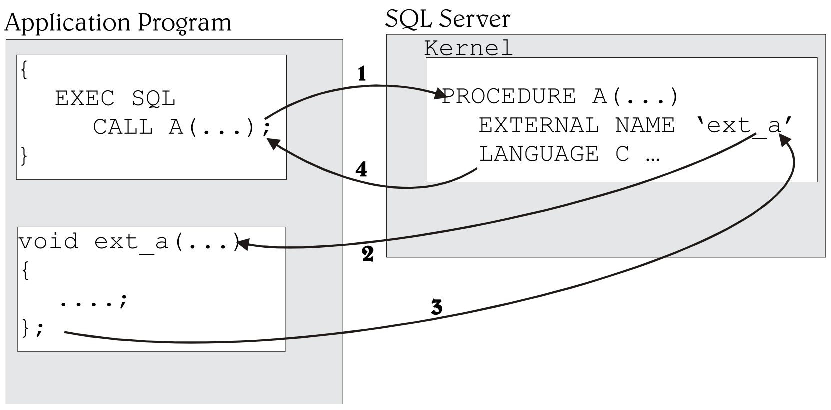
```
PROCEDURE A(...) AS
BEGIN
   ....;
END;
```

When an application program makes a call to a stored procedure A with a set of parameters, the call is passed to the SQL server's kernel, which passes them on to the actual, external procedure. Since the procedure runs in the same memory space as the kernel, the parameter passing between the procedure and the kernel can be done very efficiently.

The drawback of this solution is that bug infected procedures may corrupt the kernel of the database server. If, for instance, the index of an array reference is out of bounds and this situation is allowed in the execution environment for the procedure, an erroneous write operation could bring down the whole database system.

## In the application program

In this approach, the external procedure actually runs in the same memory space as the application program.



```
{
   EXEC SQL
      CALL A(...);
}
```

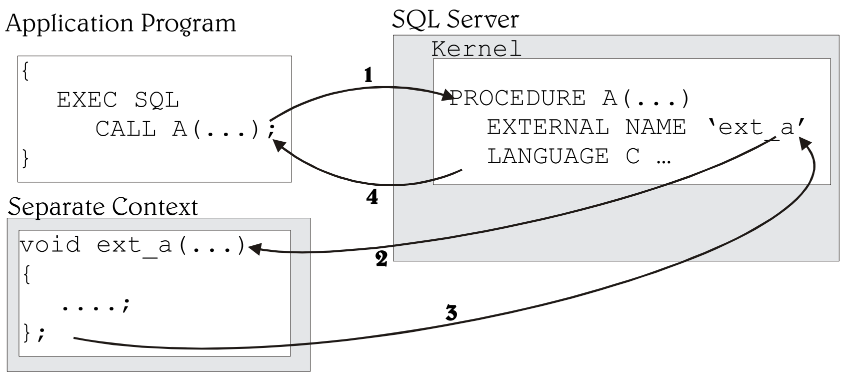
```
PROCEDURE A(...)
   EXTERNAL NAME 'ext_a'
   LANGUAGE C …
```

```
void ext_a(...)
{
   ....;
};
```

The application program makes a call to a stored procedure A registered in the database server. The database server sees that this procedure is an external procedure ext-a, and executes this through some kind of callback mechanism to the application program's execution environment. This zigzagging can repeat several times, dependent on the interrelationship between procedures in the application program and the database server. Each context switch, with the implicated parameter passing, is a real performance challenge for the execution environment. If these context switches also include sending packages across some network connection, the overhead will be even bigger. However, this approach protects the runtime environment of the database server from any bugs in the stored procedures implementation.

### In a separate context

The last approach is to even keep the external routine in a separate memory space.



As the illustration above shows, the number of context switches is the same as for the previous approach, but the operating system has to handle one more context. This approach gains protection of both the database runtime context and the application runtime context.

## 3.5. Operators

Some object-oriented programming languages (such as `C++`) provide means for the programmer to define and/or overload operators. This could also prove to be useful in ORDBMs. As is well known for programmers familiar with user-defined operators, these are nothing more than methods with special syntaxes. If a binary `+` operator is defined on a `C++` class `c`, taking an object of class `c` as both its arguments and resulting in an object of class `c`, this would be used like this:

```
{
      ....  ;
      x = y + z;
      ....  ;
}
```

This would be equivalent to having a method `+( c i , c j )`:

```
{
      ....  ;
      X.+(y , z);
      ....  ;
}
```

Using this paradigm, a user-defined operator in an ORDBMS could simply be a registration of an operator against an already defined user-defined function[29]:

[29] As is how it is done in the database system formerly known as Illustra.

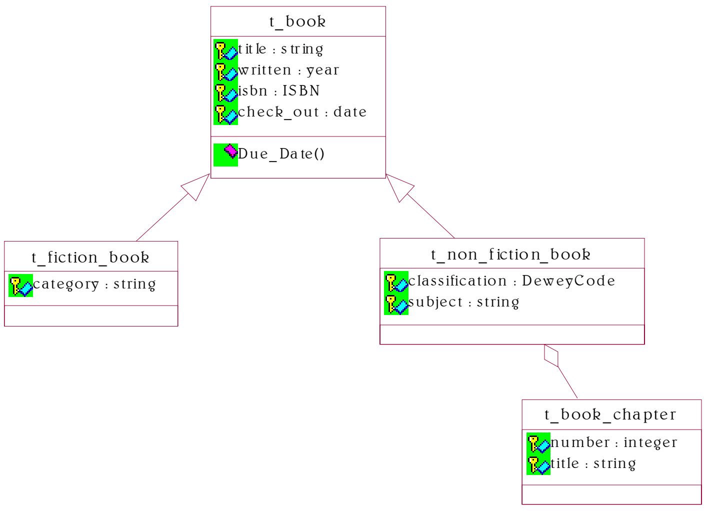
```
        create operator
             binding operator_name to functionnatne;
```

User-defined operators are not considered a part of SQL-99. Furthermore, they are not presented as an important issue in (Stonebraker & Moore 1996).

## 3.6. Inheritance

No class concept has been introduced into the ORDBMS description, and inheritance is usually associated with classes. In an ORDBMS inheritance should apply to user-defined types, or abstract data types to be more specific.

In the library database, it is useful to distinguish between fictional books and non-fictional books partly since they often are categorised differently. Assume that information about a book's chapters is valuable only for non-fictional books. This could be expressed in UML as:



This could naturally be implemented in a traditional RDBMS, without any inheritance functionality, by means of either two entirely independent tables, or one single table containing all attributes for `t_book`, `t_fiction_book` and `t_non_fiction_book`. In an ORDBMS however, this can be implemented like this:

```
        create type t-book
        (
             title                    string,
             written                  year,
             isbn                     ISBN,
             check_out                date,
             function Due_Date()      return date
        );

        create type t_book_chapter
        (
             number                   integer,
             title                    string
        );

        create type t_fiction_book
        (
             category                 string
        ) under t_book;
```

```
create type t_non_fiction_book
(
     classification       DeweyCode,
     subject              string
) under t_book;
```

It can be argued whether an ORDBMS should support multiple inheritance or not. Providing this feature certainly empowers the modelling possibilities, but then again, there are always the usual problems such as name conflicts, etc.

As can be seen from the UML diagram above, a function (or method) `Due_Date ()` is defined on the `t_book` type. It is reasonable to expect that this function is made available to `t_book`'s heirs by means of inheritance.

## 3.7. Constraints

Having a good data model is all very fine. However, it is no good if the actual data stored in the database is not reliable. Therefore, every ORDBMS should have some means to enforce data quality. This not only goes for ORDBMSs; the same aspects are just as important to more traditional RDBMSs. In SQL-99, data quality is the concern of a set of schema objects called integrity constraints. These are described in detail in (Türker & Gertz 2001).

In general, constraints enforcement can be seen to have one of three granularities, or constraints levels:

Row constraints:
These constraints works on a single row in a single table.

Table constraints:
These constraints works on two or more rows within the same table.

Inter-table constraints:
This is the most general constraint level, and operates on one or more rows for two or more tables.

Looking at constraints from a different angle, the constraint enforcement is differentiated according to how many and which database states are involved when the constraints are enforced:

Single state constraints:
Constraints are evaluated for a single database state. This means that the constraints are used to check that its conditions are fulfilled as the database enters a new state.

State transition constraints:
These constraints are used to compare to consecutive database states. In general, these constraints are of the form:

```
if ( old state satisfied condition1 ) then
     new state must satisfy condition2;
```

Typically `condition2` has some functional dependency on `condition1`.

State sequence constraints:
These constraints are used to compare the new database state that is sought to be entered with some state that the database has been in sometime in the past.

### 3.7.1. SQL-99 Language Constructs

The support for constraints in SQL-99 can roughly be divided into four groups:

Simple constraints:
These are constraints that can be seen as part of the structure of a table. They are:

- `CHECK`:
  A check constraint enforces that a condition involving one or more values within a single row of a table. This is a row constraint.

- `NOT NULL`:
  This can be regarded as a special case of a `CHECK` constraint. The condition always involves a single value, and enforces that that value is never `NULL`.

- `DEFAULT`:
  This condition makes sure that if a specific value in a row is set to `NULL`, it gets a default value instead. It is thus a `NOT NULL` constraint with an assignment action.

- `PRIMARY KEY`:
  The primary key of a table is enforced by the single `PRIMARY KEY` constraint of the table. This is a table constraint.

- `UNIQUE`:
  This constraint enforces that the value(s) of the specified attribute(s) are unique within the whole of the table. Alternate keys are defined by means of `UNIQUE` constraints. This is a table constraint.

- `FOREIGN KEY`:
  A constraint of this type enforces that before the value(s) of the involved attribute(s) are set, corresponding value(s) must already exist in the primary key of the referenced table. This is a table constraint.

`DOMAIN`:
These constraint are describe on page 28.

`ASSERTION`:
An assertion is an inter-table check constraint. The assertion definition will contain a search condition involving one or more tables. The assertion is satisfied when the search condition evaluates to true.

`TRIGGERS`:
Triggers are the realisation of event-condition-action (ECA) rules. Triggers are (among other things) used to enforce inter-table constraints.

## Constraint modes

All constraints can be declared to have a specific checking mode. A checking mode defines when the constraints should be enforced. A constraint can be made `DEFERRABLE` allows for a choice of whether the time of checking should be at the end of statement execution (`IMMEDIATE`) or at the end of transaction execution (`DEFERRED`). A constraint that is `NOT DEFERRABLE` will always have check time `IMMEDIATE`.

## Foreign key actions

A foreign key constraint can have actions associated with it. These actions are executed when and if a change to the value(s) of the foreign key attribute(s) causes the constraint to be violated. The events that can cause a violation are `UPDATE` and `DELETE`. A foreign key can have one action for each of these events. The actions defined in SQL-99 are:

`SET DEFAULT`:
This action sets the value(s) of the foreign key attribute(s) to its/their default values. This action requires the existence of `DEFAULT` constraint(s). If the referenced table does not have a primary key value corresponding to the given default values, this action fails.

`SET NULL`:
This action sets the value(s) of the foreign key attribute(s) to `NULL`. This action requires the non-existence of `NOT NULL` constraint(s).

`CASCADE`:
If the event is `UPDATE`, this action causes the value of the corresponding primary key to be updated according to the updates of the foreign key value. If the event is `DELETE` this action causes the corresponding rows in the referenced table to be deleted.

`RESTRICT`:
This action cancels the involved `UPDATE`/`DELETE` event unconditionally.

`NO ACTION`:
This action cancels the involved `UPDATE`/`DELETE` event if a violation results.

## Constraint scopes

A reference column in a typed table (see page 29) can be given a `SCOPE`. This is an analogue to `FOREIGN KEY` constrains for ordinary columns. A `SCOPE` states whether a reference is checked or not. If a reference is to be checked, the `SCOPE` must specify an action to be executed in the case of a `DELETE` event. The legal actions are the same as for `FOREIGN KEY` constraints.

## Inheritance of constraints

If a constraint is defined for an abstract data type, that constraint also holds for all typed tables defined on that type, and for all attributes of that type.

In both a type hierarchy and a table hierarchy constraints are inherited. This means that if a constraint holds for a specific type or table, it also holds for any sub-types or sub-tables derived from it. Constraints cannot be overridden.

### Triggers

Triggers are needed to enforce more complex data quality regulations. An event is the result of one of the SQL commands `INSERT`, `UPDATE`, `SELECT` or `DELETE`. A trigger can be specified to fire `BEFORE` or `AFTER` the event is applied to the database. SQL-99 differentiates between row-level triggers and statement-level triggers. If a trigger is to be a row-level trigger, this is indicated by a `FOR EACH ROW` clause. Otherwise, and default, a `FOR EACH STATEMENT` clause may be used. A row level trigger is fired for each row affected by the triggering event, whereas a statement-level trigger is fired only once. A trigger can be also be specified to execute only when a given condition is fulfilled.

An action will typically be some other SQL command. Alternatively, it could be some stored procedure or function, possibly implemented in a language more expressive that SQL. If the trigger action is implemented in a language that supports some form of an `IF...THEN...ELSE` structure, the support of trigger conditions are not important. The condition, or conditions, can then be programmed into the action part of the trigger.

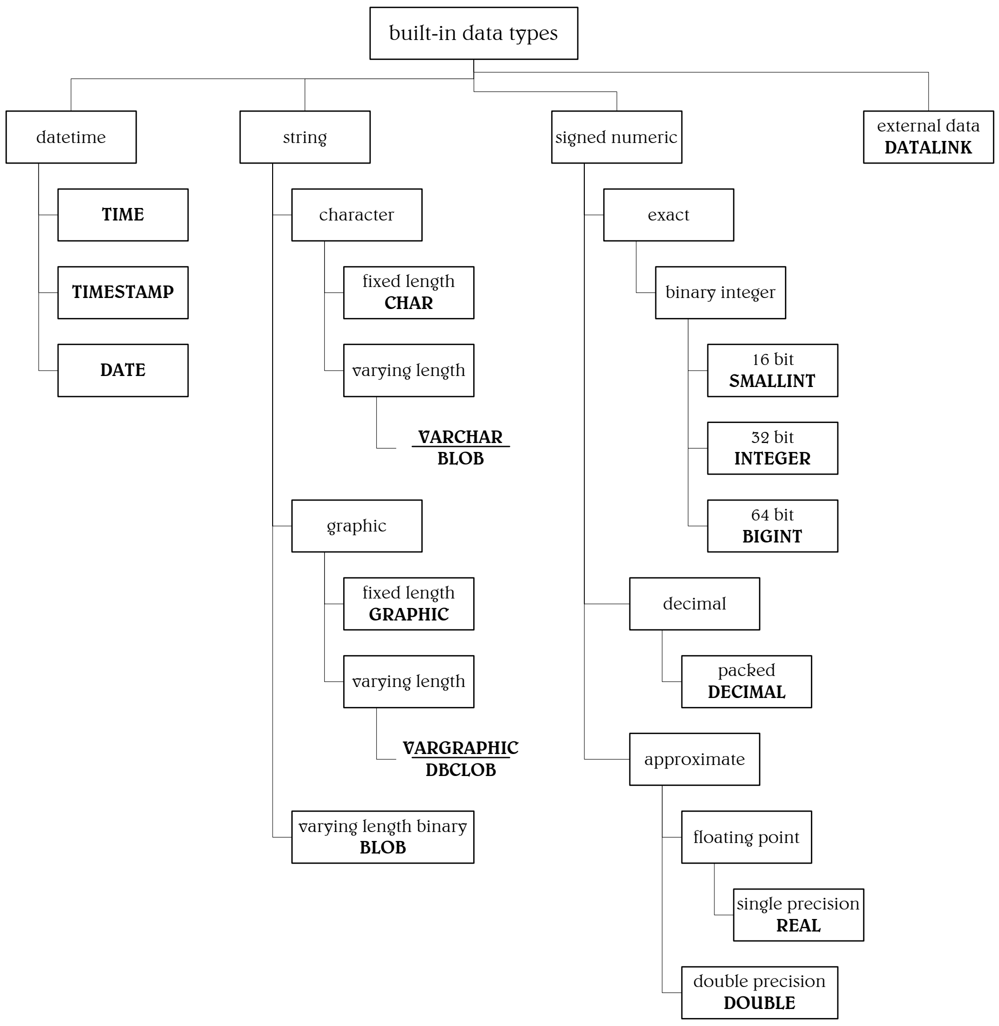
# 4. How Object-Relational is DB2?

OF COURSE, IBM WILL SUPPORT EVERY VERSION OF SQL.
AFTER ALL, WE INVENTED IT!
**DONALD CHAMBERLAIN**, *PERSONAL COMMUNICATIONS*

This chapter will follow much the same steps as chapter 2 did. The main exception is compliance to the creeds and propositions described in the beginning of chapter 2. This will be dealt with at the end of this chapter. As the syntax of the various SQL statements are very complex, the syntactical descriptions given throughout this chapter are stripped to the bare necessities, only describing or illustrating the point discussed.

## 4.1. Extended Base Type Set

Base types are also called scalar types. These are the holders of atomic values. They can all hold (or be assigned) a null value. At the turn of the millennium, the set of base types has grown rather large, so it makes sense to divide them into groups. But first, an overview of all the types in the base type set and their inter-relationships:



### 4.1.1. Large Object Types

| Type | Max size | Description |
|---|---|---|
| `CLOB` | 2 GB | Large character string consisting of single byte characters of up to the specified maximum length in bytes. |
| `DBCLOB` | 1 GB | Large character string consisting of double byte characters of up to the specified maximum length in bytes. A `DBCLOB` is regarded as a graphic string. |
| `BLOB` | 2 GB | Large binary object string of the specified maximum length in bytes. Intended to hold non-traditional data such as pictures, sound, video, etc. `BLOB` is not associated with a character set. |

All of these types will be referred to by the common denominator LOB. A LOB is usually too large to be transferred between the database and the application program as a whole. The most common approach is to transfer it piece by piece through a special host variable called a LOB locator. During transactions, the only way to refer to a LOB value is through a LOB locator.

DB2 allows LOBs (and long character fields) to be placed in a designated `LONG` tablespace. Furthermore, LOBs cannot be part of a `WHERE` clause or an `ORDER` clause in an SQL statement.

### 4.1.2. Character Strings

A character string is a sequence of bytes. This is primarily used with single-byte character strings.

| Type | Max size | Description |
|---|---|---|
| `CHAR` | 254 bytes | Fixed length character string. |
| `VARCHAR` | 32 Kb | Variable length character strings. Special restrictions apply when the maximum width of a row exceeds limits which are dependent on the page size of its tablespace: |
| `LONG VARCHAR` | The same as `VARCHAR(32Kb)` | |
| `CLOB` | 2 GB | See above |

| Page size | Row size limit |
|---|---|
| 4K | 4 005 bytes |
| 8K | 8 101 bytes |
| 16K | 16 293 bytes |
| 32K | 32 677 bytes |

### 4.1.3. Graphic Strings

A graphic string is a sequence of bytes representing double-byte character data. This is used for instance when storing text in some Asiatic writing (e.g. Chinese or Cantonese)

| Type | Max size | Description |
|---|---|---|
| `GRAPHIC` | 127 bytes | Fixed length graphic string. |
| `VARGRAPHIC` | 16 Kb | Variable length strings of double-byte characters. Similar restrictions apply to these as to `VARCHAR` and `LONG VARCHAR`. |
| `LONG VARGRAPHIC` | The same as `VARGRAPHIC (16Kb)` | |
| `DBCLOB` | 1 GB | See above |

The database manager will always assume that whatever character is inserted into a graphic string is a double-byte character. A single-byte character may be inserted without any validation preventing it. If this single-byte character is followed by a double-byte character $c_d$, the first byte of $c_d$ will be appended to the preceding single-byte character while the second byte of $c_d$ will be appended to the string as new a single-byte character.

However, the database manager will check that the whole of the graphic string contains an even number of characters when committed to the database. This means that if the application inserts a single-byte character, it is also the application's responsibility to rectify this bias.

### 4.1.4. Numbers

| Type | Min. | Max. | Size |
|---|---|---|---|
| `SMALLINT` | -32768 | 32767 | 2 bytes |
| `INTEGER` | -2147483648 | 2147483647 | 4 bytes |
| `BIGINT` | -9223372036854775808 | 9223372036854775807 | 8 bytes |
| `REAL` | -3.402e+38 | 3.402e+38 | 4 bytes |
| `DOUBLE/FLOAT` | -1.79769e+308 | 1.79769e+308 | 6 bytes |
| `DECIMAL/NUMERIC` | $-10^{31}+1$ | $10^{31}+1$ | |

### 4.1.5. Date and Time Values

| Type | Size | Description |
|---|---|---|
| `DATE` | 4 bytes | Date as a three-part value (year, month and day). Year ranges from 000 1 to 9999.[30] |
| `TIME` | 3 bytes | Time as a three-part value (hour, minutes and seconds) according to a 24H clock. |
| `TIMESTAMP` | 10 bytes | Date and time as a seven-part value (year, month, day, hour, minutes, seconds and microseconds). Ranges are as for DATE and TIME. |

### 4.1.6. Data Link Values

A `DATALINK` value is an encapsulation of a logical reference from the database to a file stored outside the database. When defining a column with data type `DATALINK`, several attributes needs to be set:

| Attribute | Description |
|---|---|
| link type | As of UDB 7.2, only URL is allowed as link type.<br><br>The other parts of the URL are:<br><br>• the file server name for the HTTP, FILE and UNC schemes<br><br>• the cell name for the DFS scheme<br><br>• the full file path name within the file server or cell |
| schema | As of UDB 7.2,: HTTP, FILE, UNC or DFS |
| comment | Up to 254 bytes of descriptive information |

## 4.2. Distinct Types

The very first user-defined data types that DB2 supported was distinct types. The syntax for creating a distinct type is:

```
        CREATE DISTINCT TYPE type-name AS source-data-type [WITH COMPARISONS]
```

The `WITH COMPARISONS` option specifies that system-generated comparison operators (to compare two values of the newly defined data type) are to be generated. This option

[30] This means that DB2 UDB does not handle a potential Y20K problem. ☺

is illegal for all distinct types based on `LOBS`, `LONG VARCHAR`, `LONG VARGRAPHIC` and `DATALINK`. For every other source data types, this option is mandatory.

Creating a distinct data type `CURRENCY` based on `REAL` is done like this:

```
CREATE DISTINCT TYPE CURRENCY AS REAL WITH COMPARISONS;
```

Distinct types are a stripped-down implementation of domain types. The distinct type definition has no support for default values, constraints or collation rules.

### 4.2.1. Repository Impact

When the distinct type `CURRENCY` above is created, this results in changes to 3 system tables' data.

One row being added to the `SYSIBM.SYSDATATYPES` table, with the following fields set (among others such as defining schema and creation date) according to the table to the right.

| Field: | Value: |
|---|---|
| NAME | CURRENCY |
| SOURCETYPE | REAL |
| SOURCESCHEMA | SYSIBM |
| METATYPE | T |
| TYPEID | -32767 |
| SOURCETYPEID | 10 |

`SOURCETYPEID` is a recursive foreign key. `REAL` has 10 as the type identification number. `TYPEID` is assigned by the database manager. The very first user-defined distinct type is given the value -32767, and the type id for subsequent user defined distinct types is always set to one larger than the previous type id used. This implies a limit of approximately 32K user defined data types. This should be more than enough for most data models, but could prove, in some cases, to be an annoying limit for some models.

In the repository table `SYSIBM.SYSFUNCTION`, nine new functions have been created. These new functions are operator functions and type cast functions[31]:

| Function name: | Parameter list: | Result type: |
|---|---|---|
| = | CURRENCY,CURRENCY | BOOLEAN |
| < | CURRENCY,CURRENCY | BOOLEAN |
| > | CURRENCY,CURRENCY | BOOLEAN |
| <= | CURRENCY,CURRENCY | BOOLEAN |
|  | CURRENCY,CURRENCY | BOOLEAN |
| <> | CURRENCY,CURRENCY | BOOLEAN |
| CURRENCY | REAL | CURRENCY |
| CURRENCY | DOUBLE | CURRENCY |
| REAL | CURRENCY | REAL |

[31] A complete description of CAST functions is given in the (IBM DB2 SQL Reference Guide)

The parameter types and the result type is defined in the system table SYSIBM.SYSFUNCPARMS.

### 4.2.2. Usage

Creating two distinct types:

```
create distinct type ohm as double with comparisons;
create distinct type amphere as double with comparisons;
```

and a table based on these:

```
create table electricity
(
     ID         integer,
     resistance ohm,
     current    amphere
);
```

gives the following table realization in DB2:

```
Column             Type        Type
name               schema      name               Length   Scale    Nulls
------------------ ----------- ------------------ -------- -------- --------
ID                 SYSIBM      INTEGER                   4        0 Yes
RESISTANCE         TEST        OHM                       0        0 Yes
CURRENT            TEST        AMPHERE                   0        0 Yes
```

With these type definitions in place, running this query:

```
select resistance*current voltage from electricity;
```

causes DB2 UDB to give a negative response:

```
=> SQL0440N  No function by the name "*" having compatible arguments
was found in the function path.  SQLSTATE=42884
```

To do this multiplication, the following solves the problem:

```
create distinct type volt as double with comparisons;
select volt(double(resistance)*double(current)) voltage
from electricity;
```

To detour via the `volt` data type is actually not needed to get the multiplication done, since it is definitely possible to write the `select` statement so that it simply returns a `double` as result. However, it is more in the spirit of strong typing to create the `volt` data type.

## 4.3. Structured Types

Structured types are created by the `CREATE TYPE` statement. The syntax for this is:

```
CREATE TYPE type_name [UNDER supertype_name] AS
(
      attribute list
)
      [ NOT INSTANTIABLE ]
      [ WITHOUT COMPARISONS ]
      [ NOT FINAL ]
      MODE DB2SQL
      [ REF USING < ref-type > ]
      [ <method specifications> ];
```

The `UNDER supertype_name` clause is the means by which inheritance is defined. As can be seen from the syntax description, only single inheritance is provided for. Other supplementary clauses are described below. The attribute list is defined mainly in the same way as the attribute list for tables. Attributes that are part of a type can be of a data type that either belongs to the base type set, are a distinct type (i.e. a synonym type), or a reference to another structured type. It is worth noting that recursive (or nested) type definitions are not allowed.

| Clause | Description |
|---|---|
| `NOT INSTANTIABLE` | When a structured type is specified to be `NOT INSTANTIABLE`, no constructors is generated for this type. It cannot be used as basis for a typed table[32]. Such type fulfils much the same function as virtual classes in C++ and Java. It can, however be used as the type of a column. A column of a `NOT INSTANTIABLE` type can only be given a NULL value or a value of one of its types' subtypes. |
| `WITHOUT COMPARISONS` | This clause results in the data type to be realised without any comparison functions/operators being generated. |
| `NOT FINAL` | This clause indicates that this type may be used as a supertype. |
| `MODE DB2SQL` | This indicates that the data type is defined in DB2 SQL mode[33]. |
| `REF USING < ref-type >` | This clause specifies which built-in data type that is to be used as the reference type[34] for this structured type and all its subtypes.<br><br>This clause can only be specified for the root type of a structured type hierarchy.<br><br>There are some limitations as to which built-in data type can be used as a reference type. For more information see (IBM DB2 SQL Reference guide). |

[32] See below.

[33] What modes the future has in store will be very interesting to see.

| | |
|---|---|
| | Default value for this option is `REF USING VARCHAR(16) FOR BIT DATA`. |
| `method specifications` | This clause defines the methods for the structured type. For a short description of the syntax, see the example below. For complete details one the syntax, see (IBM DB2 SQL Reference guide). |

### 4.3.1. Repository Impact

To define a structured data type person with attributes for first name, last name, date of birth and a method that calculates a persons age, the command would be:

```
CREATE TYPE PERSON AS
(
      FIRSTNAME  VARCHAR(50),
      LASTNAME   VARCHAR(50),
      BORN_ON    DATE
)
      WITHOUT COMPARISONS NOT FINAL MODE DB2SQL
METHOD AGE()
      RETURNS INTEGER
      LANGUAGE SQL
      NOT DETERMINISTIC
      CONTAINS SQL
      NO EXTERNAL ACTION;
```

The very long (and cumbersome) declaration of method AGE needs some explanation:

| Clause | Explanation |
|---|---|
| `RETURNS <data type>` | This specifies the data type returned from the method. Every data type described above is allowed as result type. |
| `LANGUAGE SQL` | This clause is used to indicate that the method is written in SQL with a single RETURN statement. |
| `[NOT] DETERMINISTIC` | This optional clause specifies whether the method always returns the same results for given argument values (`DETERMINISTIC`) or whether the method depends on some state values that affect the results (`NOT DETERMINISTIC`).<br><br>**Example:** The method `AGE()` defined above is not deterministic as the result is has no input arguments and the result of the method is dependent on a value stored in a `PERSON` object. A method `CAPITALISE( ... )` defined as<br><br>`METHOD CAPITALISE ( IN_STR IN VARCHAR )`<br>`RETURNS VARCHHAR`<br>`LANGUAGE SQL`<br>`DETERMINISTIC` |

[34] The data type of an objects OID.

| | |
|---|---|
| | `NO EXTERNAL ACTION;`<br><br>that simply returns its input capitalised will return the same result independent of attribute values. Since the result of `CAPITALISE( ... )` is dependent only of its arguments, this is a `DETERMINISTIC` method. |
| `CONTAINS SQL` | Indicates that SQL statements that neither read nor modify SQL data can be executed by the method. |
| `[NO] EXTERNAL ACTION` | This optional clause specifies whether the method takes some action that changes the state of an object not managed by the database manager. |

The above only declares that the `person` type is to have a method called `age()`. The actual implementation of the method is given in a separate SQL99 statement:

```
create method age()
     for person
     return ( ( current date ) - self..born_on ) / 365;
```

This is stored in the `SYSIBM.SYSFUNCTIONS` and the `SYSIBM.SYSFUNCPARAMS` tables the same way as the other functions and operators that the type owns. Since the `age()` method uses the `born_on` attribute of the `person` type, a dependency between the method `age()` and the function `born_on()` is created. This dependency is registered in the `SYSIBM.SYSDEPENDENCIES` table.

### 4.3.2. Usage

Structured types can be used as any other type; i.e. as table attribute types, procedure parameter types, etc. They can even be used as attribute types in other structured types. In addition, they may be used directly in table definitions:

```
CREATE TABLE
     table-name OF type-name
     ( REF IS oid-column-name USER GENERATED );
```

As in any use of the `CREATE TABLE` command, far more complexity may be used. The above is just an extract to illustrate this use of structured types.

The `oid-column-name` indicates that an object identifier column is to be defined as the first column of the table. This column will be of type `REF( type-name )`. To have an `oid` column, the table must be a type-based table that *is not* a sub-table.

## 4.4. Functions

DB2 provides three different types of user-defined functions:

External Scalar

External Table

Sourced

The differentiation between these functions is dependent on how they are defined (or described) to DB2, and on what they return.

### 4.4.1. External Scalar Function

An external scalar function is written in some programming language[35] external to DB2, or is an OLE object[36]. It returns a single scalar result.

In general, a reference to an external scalar function is legal wherever an SQL expression is legal.

**Example:** Assume there already exist a C function that takes the ISBN of a book as input and finds that book's Dewey classification code by searching certain servers on the Internet. This function would be very useful in a library database, and could be made available to DB2 by declaring the following external scalar function:

```
CREATE FUNCTION DEWEY CODE (VARCHAR)
     RETURNS VARCHAR
     EXTERNAL NAME 'librarian!deweyCode'
     LANGUAGE C
     PARAMETER STYLE DB2SQL,
     NOT DETERMINISTIC
     NOT FENCED
     RETURNS NULL ON NULL INPUT
     NO SQL;
```

This establishes a link to an external C function `deweyCode` that resides in the library file `librarian.lib`. Special operating system dependent rules decides where DB2 looks for the library file. The absolute library path may also be used.

The option `PARAMETER STYLE DB2SQL` is used to specify the conventions for passing parameters to and returning the value from external functions that conform to C language calling and linkage conventions.

The option `NOT DETERMINISTIC` says that this function is not guaranteed to always return the same result given the same input parameter[37]. Thus, the DB2 server cannot cache the result.

The option `NOT FENCED` informs the DB2 server that this function does not interfere with the database managers operating environment.

The option `RETURNS NULL ON NULL INPUT` indicates that the function will return a `NULL` value if the input parameter is `NULL`.

The mandatory option `NO SQL` indicates that the function cannot issue any SQL statements. The existence of this option suggests that future versions of DB2 will allow for the function to issue SQL statements.

[35] As of version 7.2, the programming languages supported are C and Java.

[36] 32-bits MS-Windows platforms only.

[37] The task of classifying a book always involves a great deal of individual judgement by the librarian performing the classification. Therefore, a classification can be altered due to misunderstanding and errors.

### 4.4.2. External Table Function

The difference between an external table function and an external scalar function is in the produced result. An external table function will return a row set result. Thus, it can be used in the `FROM` clause (or any other place where a row set is expected) of a `SELECT` statement, just as if it were a table or a view.

**Example:** Assume there exists a C function `getReferencingBooks(char* isbn)` that takes the `ISBN` of a book as input and finds the title, the authors and the ISBN of every book referencing it by searching certain servers on the Internet. This function can be wrapped in a C function that concurs with the specifications for DB2 stored procedures[38]:

```
void SQL_API_FN DB-getReferencingBooks(
          /* Input field */
          SQLUDF_VARCHAR in_isbn,
          /* Return row fields */
          SQLUDF_VARCHAR *title,
          SQLUDF_VARCHAR *authors,
          SQLUDF_VARCHAR *isbn,
          /* Input null indicatior */
          SQLUDF_NULLIND null_in_isbn,
          /* Return row field null indicators */
          SQLUDF_NULLIND *null_title,
          SQLUDF_NULLIND *null_authors,
          SQLUDF_NULLIND *null_isbn,
          /* UDF always-present (trailing) input arguments */
          SQLUDF_TRAIL_ARGS_ALL
          )
{
     if ( ! null_in_isbn )
     {
          /* Call the original function */
          getReferencingBooks( .... );
          /* And prepare the results for the UDF */
          .....;
     }
}
```

This function could be made available to DB2 by declaring the following external table function:

```
CREATE FUNCTION GET_REFERENCING_BOOKS( VARCHAR(10) )
     RETURNS TABLE
     (
          TITLE      VARCHAR(100),
          AUTHORS    VARCHAR(I00),
          ISBN       VARCHAR(10)
     )
     NOT FENCED
     NOT DETERMINISTIC
     NO SQL
     NO EXTERNAL ACTION
     LANGUAGE C
     PARAMETER STYLE DB2SQL
     EXTERNAL NAME 'librarian!DB_getReferencingBooks';
```

[38] This format is describe in detail in (IBM DB2 Application Development Guide) in the chapter *"Writing User-Defined Functions (UDFs) and Methods"* and *" Examples of UDF Code"*.

The option NO EXTERNAL ACTION specifies that the function in question does not do anything that can change the state of any object that is managed by the database manager.

For all the other options, see the descriptions in the previous example.

### 4.4.3. Sourced Function

Sourced functions are the only functions that are defined and registered entirely within the DB2 server. A sourced function is used to define a function that is based on another, already existing, scalar or column[39] function. This may be used to create functions that take distinct types as parameters, without having to write the implementation of the function all over again.

Assume the types described on page 51 are defined, and there is a need to calculate average resistance in the electricity table. Simply giving the statement

```
SELECT AVG(RESISTANCE) FROM ELECTRICITY;
```

would result in an error:

```
SQL0440N  No function by the name "AVG having compatible arguments
was found in the function path. SQLSTATE=42884
```

No function avg that takes a parameter of type ohm exists. This can be solved by casting the resistance column to double:

```
SELECT AVG(DOUBLE(RESISTANCE)) FROM ELECTRICITY;
```

This becomes clumsy and not very readable. Defining a sourced function is another solution:

```
CREATE FUNCTION AVG( OHM )
     RETURNS (OHM)
     SOURCE SYSIBM.AVG( DOUBLE );
```

With this function in place, the type casting in the SELECT statement is no longer needed, and the first SELECT statement becomes legal.

## 4.5. Procedures

Like functions, procedures are implemented in an external programming language[40]. From an SQL syntax point of view, a procedure is equivalent to a function without any return value. Curiously, however, a procedure can return a cursor to a result set. This indicates that the user in some situations is free to use procedures and table functions interchangeably.

[39] A column function is often referred to as an aggregate function.

[40] As of version 7.2, the programming languages supported are C and Java.

## 4.6. Operators

From what is written in sections 4.2 and 4.3 on distinct types and structured types, one would assume that support for user-defined operators is a feature in DB2 UDB. However strange it may seem, this is not so. The only operator extensions available are the comparison operators that may implicitly be generated as a by-product of a distinct type and those that are always generated when a structured type is created.

## 4.7. Encapsulation

There is not much to say about encapsulation in DB2. This is simply because there are none.

Naturally, it is possible to create column access procedures and functions, but the columns themselves will remain just as exposed to direct manipulation as ever.

## 4.8. Collection Constructors

When it comes to being object-relational, or SQL-99 compliant, the weakest point in DB2 UDB is the lack of collection constructors. According to internal personnel in IBM, it was IBM's initial intention to include collection constructors in DB2 UDB. Still, This was given a low priority. When Michael Carey left his position as head of the UDB team in 2000, his successor decided that collection constructors should not be included in DB2 UDB, at least not for quite a while.

## 4.9. Inheritance

### 4.9.1. Type Inheritance

As has already been mentioned, DB2 supports type inheritance through the `CREATE TYPE` statement:

```
CREATE TYPE type_name [UNDER supertype_name] AS
(
     attribute list
) WITHOUT COMPARISONS
NOT FINAL
MODE DB2SQL;
```

This gives rise to a single-inheritance hierarchy. The supertype, from which a new type inherits its attributes, must already exist in the schema of the new type. Alternatively, if the supertype exists in a different schema, the supertype name must be prefixed with the schema name. This is according to standard SQL naming rules.

The `NOT FINAL` clause indicates that the type created may be used as a supertype. As described above, this clause is mandatory as of version 5.2. Nevertheless, the presence of this clause announces that some future version of DB2 may include the option where a type can be restricted from being used as a supertype.

### 4.9.2. Table Hierarchies

In addition to allowing for the creation of a type inheritance, DB2 also supports a table inheritance mechanism through one of the variations of the `CREATE TABLE` statement:

```
CREATE TABLE table_name
     OF type_name
     UNDER supertable_name INHERIT SELECT PRIVILEGES;
```

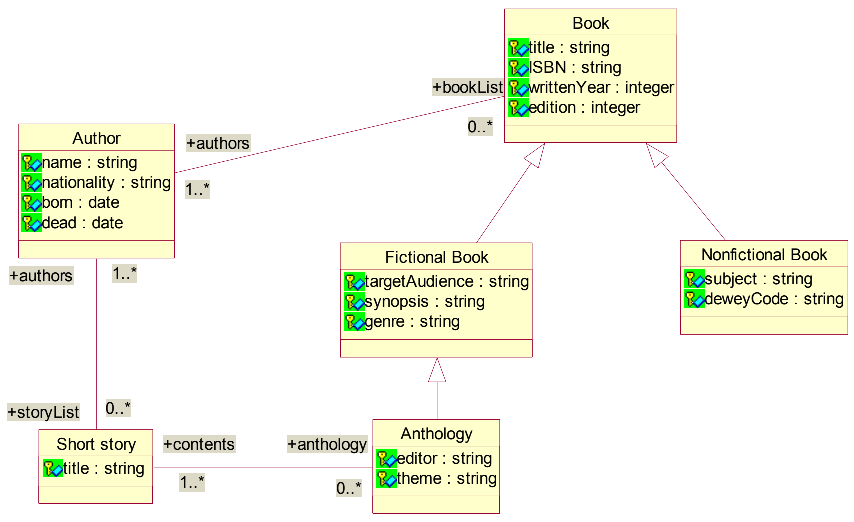
Such a statement will indicate that the new table `table_name` will become a subtable of `supertable_name`.

The supertable must be an existing table and must have been defined using a structured type that is the immediate supertype of `type_name`. The schema of `table_name` and `supertable_name` must be the same.

The resulting table will have columns based on the attributes of `type_name`, plus the object identifier column of the supertable with type modified to `REF(type_name)`.

The mandatory `INHERIT SELECT PRIVILEGES` clause indicates that any user who has a `SELECT` privilege on the supertable will be granted the same privilege on the new table. In addition to having all select privileges inherited, all storage related parameters of the table are inherited. Thus, it is not possible to specify a tablespace for the subtable.

**Example:** For the illustration of this, consider the class model described in the following UML diagram:



In DB2 this model is expressed as:

```
CREATE TYPE T_AUTHOR AS
(
     NAME            VARCHAR(100),
     NATIONALITY     VARCHAR(50),
     BORN            DATE,
     DEAD            DATE
) MODE DB2SQL REF USING INTEGER;
```

```
CREATE TYPE T_BOOK AS
(
     TITLE           VARCHAR(100),
     ISBN            VARCHAR(20),
     WRITTENYEAR     INTEGER,
     EDITION         INTEGER
) MODE DB2SQL REF USING INTEGER;

CREATE TYPE T_FICTIONAL_BOOK UNDER T_BOOK AS
(
     TARGET_AUDIENCE VARCHAR(100),
     SYNOPSIS        VARCHAR(1000),
     GENRE           VARCHAR(40)
) MODE DB2SQL;

CREATE TYPE T_NONFICT_BOOK UNDER T_BOOK AS
(
     SUBJECT         VARCHAR(100),
     DEWEY_CODE      VARCHAR(10)
) MODE DB2SQL;

CREATE TYPE T_ANTHOLOGY UNDER T_FICTIONAL_BOOK AS
(
     EDITOR          VARCHAR(100),
     THEME           VARCHAR(50)
) MODE DB2SQL;

CREATE TYPE T_SHORT_STORY AS
(
     TITLE           VARCHAR(100)
) MODE DB2SQL REF USING INTEGER;

CREATE TABLE BOOK
     OF T_BOOK
     ( REF IS ID USER GENERATED );

CREATE TABLE AUTHOR
     OF T_AUTHOR
     ( REF IS ID USER GENERATED );

CREATE TABLE SHORT_STORY
     OF T_SHORT_STORY
     ( REF IS ID USER GENERATED );

CREATE TABLE FICTIONAL_BOOK
     OF T_FICTIONAL_BOOK
     UNDER BOOK INHERIT SELECT PRIVILEGES;

CREATE TABLE NONFICTIONAL_BOOK
     OF T_NONFICT_BOOK
     UNDER BOOK INHERIT SELECT PRIVILEGES;

CREATE TABLE ANTHOLOGY
     OF T_ANTHOLOGY
     UNDER FICTIONAL_BOOK INHERIT SELECT PRIVILEGES;

CREATE TABLE BOOK_AUTHOR
(
     BOOKLIST   REF( T_BOOK )             NOT NULL,
     AUTHORS    REF( T_AUTHOR )           NOT NULL,
     CONSTRAINT PK_BOOK_AUTHOR PRIMARY KEY ( BOOKLIST , AUTHORS )
);
```

```
CREATE TABLE SHORTSTORY_AUTHOR
(
     STORYLIST  REF( T_SHORT_STORY )       NOT NULL,
     AUTHORS    REF( T_AUTHOR )            NOT NULL,
     CONSTRAINT PK_SHSTORY_AUTHOR PRIMARY KEY ( STORYLIST , AUTHORS
)
);

CREATE TABLE SHORTSTORY_ANTHOLOGY
(
     CONTENTS   REF( T_SHORT_STORY )       NOT NULL,
     ANTHOLOGY  REF( T_ANTHOLOGY )         NOT NULL,
     CONSTRAINT PK_SHSTORY_ANTH PRIMARY KEY ( CONTENTS , ANTHOLOGY )
);
```

Please note that some of the type-, table- and attribute names have been shortened due to limitations in object name lengths in DB2.

## 4.10. Constraints

As mentioned in chapter 2, constraints are a necessary means to enforce data quality. The various constraint constructs specified in SQL-99 is also described to some extent in chapter 2. DB2 UDB supports these constraint constructs to a variable degree. A very thorough evaluation of this is found in (Türker & Gertz 2001) together with a similar evaluation for several other commercial database systems.

### 4.10.1. Simple constraints

DB2 UDB supports all simple constraint types, with some limitations. In addition to the SQL-99 defined simple constraints, DB2 UDB also supports a constraint `FOR BIT DATA`. If a column has been specified to have a `FOR BIT DATA` constraint, the data in that column is always treated as binary data. The effect of this is that data transfer to and from the database, and between databases, ignores any code page translation rules. Furthermore, comparisons are done in binary, irrespective of the database collating sequence.

Not all constraint types can be used on all columns. Which types that can be used are dependent on the column's data type. The ground rule is that all constraint types may be applied unless the column's data type is `DATALINK`, `LONG CARCHAR`, any of the `LOB` types, or any of the graphic string types. For these data types, only the `NOT NULL` constraint can be used.

In connection with constraint types that somehow are involved an index, there is also a limitation as to how many columns are involved in each single constraint. The limitation is that a maximum of 16 columns can be involved, and the sum of the involved columns' stored length must not exceed 1024 bytes.

Constraint modes are partly supported in DB2 UDB. All constraints will have default modes `INITIALLY IMMEDIATE` and `NOT DEFERRABLE`. This means that all constraints are always checked at the end of each statement. DB2 UDB provides means for turning constraints checking on and off for a specific table. This is done with a `SET INTEGRITY` command. The details of this command can be found in (IBM DB2 SQL Reference guide).

### 4.10.2. Domain constraints and assertions

DB2 UDB supports SIMPLE TYPEs as an alternative to domains. However, SIMPLE TYPEs do not support constraints. This means that domain constraints are not supported in DB2 UDB.

There is no support for assertions at all in DB2 UDB.

### 4.10.3. Foreign key actions and scopes

The limitations on the cardinality and length of a key described above also hold for foreign keys. DB2 UDB supports actions for both of the potentially violating events in foreign keys, i.e. UPDATE and DELETE. In case of the DELETE event, the NO ACTION, RESTRICT, CASCADE and the SET NULL actions are allowed. NO ACTION is the default action. For the UPDATE event, only RESTRICT and NO ACTION is allowed. The default is the same as for DELETE.

Scopes are supported without actions in DB2 UDB.

### 4.10.4. Inheritance

Both table hierarchies and type hierarchies are supported in DB2 UDB. This has be described above (see 4.9). In accordance with SQL-99, structured types does not allow constraints to be defined on them. There is, however, an exception: The DB2 UDB specific constraint FOR BIT DATA is allowed in the definition of a structured type. This type attribute constraint is inherited from a super-type to its sub-types and cannot be overridden. Constraints in table hierarchies are inherited in concurrence with SQL-99.

### 4.10.5. Triggers

The support for triggers is very good in DB2 UDB. Triggers are supported for all the event types specified by SQL-99, i.e. INSERT, UPDATE, SELECT and DELETE. Triggers are allowed to fire both BEFORE and AFTER the triggering actions are executed. Both row-level and statement-level triggers are supported. DB2 UDB also supports conditions for when to execute the trigger.

As of version 7.0, DB2 UDB includes support for SQL-99 part 4: SQL/PSM. This means that the action of the triggers can be implemented in a rich language, giving a high degree of flexibility as to what the trigger action can do.

## 4.11. Implementation Issues

When extending any product by adding new features, developers are always faced with key design tradeoffs, considerations and decisions to be made. To make sure that the decisions that are to be made are as much according to policy as possible, guidelines need to be defined. According to (Carey, et.al. 1999), four main principles guided IBM's researchers and developers when implementing the new object-relational features in DB2. These principles were:

1. The performance of all features needed to be at least as good as their relational equivalents.
   It would be unacceptable to offer new object-relational features that caused applications to perform worse than equivalent relational solutions.

2. The design had to be modifiable to support future work on schema- and instance-level type migration.

3. The bulk of the initial object-relational changes should be in the query compiler if possible.
   *This was motivated by a desire to localise the changes as much as possible.*

4. Structured type instances were eventually to be storable in columns as well as rows of tables.

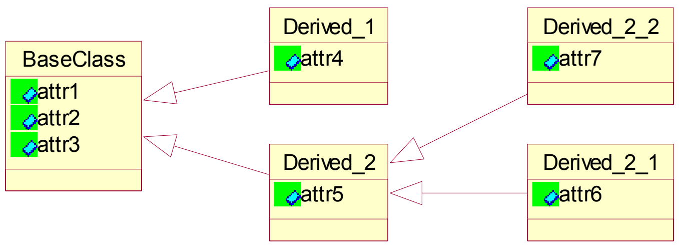
## 4.11.1. Table Hierarchies

When implementing inheritance into an object-relational database, the inheritance structure needs to be mapped into relational tables. Relational tables have to be used to satisfy IBM's design principle 4. Three different implementation approaches are considered viable. These approaches are described in (Heinckiens 1998).

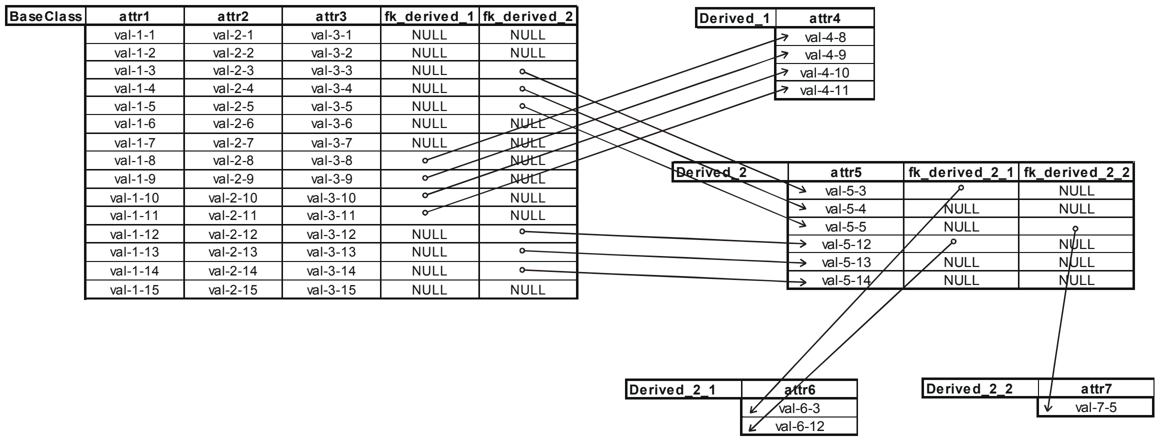
### Vertical Partitioning

In a vertical partitioning approach, one table is used for the base class and a separate table is used for each derived class. The tables for the derived classes only contain rows to hold the additional information that the derived class shall contain. Any base table additionally needs a foreign key column to each of its derived class tables.

Consider the class hierarchy presented in the UML diagram to the left.



Implementing this using a vertical partitioning would result in a table structure looking like this:



| BaseClass | attr1 | attr2 | attr3 | fk_derived_1 | fk_derived_2 |
|---|---|---|---|---|---|
| | val-1-1 | val-2-1 | val-3-1 | NULL | NULL |
| | val-1-2 | val-2-2 | val-3-2 | NULL | NULL |
| | val-1-3 | val-2-3 | val-3-3 | NULL | ○ |
| | val-1-4 | val-2-4 | val-3-4 | NULL | ○ |
| | val-1-5 | val-2-5 | val-3-5 | NULL | ○ |
| | val-1-6 | val-2-6 | val-3-6 | NULL | NULL |
| | val-1-7 | val-2-7 | val-3-7 | NULL | NULL |
| | val-1-8 | val-2-8 | val-3-8 | ○ | NULL |
| | val-1-9 | val-2-9 | val-3-9 | ○ | NULL |
| | val-1-10 | val-2-10 | val-3-10 | ○ | NULL |
| | val-1-11 | val-2-11 | val-3-11 | ○ | NULL |
| | val-1-12 | val-2-12 | val-3-12 | NULL | ○ |
| | val-1-13 | val-2-13 | val-3-13 | NULL | ○ |
| | val-1-14 | val-2-14 | val-3-14 | NULL | ○ |
| | val-1-15 | val-2-15 | val-3-15 | NULL | NULL |

| Derived_1 | attr4 |
|---|---|
| | val-4-8 |
| | val-4-9 |
| | val-4-10 |
| | val-4-11 |

| Derived_2 | attr5 | fk_derived_2_1 | fk_derived_2_2 |
|---|---|---|---|
| | val-5-3 | ○ | NULL |
| | val-5-4 | NULL | NULL |
| | val-5-5 | NULL | ○ |
| | val-5-12 | ○ | NULL |
| | val-5-13 | NULL | NULL |
| | val-5-14 | NULL | NULL |

| Derived_2_1 | attr6 |
|---|---|
| | val-6-3 |
| | val-6-12 |

| Derived_2_2 | attr7 |
|---|---|
| | val-7-5 |

Columns for OIDs are not included in the illustration.

This approach is by some considered to be the most correct way to implement a class hierarchy into relational tables. A major drawback however, is the introduction of extra joins. The simple query

```
SELECT * FROM DERIVED_2_2;
```

will actually cause a query involving two joins to be executed:

```
SELECT
     T1.ATTR1, T1.ATTR2, T1.ATTR3,
     T2.ATTR5, T3.ATTR7
FROM
     BASECLASS T1, DERIVED_2 T2, DERIVED_2_2 T3
WHERE
     T1.FK_DERIVED_2 = T2.OID
AND  T2.FK_DERIVED_2_2 = T3.OID;
```

## Horizontal Partitioning

In a horizontal partitioning, one table is created for each class in the class hierarchy. Each table will include every attribute that comprises the corresponding class, so that the attributes inherited from the root class are duplicated in all derived classes. Thus, the tables needed to implement the hierarchy given above, are:

| BaseClass | attr1 | attr2 | attr3 |
|---|---|---|---|
| | val-1-1 | val-2-1 | val-3-1 |
| | val-1-2 | val-2-2 | val-3-2 |
| | val-1-6 | val-2-6 | val-3-6 |
| | val-1-7 | val-2-7 | val-3-7 |
| | val-1-15 | val-2-15 | val-3-15 |

| Derived_1 | attr1 | attr2 | attr3 | attr4 |
|---|---|---|---|---|
| | val-1-8 | val-2-8 | val-3-8 | val-4-8 |
| | val-1-9 | val-1-9 | val-1-9 | val-4-9 |
| | val-1-10 | val-1-10 | val-1-10 | val-4-10 |
| | val-1-11 | val-1-11 | val-1-11 | val-4-11 |

| Derived_2 | attr1 | attr2 | attr3 | attr5 |
|---|---|---|---|---|
| | val-1-14 | val-2-14 | val-3-14 | val-5-14 |

| Derived_2_1 | attr1 | attr2 | attr3 | attr5 | attr6 |
|---|---|---|---|---|---|
| | val-1-3 | val-2-3 | val-3-3 | val-5-3 | val-6-3 |
| | val-1-12 | val-2-12 | val-3-12 | val-5-12 | val-6-12 |

| Derived_2_2 | attr1 | attr2 | attr3 | attr5 | attr7 |
|---|---|---|---|---|---|
| | val-1-5 | val-2-5 | val-3-5 | val-5-5 | val-7-5 |

Since all tables are disjoint and unrelated, this approach makes joins superfluous. However, in queries where polymorphism is needed, unions will be compulsory. For the query

```
        SELECT * FROM BASECLASS;
```

to be executed, and to include data from subclasses, the query processor will have to execute:

```
        SELECT
             ATTR1, ATTR2, ATTR3
             FROM BASECLASS
        UNION
        SELECT
             ATTR1, ATTR2, ATTR3
             FROM DERIVED_1
        UNION
        SELECT
             ATTR1, ATTR2, ATTR3
             FROM DERIVED_2
        UNION
        SELECT ATTR1, ATTR2, ATTR3
             FROM DERIVED_2_1
        UNION
        SELECT ATTR1, ATTR2, ATTR3
             FROM DERIVED_2_2;
```

## Hierarchy table

The third implementation approach for implementing inheritance in a relational table structure makes use of a single table. This table has a column for all attributes in the inheritance hierarchy. It is often referenced to as the *hierarchy table.* The example hierarchy used above will then be implemented as a table looking like this:

| BaseClass_Hierarchy | attr1 | attr2 | attr3 | attr4 | attr5 | attr6 | attr7 |
|---|---|---|---|---|---|---|---|
| | val-1-1 | val-2-1 | val-3-1 | NULL | NULL | NULL | NULL |
| | val-1-2 | val-2-2 | val-3-2 | NULL | NULL | NULL | NULL |
| | val-1-3 | val-2-3 | val-3-3 | NULL | val-5-3 | val-6-3 | NULL |
| | val-1-4 | val-2-4 | val-3-4 | NULL | val-5-4 | NULL | NULL |
| | val-1-5 | val-2-5 | val-3-5 | NULL | val-5-5 | NULL | val-7-5 |
| | val-1-6 | val-2-6 | val-3-6 | NULL | NULL | NULL | NULL |
| | val-1-7 | val-2-7 | val-3-7 | NULL | NULL | NULL | NULL |
| | val-1-8 | val-2-8 | val-3-8 | val-4-8 | NULL | NULL | NULL |
| | val-1-9 | val-2-9 | val-3-9 | val-4-9 | NULL | NULL | NULL |
| | val-1-10 | val-2-10 | val-3-10 | val-4-10 | NULL | NULL | NULL |
| | val-1-11 | val-2-11 | val-3-11 | val-4-11 | NULL | NULL | NULL |
| | val-1-12 | val-2-12 | val-3-12 | NULL | val-5-12 | val-6-12 | NULL |
| | val-1-13 | val-2-13 | val-3-13 | NULL | val-5-13 | NULL | NULL |
| | val-1-14 | val-2-14 | val-3-14 | NULL | val-5-14 | NULL | NULL |
| | val-1-15 | val-2-15 | val-3-15 | NULL | NULL | NULL | NULL |

As can be seen from this table, this approach implies an extensive presence of NULL values. Dependent of the database implementation, this could result in a waste of storage space. Query processing, on the other hand, is very easy: Since all data is stored in a single table, no joins or unions are needed for queries involving only classes within the inheritance hierarchy.

## IBM's Choice

As it happens, the three approaches presented above are exactly the alternatives that IBM chose to evaluate for the implementation of inheritance hierarchies (Heinckiens 1998). Preliminary implementations where done for each approach, and tested for performance. In the test a three level hierarchy with two generalisations at the root and at each intermediate level was implemented. Every table was populated with 40,000 rows of data. The table[41] below shows the result normalised with the results for the hierarchy table approach set to one.

| | Hierarchy Table | Vertical Partitioning | Horizontal Partitioning |
|---|---|---|---|
| **Count all rows at root** | 1.00 | 1.13 | 1.27 |
| **select 1 row at root** | 1.00 | 0.96 | 0.93 |
| **select 1 row at leaf** | 1.00 | 1.25 | 0.90 |
| **select 1 row and join at root** | 1.00 | 1.23 | 111.27 |
| **select 1 row and join at leaf** | 1.00 | 1.65 | 9.90 |
| **join all rows at root** | 1.00 | 0.68 | 3.81 |
| **join all rows at leaf** | 1.00 | 4.66 | 1.04 |
| **Average:** | 1.00 | 1.65 | 18.45 |

Even if both the vertical and the horizontal partitioning approach have a couple of results that are better than the hierarchy table approach, the overall performance of the hierarchy table approach convinced IBM's researchers and developers that this was the best solution, and they went for it.

The hierarchy table in DB2 UDB will always follow the naming convention `<rootclass>_HIERARCHY` where `<rootclass>` is the name of the root table in the hierarchy. DB2 UDB will not allow queries on the hierarchy table directly, but any query on any table in the inheritance hierarchy will be redirected to the hierarchy table.

## 4.12. Is DB2 Object-Relational?

The question this chapter set out to answer was: Is DB2 UDB an object-relational database? For better or for worse: Taking SQL-99 to be the recognised definition of what an object-relational database is, maybe the best thing to do is to make a checklist over what is expected of an ORDB and what DB2 UDB has to offer. This checklist is found on in the table on the next page.

It would be very tempting to conclude that IBM DB2 UDB supports enough object-relational (or SQL-99) features to justifiably be called an object-relational database system. However, a very vital ingredient is missing: Collection constructors. Without collection constructors, application object-oriented models are very difficult to map into the database model. The verdict must then be that DB2 UDB is not object-relational until at least one collection constructor is supported.

[41] This table is adopted from (Carey, et.al. 1999)

| SQL-99 | | DB2 UDB |
|---|---|---|
| Extended base type set | | Yes. |
| Domain types | | Yes, but no default values or domain constraints |
| Abstract data types | | Yes. |
| Collection constructors | | No. |
| Reference types | | Yes. |
| Function and procedures | | Yes. |
| External function and procedures | | Yes. |
| Operators | | Partly. Some operators are defined implicitly when new types are defined. No user defined operators supported. |
| Inheritance | | Yes. Both table and type inheritance. |
| Constraints | Simple constraints | Yes. Some limitations exist with respect to some data types. Also, some limitations exist on the cardinality and size of columns involved in index related constraints. |
| | Domains | No. |
| | Assertions | No. |
| | Foreign key actions | Partly. |
| | Scope actions. | No. |
| | Inheritance | Yes. |
| | Triggers | Yes. |

# 5. Object Persistence in an ORDBMS

WE CANNOT PROVIDE A PRIORI OR A POSTERIORI
JUSTIFICATIONS FOR A NUMBER OF OUR BELIEFS LIKE,
"OBJECTS AND SUBJECTS PERSIST IDENTICALLY OVER TIME"
**DAVID HUME**

It would be of great value to the database community to get a conclusive answer to the question *"Can an ORDBMS give orthogonal persistence?"*. It is, however, rather pretentious to try to answer this question within the scope of a masters thesis. It has been said that given an unlimited amount of time and money, anything can be done in a computer program. Although this probably is not true, if every involved part in the development of computer programming standards and APIs had managed to pull in the exact same direction, and had set forth to make every ORDBMS to support orthogonal persistence, they would most likely succeed. This statement is however, at best, qualified guesswork.

A more reasonable question to ask is: Can an ORDBMS deliver orthogonal persistence given the existing standards and open APIs? Narrowing even further: Can we achieve orthogonal persistence in an ORDBMS through JDBC?

Database access from Java is always done through JDBC directly or alternatively through some API built on top of JDBC. This means that JDBC represents the limit of what you can do vis-à-vis an object-relational database from a Java application. There are several such add-on APIs on the market: Java Blend from JavaSoft, Java database components in Borland Jbuilder, just to mention two of the more well known. This section will only discuss JDBC in detail. However, one of these alternative APIs built on top of JDBC must be briefly discussed, namely SQLJ. SQLJ is a proposed standard seeking to embed SQL in Java. This section will by no means be a tutorial on Java, JDBC or SQLJ. There are excellent reference books available for this[42]. SQLJ is being developed as a standard and will most likely become a part of SQL-99-part 10: Object Language Bindings (SQL/OLB).

JDBC has been through several major releases:

JDBC 1.0

JDBC 2.0

JDBC 2.0 – Standard Extension

JDBC 2.1

JDBC 3.0

[42] For a very good guide and reference to Java, see (Flanagan 1999). (White et.al. 1999) is a thorough guide and reference to JDBC. Both JDBC and SQLJ are well described in (Melton & Eisenberg 2000).

This is the situation at the time of writing[43]. The JDBC release used by an application is first dependent on the release available when the application was made. Secondly, but more importantly, release compliance of the database vendor's JDBC driver(s) imposes on the JDBC compliance of the application. Sadly, but probably necessarily, there seems to be a constant lag between the current release of JDBC and the release supported by the database vendors. The different releases will not be covered individually. JDBC 3.0 will be regarded as the current version, even if not necessarily all database vendors support this release at the time of writing. SQLJ is subdivided into three parts, namely parts 0, 1 and 2.

Discussions with senior development staff at Norwegian and international software development companies and consultancy companies[44] indicate that developers are not very keen on the embedded SQL approach. Developers tend to dislike both the pre-compiler concept, and the mixing of two different programming languages. This is an attitude that might just as well be based on gut feeling as on professional considerations. Nevertheless, none of the consulted persons expressed any interest in SQLJ as a possible solution to the impedance mismatch problem.

The choice of JDBC as an acronym for a Java database API has proven to be somewhat unfortunate. It gives developers too many associations with ODBC[45]. This vast API has been the cause of many programmers' headaches and many slow applications. The JDBC API is a much simpler API. It also gives much better performance than ODBC, as every database system vendor who has a Java strategy provides a JDBC driver that is closely integrated with the database engine.

The main purpose with this section is thus to try to answer the question: Can we achieve orthogonal persistence in an ORDBMS through JDBC? Alternatively, do JDBC provide a solution to the impedance mismatch problem? More specifically, an attempt to find answers to the following questions[46] will be given:

1. Can the set-at-a-time/element-at-a-time conflict be solved?

2. How are NULL values handled?

3. Is there a mismatch between SQL data types and Java data types? If so, how is this problem bridged?

4. Can SQL errors and Java errors be handled in a consistent way?

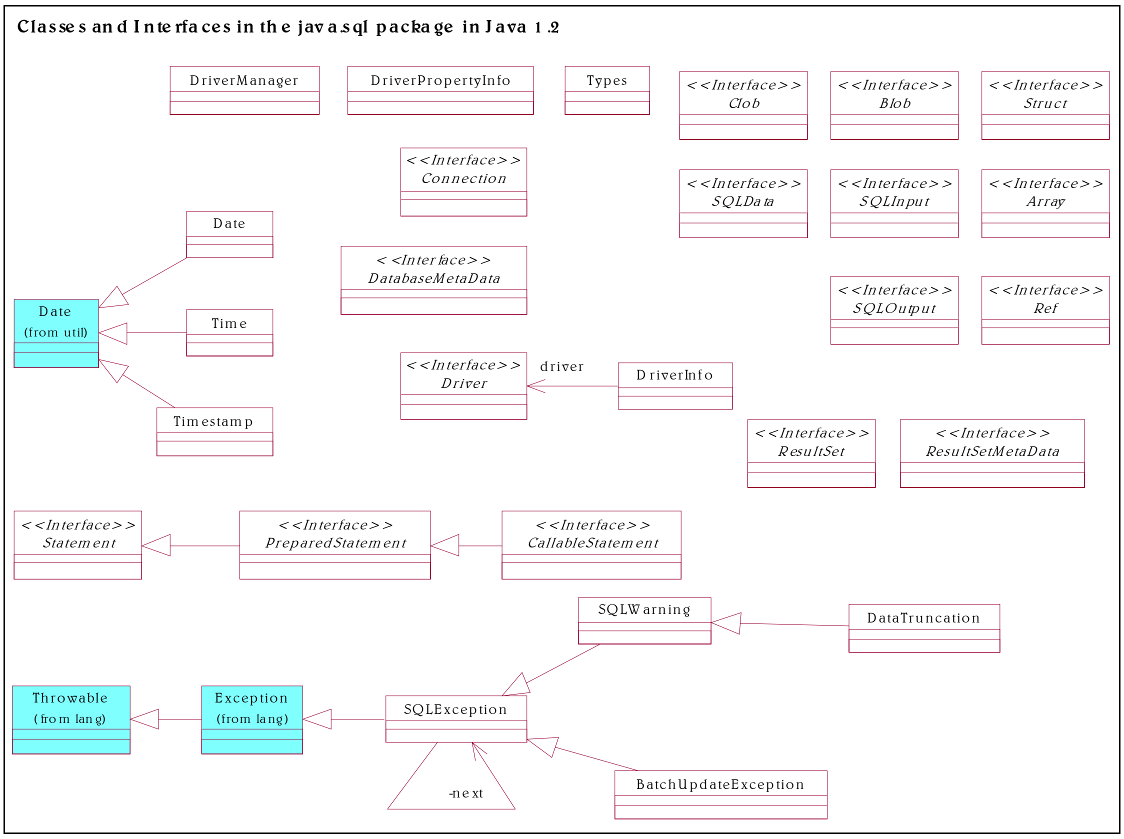
The classes and interfaces of the JDBC 3.0 API are shown in the UML diagram below. The classes `Date`, `Throwable` and `Exception` are not part of the JDBC 2.0 API, but still of major relevance. Not all of the JDBC classes are of interest for the purpose of this chapter.

[43] I.e., first half of 2002.

[44] Staff from companies such as Ergo Group AS, Logica Ltd., Microsoft Inc., Fast Search & Transfer ASA has been involved in discussions about these issues.

[45] Microsoft's Open Database Connection API

[46] These questions emerge when considering the problems described above in chapter 2.1.

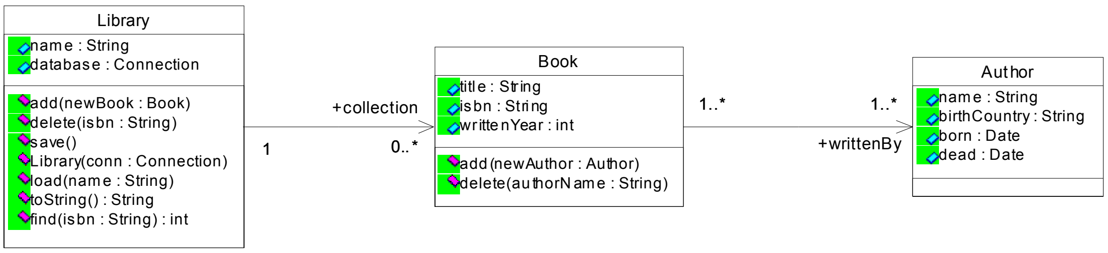


In the example Java code following, a general knowledge of Java is assumed. Java concepts will not be explained.

## 5.1. Test Case

### 5.1.1. Example Data Model

Throughout the rest of this chapter, the following model will be used (note that private attributes and operations are not shown in the diagram):



It must be admitted that a more sensible and thorough model could have been constructed, but this will suffice for the purpose of the rest of this chapter.

The model can be described as follows: A library (which servers as a persistent root) can have zero or more books. A book is written by a single author or several authors, and an author may have written one or more books[47].

The `Author` class provides no logic whatsoever. All logic is implemented in the `Book` class and the `Library` class:

| Method | Description |
|---|---|
| `Library.Library` | The `Library` constructor takes a `java.sql.Connection` object as input parameter, and thus makes the object persistent. |
| `Library.save` | The `save` method updates the `Library` object (complete with books and authors). Any locks acquired for the object is released. |
| `Library.delete` | The `delete` method removes a `Book` object specified by its ISBN (complete with authors) from the Library's book collection. |
| `Library.add` | Adds a new `book` object to a `Library's` book collection. |
| `Library.toString` | Standard `toString()` method. |
| `Library.load` | Reads the library with `LIBRARY.NAME=name` into the client application. |
| `Library.find` | Returns the index of the book specified by `isbn`; |
| `Book.add` | Adds a new `author` object to a `Book's` author list. |
| `Book.delete` | This `delete` method removes an `Author` object specified by its name from the `Book's` author list. |

Please note that this is by no means to be regarded as a complete model. Both obvious attributes and methods are absent in all of the three classes, but as already said: This model will suffice for the purpose of illustration.

The model must be realised in both Java and SQL for the use in the application and the database respectively. The methods will only be implemented in Java. The database used for the implementation of the database will be Oracle*9i*, as a database that supports collections is needed. The SQL-99 support in Oracle*9i* will not be dealt with in any details, nor will Oracle*9i* specific syntaxes be described. For more information on these areas, please refer to Oracle's manuals (Oracle9i A88878-01)and (Oracle9i A90125-01) or to other off-the-shelf references such as (Loney & Koch 2000).

Oracle's implementation of the `ARRAY` constructor is called a `VARRAY`. `VARRAY` stands for varying array. A varying array is an array which is given a maximal size instead of an absoulte size. The difference between `ARRAY` and `VARRAY` is much the same as the difference between `CHARACTER` and `CHARACTER VARYING` (see chapther 2). Oracle's preferred syntax is to use `VARRAY` when declaring collection types, but `ARRAY` is also allowed. This, however, is an undocumented feature.

Since collection types must be arrays, and arrays must have a maximal size, the model will only allow a book to have at most 100 authors, and a library to have at most 10 million books.

[47] A more philosophical question is whether you should be considered an author if your works haven't been published, maybe due to controversy. ☺

The realisation of the database model is as follows:

```
CREATE TYPE T_AUTHOR AS OBJECT
(
     NAME            VARCHAR( 100 ),
     BIRTHCOUNTRY    VARCHAR( 100 ),
     BORN            DATE,
     DEAD            DATE
);

CREATE TYPE T_AUTHORS AS ARRAY(100) OF T_AUTHOR;

CREATE TYPE T_BOOK AS OBJECT
(
     TITLE           VARCHAR( 500 ),
     ISBN            VARCHAR( 11 ),
     WRITTENYEAR     INTEGER,
     WRITTENBY       T_AUTHORS
);

CREATE TYPE T_BOOKS AS ARRAY(10000000) OF T_BOOK;

CREATE TYPE T_LIBRARY AS OBJECT
(
     ID              INTEGER,
     NAME            VARCHAR( 100 ),
     COLLECTION      T_BOOKS
);

CREATE TABLE LIBRARY OF T_LIBRARY
     OBJECT IDENTIFIER IS SYSTEM GENERATED;
```

In a more lifelike situation, it would be sensible to keep both books and authors in separate, autonomous tables. To facilitate for this, two alternative collection types must be defined:

```
CREATE TYPE T_AUTHORS_REF AS ARRAY(100) OF REF T_AUTHOR;
CREATE TYPE T_BOOKS_REF AS ARRAY(10000000) OF REF T_BOOK;
```

Furthermore, `T_BOOK.WRITTENBY` would have to be of type `T_AUTHORS_REF` and `T_LIBRARY.COLLECTION` of type `T_BOOKS_REF`. Finally, the two object tables `BOOKS` and `AUTHORS` would have to be defined. The definition given in the script above will be used for the remainder of this chapter. Another benefit of using the alternative approach would be that constraints could have been placed on the `BOOKS` and `AUTHORS` tables. Since constraints cannot be placed on types[48], the only way to enforce rules on the chosen solution would be to apply triggers or to implement access methods in the types. Neither triggers nor access methods are necessary for the purposes of this chapter, and will thus not be included.

The example database is initialised with data about the two books *"Dune"* by Frank Herbert and *"Child of Time"* by Isaac Asimov and Robert Silverberg. These data is entered (with some deliberate informational errors) with the statement:

[48] Neither Oracle*9i* nor DB2 UDB supports assertions.

```
INSERT INTO LIBRARY VALUES
(     1 ,
      'Alexandria' ,
      T_BOOKS
      (     T_BOOK
            (     'Dune' ,
                  '0450011844' ,
                  1965 ,
                  T_AUTHORS
                  (     T_AUTHOR(  'Herbert,Frank' ,
                                   'USA' ,
                                   '08-Oct-1920' ,
                                   '11-Feb-1986' ) ) ),
            T_BOOK
            (     'Child of Time' ,
                  '0330325795' ,
                  1999 ,
                  T_AUTHORS
                  (     T_AUTHOR(  'Asimov,Isaac' ,
                                   'Russia' ,
                                   '02-Jan-1920' ,
                                   NULL ) ) ) ) );
```

### 5.1.2. Main Program

The main program in this example will make several modification to the data entered through the statement above. The modification will be:

Correct to year of writing of “Child of Time” to be 1991.

Enter the sad fact that Isaac Asimov died 16th April 1992.

Add Robert Silverberg to the list of authors for “Child of Time”.

Add the newly purchased book “Look to Windwards” by Iain M. Banks to the library.

Since some scoundrel of a loaner has managed to “loose” the library’s only copy of “Dune”: Delete it from the library database.

In addition to these data manipulation operations, the test program establishes a connection to the database, and maintains this in a `java.sql.Connection` object.

```
import java.sql.*;
import java.io.*;
```

```
class test
{
     private static Connection connectToDatabase()
     {
          Connection conn;
          try
          {
               DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
               String url = "jdbc:oracle:oci:hovedfag/hovedfag@library";
               conn = DriverManager.getConnection(url);
               return conn;
          }
          catch( Exception e )
          {
               e.printStackTrace();
               System.exit(1);
          }
          return null;
     }

     public static void main(String argv[])
     {
          try
          {
               Connection db = connectToDatabase();
               if ( db != null )    // Connected OK!
               {
                    Library library = new Library( db );
                    library.load( "Alexandria" );

                    int i = library.find( "0330325795" );
                    library.collection[i].writtenYear = 1991;
                    library.collection[i].writtenBy[0].dead =
                         new Date( 92 , 03 , 16 );

                    Author silverberg = new Author();
                    silverberg.name = "Silverberg,Robert";
                    silverberg.bornCountry = "Usa";
                    silverberg.born =
                         new java.sql.Date( 35 , 00 , 15 );
                    library.collection[i].add(
                         silverberg , "T_AUTHOR" );

                    Book windward = new Book();
                    windward.title = "Look to Windward";
                    windward.isbn = "1857239695";
                    windward.writtenYear = 2000;

                    Author banks = new Author();
                    banks.name = "Banks,Iain M.";
                    banks.bornCountry = "Scotland";
                    windward.add( banks , "T_AUTHOR" );
                    library.add( windward , "T_BOOK" );

                    library.delete( "0450011844" );

                    library.save();
```

```
                        System.out.println( library );
                    }
                }
                catch ( SQLException e )
                {
                    e.printStackTrace();
                }
        }
    }
```

First of all, the program creates a `Library` object `library`, and loads the entire contents of the "Alexandria" library into `library`[49]:

```
        Library library = new Library( db );
        library.load( "Alexandria" );
```

The program then locates "Child of Time", and corrects the year it was written, and enters Asimov's date of death:

```
        int i = library.find( "0330325795" );
        library.collection[i].writtenYear = 1991;
        library.collection[i].writtenBy[0].dead =
             new Date( 92 , 03 , 16 );
```

Next, the program creates a new `Author` object `silverberg`, sets the appropriate attributes and adds `silverberg` to the author list of "Child of Time";

```
        Author silverberg = new Author();
        silverberg.name = "Silverberg,Robert";
        silverberg.bornCountry = "Usa";
        silverberg.born = new java.sql.Date( 35 , 00 , 15 );
        library.collection[i].add( silverberg , "T_AUTHOR" );
```

A new `Book` object `windward` is created and set, a new `Author` object `banks` is likewise created and set. Then `banks` is added to the author list of `windward`, and `windward` is added to the library's collection:

```
        Book windward = new Book();
        windward.title = "Look to Windward";
        windward.isbn = "1857239695";
        windward.writtenYear = 2000;

        Author banks = new Author();
        banks.name = "Banks,Iain M.";
        banks.bornCountry = "Scotland";
        windward.add( banks , "T_AUTHOR" );
        library.add( windward , "T_BOOK" );
```

Next, alas, "Dune" is removed from the library's collection:

```
        library.delete( "0450011844" );
```

[49] This would naturally not be done in a real-life situation where the library might include a collection of a very large quantity of books.

Finally, the library object is saved back to the database[50]:

```
        library.save();
```

The resulting library object is then dumped to the screen:

```
        System.out.println( library );
```

Result:

```
        Library:
            Name:   Alexandria
            Id.:    1
                Book #1:    Child of Time    0330325795             1991
                       Asimov,Isaac     Russia       1920-01-02   1992-04-16
                       Silverberg,RobertUsa          1935-01-15   null

                Book #2:    Look to Windward 1857239695             2000
                       Banks,Iain M.    Scotland     null         null
```

A complete listing of the implementation of the classes Library, Book and Author is found in appendix A.

## 5.2. Observations

### 5.2.1. Reading and Writing

A JDBC based application working with user-defined SQL-99 types does all reading and writing of objects through implementations of the `java.sql.SQLData` interface. Any implementation of this interface must implement the following methods:

`String getSQLTypeName()`

Returns the fully-qualified name of the SQL user-defined type that this object represents. Usually, the body of this implementation is:

```
        {
             return sql_type;
        }
```

Where `sql_type` is a `private String` attribute initialised by the `readSQL(…)` method.

`void readSQL(SQLInput stream, String typeName)`

Populates this object with data read from the database.

`void writeSQL(SQLOutput stream)`

Writes this object to the given SQL data stream, converting it back to its SQL value in the data source.

[50] Wouldn't it have been nice if they could have done that in Alexandria back in 47 b.c.?

The effort needed to implement these methods is directly dependent on the complexity of the class' attribute list. A comparison of the number of lines of code for the `readSQL` and `writeSQL` methods of the `Book` class and the `Author` class illustrates this:

| | |
|---|---|
| Book.readSQL | 22 |
| Book.writeSQL | 10 |
| Author.readSQL | 5 |
| Author.writeSQL | 4 |

In principle, the implementation of `readSQL` and `writeSQL` should be straightforward. However, it turns out that the standard JDBC 2.0 classes have no support for arrays of objects. Fortunately, Oracle's implementation of the JDBC 2.0 interfaces adds support for this, allowing the programmer to use an `oracle.sql.ARRAY` object to retrieve arrays of objects. This works very fine, but makes the program less database transparent.

When the methods of the `java.sql.SQLData` interface has been implemented, the complexity of the actual read and write operations are dependent on the complexity of the objects that are to be read or written. For the details, see the source code in appendix A.

Before using these implementations of the `java.sql.SQLData` interface, the association of the Java classes and the ORDBMS UDT classes must be mapped for JDBC. This means that a mapping must be inserted into the JDBC connection's type map for each class pair. In the example program, this is done in the `Library` class constructor:

```
Library( Connection conn )
      throws SQLException
{
      db = conn;
      try
      {
            Map map = conn.getTypeMap();
            map.put( "T_BOOK", Class.forName("Book") );
            map.put( "T_AUTHOR", Class.forName("Author") );
      }
      catch ( ClassNotFoundException e )
      {
            e.printStackTrace();
      }
}
```

It could, just as well have been done somewhere else in the program (i.e. before any data is transferred), e.g. in `test.ConnectToDatabase()`.

### 5.2.2. The set-at-a-time/element-at-a-time conflict

There is little doubt that the dynamics of both `java.sql.SQLData.readSQL` and `java.sql.SQLData.writeSQL` could have been more streamlined when it comes to the handling of complex objects. Nevertheless, as soon as a program gets the "feeling" of how to do this, it will work smoothly. Moreover, as soon as these methods are implemented, the data exchange works satisfactory. Most importantly, from the Java programmer point of view, it **is** objects that are exchanged. However, it is a completely different question how this is handled internally in JDBC. This is wholly up to each individual JDBC driver vendor.

The example Java code does not include any `INSERT` statements. This follows from the fact that the program handles the same `Library` object throughout its execution. In `Library.save` an UPDATE command is used:

```
sql  = "UPDATE LIBRARY L SET ";
sql += "L.ID = ?,";
sql += "L.NAME = ?, ";
sql += "COLLECTION = ? ";
sql += "WHERE L.NAME = ?";

PreparedStatement stmt = db.prepareStatement( sql );

....

stmt.setInt( 1, id );
stmt.setString( 2 , new_name );
ARRAY tmp = new ARRAY( collectionDescriptor , db , collection );
stmt.setArray( 3 , tmp );
stmt.setString( 4 , old_name );

stmt.executeUpdate();
```

This illustrates that objects can be passed as single SQL parameters in UPDATE commands too. The same holds for INSERT commands

From these examples, the set-at-a-time/element-at-a-time conflict seems to be solved.

## 5.2.3. Handling NULL values

When inserting the `banks` object into the list of authors of the `windward` object on page 75, the `banks` object does not get any values for the `born` and `dead` attributes. Their Java values are thus `null`. Still, no specific action has to be taken before the `banks` and `windward` objects are sent to the database. Furthermore, the final results of the program (shown on page 76) show that the corresponding attributes of the objects in the database have correctly been set to `NULL` and are retrieved back as `null`.[51]

It can be concluded that NULL values are handled in a satisfactory way.

## 5.2.4. SQL data types versus Java data types

When it comes to SQL data types versus Java data types, it has already been shown in chapter 2 that the type system of an SQL-99 compliant database can express the same data structures that most object-oriented programming languages. It is up to the individual ORDBMS and/or JDBC driver vendor to implement the constructs specified in the standard. The base type set of an SQL-99 compliant database is for the most part matched in Java through the base type set of the programming language itself. The base types that are not matched by Java's base type set, are implemented in the `java.sql` package. A sufficiently malevolent mind could probably always figure out

[51] `NULL` (all capitals) designates a non-existent database attribute value, whereas `null` (all lower-case) designates a non-initialised Java variable value.

some Java type construct that becomes more or less unfeasible to implement in SQL-99, but the data type mismatch problem seems to be solved in a satisfactory way in Java.

### 5.2.5. SQL errors versus Java errors

Any error situations that occur the database systems are reported through an SQL`Exception`. This is a specialization of a `java.lang.Exception`. From this it is reasonable to conclude that the handling of SQL errors and Java errors are done in a consistent way.

## 5.3. Conclusion

Can an ORDBMS deliver orthogonal persistence? Well, as said above (on page 68) this is a question that is too ambiguous to try to answer within the scope of a Masters Thesis. The answers to the questions on page 69 have all turned out to be yes. This means that JDBC *can* provide a solution to the impedance mismatch problem, and that we thus *can* achieve orthogonal persistence in an ORDBMS through JDBC.

# 6. Conclusions

THE SKILL WITH WHICH TERMINATION, OR A CONDITION WE MIGHT CALL "NEAR TERMINATION", IS MANAGED HAS A GREAT DEAL TO DO WITH THE QUALITY OF LIFE AFTER THE PROJECT.
**(MEREDITH & MANTEL 1989)**

## 6.1. Recapturing the Findings

### 6.1.1. SQL-99

During the 1990s and into the new millennium, the SQL standard has been subject to major revisions and additions. What has commonly become known as SQL-99 is a multipart standard, and this thesis has mainly been looking at the object-oriented extensions in part 2 (ISO/IEC 9075-2 1999) of the new standard. Many object-oriented concepts have been incorporated into SQL-99, among these are:

Extended and extendable type set, including object types with methods.

Collection and reference types.

Typed tables.

Type and table inheritance.

Function and procedure overloading.

Despite these major achievements, there are still constructs missing from making SQL-99 a completely object-oriented language. To facilitate the user with tools to freely build complex data types and classes, better support for collection types is needed. Although user defined operators are not considered a necessity for an object-oriented language, the support for this would be a valuable asset to the object-orientation of SQL. The lack of support for encapsulation and information hiding is probably the weakest point in new SQL-99 standard. This shows that despite a significant revision of the SQL standard, there are still reasons to be reluctant as to declaring SQL-99 as object-oriented.

### 6.1.2. DB2

In DB2 Universal Database Server, IBM has delivered a database product with very good support for the new features and facilities in the 2nd part of SQL-99. As already mentioned in chapter 3: The largest "hole" in IBM DB2 UDB is the absolute absence of collection constructors. The decision to put this "on ice" may have been a good business decision (as probably few have started to take advantage of the object-relational aspects of SQL-99), but it is still the one thing that might justify the conclusion that DB2 UDB is not an object-relational database.

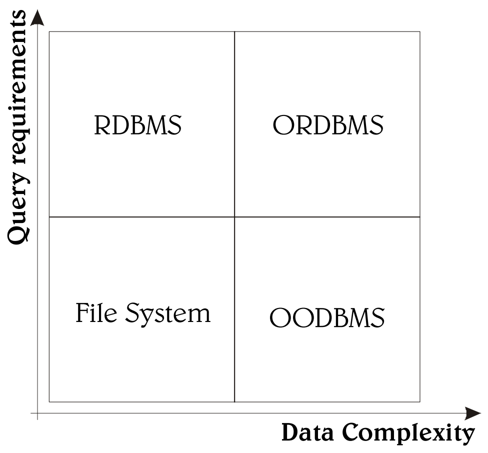
DB2 UDB will also become a better database product if the `DISTINCT TYPE` construct is converted into a `DOMAIN` construct and made to include a domain constraint list.

### 6.1.3. Persistent Objects and Java

This thesis has shown that it is possible to solve the impedance mismatch problem by means of the object-oriented constructs already defined in SQL-99 and the APIs defined in JDBC 2.0. It is also possible to achieve some object persistence by means of the technology available today. This thesis has not shown whether orthogonal persistence can be achieved, and if not, what it does take to get there.

## 6.2. The DBMS Matrix

In (Stonebraker & Moore 1996) the situation on the database systems market is described in a simple 2x2 matrix:

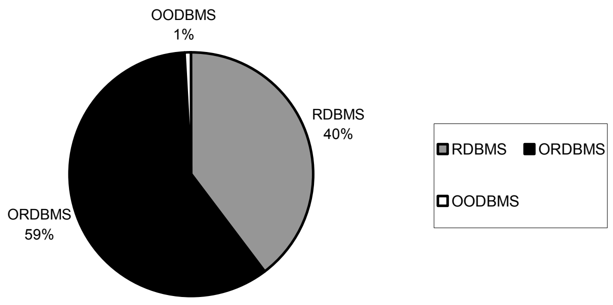


According to this matrix, it is only an ORDBMS that can satisfy a situation with complex data that also has a need for query capabilities.

Based on what he sees as two "dramatic" driving forces, namely:

1. Computerisation of new multimedia applications.
2. Business data processing will show a growing need for complex query possibilities on complex data.

Stonebraker predicts a development that results in the following distribution of market shares in the year 2005:



There are several other aspects of the database community that should be taken into consideration when making such a prediction.

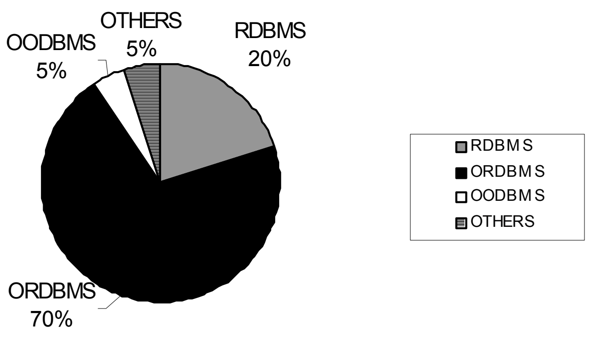
First, there are at least two very good reasons why OODBMSs will play a much larger part in the future. Given the fact that OQL is being standardised and that it or some other object query mechanism (such as SQL) is being implemented in most of the commercially available OODBMSs, these are no longer restricted to the lower, right quadrant of the DBMS matrix. Therefore, OODBMSs will be able to answer the demand for database systems managing complex data and provide query capabilities. Thus, taking a larger part of the ORDBMS's market shares.

In addition, developers are today giving more and more of their attention to object-oriented analysis, modelling and programming. It is not unreasonable to assume that by the year 2005, a major part of new development is done with object-oriented tools and languages. Programmers are, even today, struggling with the interfaces between object-oriented classes and relational tables. Object-oriented databases, with or without flaws, seem to be the "promised land" for many programmers developing database applications in an object-oriented environment.

Second, it is highly unlikely that the leading RDBMS vendors will keep on developing (or even supporting) pure RDBMS products. Oracle 8 and 9 will totally replace Oracle 7 within very few years, and DB2 Universal Server will, in time, replace older DB2 versions. So, even if some database customers will be conservative and cautious taking advantage of the new object-relational functionality in these systems, the number of pure relational database systems will dwindle and in time fade to zero.

Furthermore, it should not be forgotten that there still are large hierarchical databases and network databases in use today. These are in many cases critical databases in governmental and military systems, and are not necessarily replaced in the immediate future.

Based on these considerations, a more realistic prediction might look like this:



These predictions are by no means claimed to be accurate, but might be closer to what will prove to be the future situation.

## 6.3. Future Work

### 6.3.1. SQL-99

Several issues concerning SQL-99 have deliberately been omitted from this thesis due to its intended scope. None of the parts 3 or 5 have been looked into, and part 4 has only briefly been examined in chapter 2. Also, this thesis has not discussed whether

SQL-99 is the right way to go to achieve object-orientation within the relational database framework.

Possible topics for future research is:

- A comparison between the different approaches to object-orientation in databases described in (Atkinson, et.al. 1990), (Stonebraker et.al. 1990) and (Date & Darwen 1998).

- Objects persistence by means of the API described in SQL-99 Part 3 - Call-Level Interface (SQL/CLI).

- Objects persistence by means of the API described in SQL-99 Part 5 - Host Language Bindings (SQL/Bindings).

### 6.3.2. DB2

As with SQL-99, this thesis has only studied DB2 UDB with respect to the implementation and support for SQL-99 Part 2 - Foundation (SQL/Foundation) and to some extent SQL-99 Part 4 - Persistent Stored Modules (SQL/PSM). Worthwhile future work with regards to IBM DB2 (and other commercial SQL-99 compliant databases) includes:

Compliance to SQL-99 Part 3 - Call-Level Interface (SQL/CLI).

More on compliance to SQL-99 Part 4 - Persistent Stored Modules (SQL/PSM).

Compliance to SQL-99 Part 5 - Host Language Bindings (SQL/Bindings).

### 6.3.3. Persistent Objects and Java

In chapter 4 this thesis has looked into solving the impedance mismatch problem and persistent objects in object-relational databases by means of JDBC 2.0. There are several possible ways to proceed further from this work. Examples are:

Persistent objects in object-relational databases by means of C#.

Persistent objects in object-relational databases by means of CORBA services such as "Persistent Object", "Relationship", "Transaction" and "Query"[52].

[52] See (Mowbray & Ruh 1997)

# App. A. Java Source Code

## A.1 Library.java

```
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.*;
import java.util.*;

public class Library
{
	public int		id;
	public String		name;
	public Object[]		objects;
	public Book[]		collection;

	private ArrayDescriptor collectionDescriptor;
	private Connection	db;
	private boolean		libr_exists_in_db;
	private boolean		have_read_collection = false;

	Library( Connection conn )
		throws SQLException
	{
		db = conn;
		try
		{
			Map map = conn.getTypeMap();
			map.put( "T_BOOK", Class.forName("Book") );
			map.put( "T_AUTHOR", Class.forName("Author") );
		}
		catch ( ClassNotFoundException e )
		{
			e.printStackTrace();
		}
	}

	public int find( String isbn )
	{
		int i = 0;
		boolean found = ( collection[i].isbn.equals( isbn ) );

		while ( ( ! found ) && ( i < collection.length - 1 ) )
		{
			++i;
			found = ( collection[i].isbn.equals( isbn ) );
		}
		return i;
	}

	public void save()
	{
		String sql;
		Boolean success;
		try
		{
			sql  = "UPDATE LIBRARY L SET ";
			sql += "L.ID = ?,";
			sql += "L.NAME = ?, ";
			sql += "COLLECTION = ? ";
			sql += "WHERE L.NAME = ?";

			PreparedStatement stmt = db.prepareStatement( sql );
```

```
            if ( ! have_read_collection )
                collectionDescriptor =
                    ArrayDescriptor.createDescriptor( "T_BOOKS" , db );

            stmt.setInt( 1, id );
            stmt.setString( 2 , name );
            ARRAY tmp =
                new ARRAY( collectionDescriptor , db , collection );
            stmt.setArray( 3 , tmp );
            stmt.setString( 4 , name );

            stmt.executeUpdate();

        }
        catch ( SQLException e)
        {
            e.printStackTrace();
        }
    }

    public void load( String libname )
    {
        String sql;
        Boolean success;
        try
        {
            sql  = "SELECT * FROM LIBRARY ";
            sql += "WHERE NAME = '" + libname + "'";
            ResultSet rs = db.createStatement().executeQuery( sql );

            libr_exists_in_db = ( rs.next() );

            if ( libr_exists_in_db )
            {
                id              = rs.getInt( "ID" );
                name       = rs.getString( "NAME" );

                ARRAY tmp  = (ARRAY)(rs.getArray( "COLLECTION" ));

                objects         = (Object[])tmp.getArray();
                collection = new Book[ objects.length ];

                for ( int i = 0 ; i < objects.length ; ++i )
                {
                    collection[i] = (Book)objects[i];
                    collection[i].setConnection( db );
                }

                collectionDescriptor = tmp.getDescriptor();
            }
            have_read_collection = true;
        }
        catch ( SQLException e)
        {
            e.printStackTrace();
        }
    }
```

```
public String toString()
{
      String buff;
      buff =     "Library:\n" +
                 "\tName:\t" + name +
                 "\n\tId.:\t" + id +
                 "\n";
      for ( int i = 0 ; i < collection.length ; ++i )
      {
            buff += "\t\t" + "Book #" + ( i + 1 ) + ": ";
            buff += collection[i].title + "\t";
            buff += collection[i].isbn + "\t";
            buff += collection[i].writtenYear + "\n";

            buff += "\t";
            for ( int j = 0 ; j < collection[i].writtenBy.length ; ++j )
            {
                  buff += "\t";
                  buff += "\t" + collection[i].writtenBy[j].name;
                  buff += "\t" + collection[i].writtenBy[j].bornCountry;
                  buff += "\t" + collection[i].writtenBy[j].born;
                  buff += "\t" + collection[i].writtenBy[j].dead;
                  buff += "\n\t";
            }
            buff += "\n";
      }

      return buff;
}

public void add( Book b , String typename )
      throws SQLException
{
      if ( ! have_read_collection )
            collection = new Book[0];

      Book[] newBookList = new Book[ collection.length + 1 ];
      int i;
      for ( i = 0 ; i < collection.length; ++i )
      {
            newBookList[i] = new Book();
            newBookList[i] = collection[i];
      }
      newBookList[i] = b;
      newBookList[i].setConnection( db );

      collection = new Book[ newBookList.length ];

      for ( i = 0 ; i < newBookList.length ; ++i )
      {
            collection[i] = new Book();
            collection[i] = newBookList[i];
            collection[i].setSQLTypeName( typename );
      }
}
```

```
     public void delete( String isbn )
     {
          Book[] newBookList = new Book[ collection.length - 1 ];
          int i;
          int j = 0;
          for ( i = 0 ; i < collection.length; ++i )
               if ( ! collection[i].isbn.equals( isbn ) )
               {
                    newBookList[j] = new Book();
                    newBookList[j] = collection[i];
                    ++j;
               }

          String typename = "";
          if ( newBookList.length > 0 )
               typename = newBookList[0].getSQLTypeName();

          collection = new Book[ newBookList.length ];

          for ( i = 0 ; i < newBookList.length ; ++i )
          {
               collection[i] = new Book();
               collection[i] = newBookList[i];
               collection[i].setSQLTypeName( typename );
          }
     }
}
```

## A.2 Book.java

```
import java.sql.*;
import oracle.sql.*;

public class Book implements SQLData
{
     public String   title;
     public String   isbn;
     public int           writtenYear;
     public Author[] writtenBy;
     public int x;

     private ArrayDescriptor authorsDescriptor;
     private String           sql_type;
     private Connection       db;
     private boolean          have_read_authors = false;

     public void setConnection( Connection conn )
     {
          db = conn;
     }

     public void add( Author a , String typename )
          throws SQLException
     {
          if ( ! have_read_authors )
               writtenBy = new Author[0];

          Author[] newAuthorList = new Author[ writtenBy.length + 1 ];
          int i;
          for ( i = 0 ; i < writtenBy.length; ++i )
          {
               newAuthorList[i] = new Author();
               newAuthorList[i] = writtenBy[i];
          }
          newAuthorList[i] = a;
          writtenBy = new Author[ newAuthorList.length ];

          for ( i = 0 ; i < newAuthorList.length ; ++i )
          {
               writtenBy[i] = new Author();
               writtenBy[i] = newAuthorList[i];
               writtenBy[i].setSQLTypeName( typename );
          }
     }

     public void setSQLTypeName( String name )
     {
          sql_type = name;
     }

     public String getSQLTypeName()
     {
          return sql_type;
     }

     public void readSQL( SQLInput stream, String type )
          throws SQLException
     {
          sql_type = type;

          title          = stream.readString();
          isbn      = stream.readString();
          writtenYear    = stream.readInt();
          ARRAY tmp  = (ARRAY)(stream.readArray());

          if ( tmp == null ) // No authors
          {
               writtenBy = new Author[0];
          } else
          {
               Object[] objects     = (Object[])tmp.getArray();

               writtenBy = new Author[ objects.length ];
               for ( int i = 0 ; i < objects.length ; ++i )
                    writtenBy[i] = (Author)objects[i];

               authorsDescriptor = tmp.getDescriptor();

               have_read_authors = true;
          }
     }

     public void writeSQL( SQLOutput stream )
          throws SQLException
     {
          if ( ! have_read_authors )
               authorsDescriptor =
                    ArrayDescriptor.createDescriptor( "T_AUTHORS" , db );

          stream.writeString( title );
          stream.writeString( isbn );
          stream.writeInt( writtenYear );

          ARRAY tmp = new ARRAY( authorsDescriptor , db , writtenBy );

          stream.writeArray( tmp );
     }
}
```

## A.3 Author.java

```
import java.sql.*;
import oracle.sql.*;

public class Author implements SQLData
{
     public String   name;
     public String   bornCountry;
     public Date     born;
     public Date     dead;

     private String       sql_type;

     public String getSQLTypeName()
     {
          return sql_type;
     }

     public void setSQLTypeName( String name )
     {
          sql_type = name;
     }

     public void readSQL( SQLInput stream, String type )
          throws SQLException
     {
          sql_type   = type;
          name       = stream.readString();
          bornCountry     = stream.readString();
          born       = stream.readDate();
          dead       = stream.readDate();
     }

     public void writeSQL( SQLOutput stream )
          throws SQLException
     {
          stream.writeString( name );
          stream.writeString( bornCountry );
          stream.writeDate(    born );
          stream.writeDate(    dead );
     }
}
```

# References

Atkinson, et.al. 1990:
Atkinson, M.P. & Bancilhon, F. & DeWitt, D. & Dittrich, K. & Maier, D. & Zdonik, S.:
*"The object-oriented database system manifesto"*
Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, pp. 223-240
Elsevier Science, New York, USA, 1990

Bernstein, et.al. 1998:
Bernstein,Phil & Brodie,Michael & Ceri,Stefano & DeWitt,David & Franklin,Mike & Garcia-Molina,Hector & Gray,Jim & Held,Jerry & Hellerstein,Joe & Jagadish,H.V. & Lesk,Michael & Maier,Dave & Naughton,Jeff & Pirahesh,Hamid & Stonebraker,Mike & Ullman,Jeff:
*" The Asilomar Report on Database Research"*
SIGMOD Record, vol.27, no.4, ACM 1998

Carey, et.al. 1999:
Carey,Michael & Doole,Doug & Mattos,Nelson:
*"O-O, What Have They Done to DB2?"*
Proceedings of the 25th VLDB Conference, pp.542-553
Morgan Kaufmann Publishers, San Francisco, California, USA, 1999.
ISBN 1-55860-615-7

Cattell 1997:
Cattell, R.G.G., et.al (eds):
*"The Object Database Standard: ODMG 2.0"*
Morgan Kaufmann Publishers, San Francisco, California, USA, 1997.
ISBN 1-55860-463-4

Chen 1976:
Chen,P.P.:
*"The entity-relationship model – Towards a unified view of data"*
ACM Transactions on Database Systems, Vol.1, No.1, pp.9-36, 1976.

Codd 1970:
Codd, E.F.:
*"A Relational Model of Data for Large Shared Data Banks"*
Communications of the ACM, Vol.13, No.6, pp. 377-387, 1970.

Cooper 1997:
Cooper, Richard:
*"Object Databases – An ODMG Apporach"*
International Thompson Computer Press, 1997.
ISBN 1-85032-294-5

Date & Darwen 1998:
Date,C.J. & Darwen,Hugh:
*"Foundation for Object/Relational Databases – The Third Manifesto"*
Addison Wesley, Reading, Massachusetts, USA, 1998.
ISBN 0-201-30978-5

FIPS/184 1993:
*Standard For Integration Definition For Information Modeling (Idef1x)*
http://www.nist.gov/
National Institute of Standards and Technology, 1993

Gray 1996:
Gray, Jim:
*"Evolution of Data Management"*
IEEE Computer, Vol.29, No.10, pp.38-46

Gray & Reuter 1993:
Gray, Jim & Reuter, Andreas:
*"Transaction Processing: Concepts and Techniques"*
Morgan Kaufmann Publishers, San Francisco, California, USA, 1993.
ISBN 1-55860-190-2

Gulutzan & Pelzer 1999:
Gulutzan,Peter & Pelzer,Trudy:
*"SQL-99 Complete, Really – An example based reference manual of the new standard"*
R&D Books, Lawrence, Kansas, USA. 1999.
ISBN 0-87930-568-1

Flanagan 1999:
Flanagan, David:
*"Java in a Nutshell, 3rd edition"*
O'Reilly & Associates, Sebastopol, California, USA, 1999.
ISBN 1-56592-487-8

Heinckiens 1998:
Heinckiens,Peter M.:
*"Building Scalable Database Applications"*
Addison Wesley, Reading, Massachusetts, USA, 1998.
ISBN 0-201-31013-9

IBM DB2 SQL Reference guide:
*IBM DB2 Universal Database SQL Reference*
DB2 Online Books: `<DB2DIR>/doc/html/db2s0/index.htm`

IBM DB2 Application Development Guide:
*IBM DB2 Application Development Guide*
DB2 Online Books: `<DB2DIR>/doc/html/db2a0/index.htm`

ISO/IEC 9075-1 1999:
*Information Technology - Database Languages - SQL*
Part 1: Framework (SQL/Framework)
ISO 1999

ISO/IEC 9075-2 1999:
*Information Technology - Database Languages - SQL*
Part 2: Foundation (SQL/Foundation)
ISO 1999

ISO/IEC 9075-3 1999:
*Information Technology - Database Language SQL*
*Part 3: Call Level Interface (SQL/CLI)*
ISO 1999

ISO/IEC 9075-4 1999:
*Information Technology - Database Languages – SQL*
*Part 4: Persistent Stored Modules (SQL/PSM)*
ISO 1999

ISO/IEC 9075-5 1999:
*Information Technology - Database Languages – SQL*
*Part 5: Host Language Bindings (SQL/Bindings)*
ISO 1999

Korth & Silberschatz 1991:
Korth, Henry F. & Silberschatz, Abraham:
*"Database System Concepts"*
McGraw Hill, New York, 1991.
ISBN 1-55860-397-2

Lakshmanan et.al 2001:
Lakshmanan,Laks V.S. & Sadri,Fereidoon & Subramanian,Subbu N.
*"SchemaSQL – An Extension to SQL for Multidatabase Interoperability"*
ACM Trans. on Database Systems, Vol.26, No.4, pp. 476-519, ACM 2001

Loney & Koch 2000:
Loney,Kevin & Koch,George
*"Oracle8i : The Complete Reference"*
Osborn McGraw-Hill, Berkeley, California, USA, 2000.
ISBN: 0072123648

Melton 1998:
Melton, Jim:
*"Understanding SQL's Stored Procedures – A Complete Guide to SQL/PSM"*
Morgan Kaufmann Publisher, Inc. San Francisco, California, USA, 1998.
ISBN 1-55860-461-8

Melton 2000a:
Melton, Jim:
*Personal e-mail communications*
18th March 2000

Melton & Eisenberg 2000:
Melton, Jim & Eisenberg, Andrew:
*"Understanding SQL and Java Together – A Guide to SQLJ, JDBC and Related Technologies"*
Morgan Kaufmann Publisher, Inc. San Francisco, California, USA, 2000.
ISBN 1-55860-562-2

Melton & Eisenberg 2001:
Melton, Jim & Eisenberg, Andrew:
*"SQL Multimedia and Application Packages (SQL/MM)"*
SIGMOD Record, vol.30, no.4, pp.97-102, ACM 2001

Meredith & Mantel 1989:
Meredith,Jack R. & Mantel,Samuel J. Jr.:
*"Project Management - A Managerial Approach"*
John Wiley & Sons, Toronto, Canada, 1989.
ISBN 0-471-50534-x

Mowbray & Ruh 1997:
Mowbray,Thomas J. & Ruh,William A.:
*"Inside CORBA – Distributed Standards and Applications"*
Addison Wesley, Reading, Massachusetts, USA, 1998.
ISBN 0-201-89540-4

Muller 1999:
Muller,Robert J.:
*"Database Design for Smarties – Using UML for Data Modelling"*
Morgan Kaufmann Publisher, Inc. San Francisco, California, USA, 1999.
ISBN 1-55860-515-0

Naiburg & Maksimchuk 2001:
Naiburg,Eric J. & Maksimchuk,Robert A.:
*"UML for Database Design"*
Addison Wesley, Reading, Massachusetts, USA, 2001.
ISBN 0-201-72163-5

Ng 2001:
Ng,Wilfred:
*"An extension of the Relational Data Model to Incorporate Ordered Domains"*
ACM Trans. on Database Systems, Vol.26, No.4, pp. 344-383, ACM 2001

Oracle9i A88878-01:
Oracle*9i* Application Developer's Guide - Object-Relational Features,
Release 1 (9.0.1), Oracle Coorporation, June 2001
`http://download-uk.oracle.com/`
`otndoc/oracle9i/901_doc/appdev.901/a88878.pdf`

Oracle9i A90211-01:
JDBC Developer's Guide and Reference.
Release 1 (9.0.1) , Oracle Coorporation, June 2001
`http://download-uk.oracle.com/`
`otndoc/oracle9i/901_doc/java.901/a90211.pdf`

Oracle9i A90125-01:
Oracle*9i* SQL Reference.
Release 1 (9.0.1) , Oracle Coorporation, June 2001
`http://download-uk.oracle.com/`
`otndoc/oracle9i/901_doc/server.901/a90125.pdf`

Özsu & Valduriez 1999:
Özsu, M.Tamer & Valduriez,Patrick:
*"Principles of Distributed Database Systems", 2nd ed.*
Prentice Hall, Upper Saddle River, New Jersey, 1999.
ISBN 0-13-659707-6

Papadimitriou 1986:
Papadimitriou, Christos:
*"The Theory of Database Concurrency Control"*
Computer Science Press, Rockville, Maryland,1986.
ISBN 0-22715-027-1

Rumbaught et.al 1999:
Rumbaugh,James & Jacobsen,Ivar & Booch,Grady:
*"The Unified Modelling Language Reference Manual"*
Addison Wesley, Reading, Massachusetts, USA, 1999
ISBN 0-201-30998-x

Silberschatz, et.al. 1996:
Silberschatz,Avi & Stonebraker,Mike & Ullman,Jeff:
*"Database Research Achievements and Opportunities Into the 21st Century"*
SIGMOD Record, vol.25, no.1, ACM 1996

Stonebraker et.al. 1990:
Stonebraker, M. & Rowe, L.A. & Lindsay, B. & Gray, J. & Carey, M. & Brodie, M. & Bernstain, P. & Beech, D.:
*"The Third-Generation Database System Manifesto",*
SIGMOD Record, vol.19, no.3, ACM 1990

Stonebraker & Moore 1996:
Stonebarker, Michael & Moore, Dorothy:
*"Object-Relational DBMSs - The Next Great Wave"*
Morgan Kaufmann Publisher, Inc. San Francisco, California, USA, 1996.
ISBN 1-55860-397-2

Türker & Gertz 2001:
Türker, Can & Gertz, Michael:
*"Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems"*
The VLDB Journal, vol.10, no.4, pp. 241-269, Springer-Verlag Berlin Heidelberg 2001.

Ullman 1988:
Ullman, Jeffrey D.:
*"Principles of Database and Knowledge-base Systems" vol. I & II*
Computer Science Press, Rockville, Maryland,1988.
ISBN 0-7167-8158-1 and ISBN 0-7167-8162-x

White et.al. 1999:
White, Seth & Fisher, Maydene & Cattell, Rick & Hamilton, Graham & Hapner, Mark:
*"JDBC API Tutorial and Reference, 2nd Edition"*
Addison Wesley, Reading, Massachusetts, USA, 1999.
ISBN 0-201-43328-1

Yu & Meng 1998:
Yu, Clement T. & Meng, Weiyi:
*"Principles of Database Query Processing for Advanced Applications"*
Morgan Kaufmann Publishers, San Francisco, California, USA, 1998.
ISBN 1-55860-434-0

# Index