Objects in the Database: A Reality Check

Parvez Bashir
Information Officer, ISGEC

November 2001
Objects in the Database: A Reality Check

The RDBMS (Relational Database Management System) ruled the database roost in the 1980s and 00 (Object Oriented) programming languages revolutionized software development in the 1990s. RDBMSs and OO programming languages have complementary strengths. Relational database systems are good for managing large amounts of data; object oriented programming languages are good at expressing complex relationships among objects. Relational database systems are good for data retrieval but provide little support for data manipulation; object oriented programming languages are excellent at data manipulation but provide little or no support for data persistence and retrieval. These complementary strengths have given birth to a new breed of database management systems popularly known as Object Relational Database Management Systems (ORDBMS). The other database model which deals purely with objects is the ODBMS (Object Database Management System). ODBMSs pre-date the object relational model and have been in use since time immemorial in niche applications including multimedia.

This paper mainly focuses on database object concepts, specifications, and implementations using the two core models introduced above. Given the mountain of information on DB object orientation, the author has attempted to cut through hype and superficial information in order to arrive at a precise JEI (just enough info) treatment of OO implementation in database. Factual information in this paper has been drawn from various sources.

- The first section of the paper introduces both the object model and object database model concepts.
- In the second section, the ODMG 2.0 object database standard is discussed followed by a skeleton ODMG 2.0 specification and an implementation example using the Jasmine ODBMS. This section also covers desired ORDBMS features, and introduces a skeleton SQL-99 specification followed by implementation examples in Informix and Oracle 8i.
- The third section uncovers the basic concepts of the ORDBMS extensibility system, compares ORDBMS extended model implementations by IBM/Informix/Oracle, and provides an extended or specialized implementation example using the Oracle 8i extensibility type system.
- The fourth and final section provides some thoughts on database object implementation applicability in various application areas in an enterprise followed by guidelines on evaluating Object Relational Database Management Systems.
# Table of Contents

## 1. BASIC OBJECT CONCEPTS

1.1 The Object Model

- Encapsulation ........................................................ 1
- Inheritance ........................................................ 2
- Polymorphism ....................................................... 2
- Object Identity, References among Objects, and Collections ......................... 3

1.2 The Object Database Model

- Encapsulation ....................................................... 4
- Inheritance ........................................................ 4
- Polymorphism ....................................................... 4
- Object Identity........................................................ 4
- References among Objects ............................................. 4
- Storing Objects ....................................................... 4
- Changing Objects ....................................................... 4
- Deleting Objects........................................................ 4
- Queries ........................................................ 4

## 2. DATABASE OBJECT SPECIFICATIONS

2.1 ODBMS DATABASE OBJECT SPECIFICATION

- 2.1.1 ODBMS Object Features [ODMG 2.0] .................................................. 5
- 2.1.2 Core ODBMS Object Specification [ODMG 2.0] ........................................ 5
- Object Definition Language (ODL) ............................................... 6
- Object Query Language (OQL) ....................................................... 7
- OQL Basic Notation ........................................................ 8
- OQL Expressions ........................................................ 8
- OQL Expressions (Continued) ....................................................... 9
- OQL Examples ........................................................ 9

- 2.1.3 ODMG ODBMS IMPLEMENTATION EXAMPLE: JASMINE ...................................... 10

- Jasmine ODQL Persistent Class Family Construction ........................................ 10
- Jasmine ODQL Database Object Creation ....................................................... 11
- Jasmine ODQL Object Queries ....................................................... 12

2.2 ORDBMS DATABASE OBJECT SPECIFICATION

- 2.2.1 ORDBMS Object features [Michael Stonebraker] ........................................ 12
- Base Type Extension ........................................................ 13
- Dynamic Linking ........................................................ 13
- Client or Server Activation ....................................................... 13
- Security ........................................................ 13
- Callback ........................................................ 13
- User Defined Access Methods ....................................................... 13
- Arbitrary length Data Types ....................................................... 13
- Complex Types ........................................................ 13
- Inheritance ........................................................ 13
1. BASIC OBJECT CONCEPTS

1.1 The Object Model

The object oriented model is based on objects. Objects are constructed using class declarations in a high level language like C++ or Java and contain data as well as methods that manipulate the data. Objects properties are mainly categorized by encapsulation, polymorphism, and inheritance.

![Object Hierarchy Diagram]

Figure 1.

Figure 1. depicts classes representing different kinds of vehicles. The figure shows the object hierarchy in a vertical direction: an Auto is a special case of a Land vehicle, which in turn is a special case of a Vehicle. The class Vehicle is the greatest common denominator in the classification system.

Encapsulation

Encapsulation means that implementation of these objects can be hidden from other classes. As an example, let us take the following C++ simplified representation of the Vehicle class.
class Vehicle {
public:
   // constructors
   Vehicle();
   Vehicle(int wt);
   // interface
   int getweight() const;
   void setweight(int wt);
private:
   // data
   int weight;
};

In the above example, private variables are hidden or encapsulated. Any program using this class is only exposed to the public data or methods.

**Inheritance**
Inheritance is a technique which lets subordinate classes use data and methods of a base class. Common features can be expressed in a parent class. In the following examples, Land inherits all the instance variables from the Vehicle class.

class Land: public Vehicle 
{
public:
   // constructors
   Land();
   Land(int wt, int sp);
   // interface
   void setspeed(int sp);
   int getspeed() const;
private:
   // data
   int speed;
};

**Polymorphism**
A function which calculates the weight for a vehicle can ask each vehicle for its weight without knowing how it is computed for the vehicle. This functional behavior is termed polymorphism.

In C++, one of the techniques to implement the above is through late or dynamic binding using virtual functions. A function becomes virtual when its declaration starts with the keyword virtual. Once a function is declared virtual in a base class, its definition remains virtual in all derived classes — even when the keyword virtual is not repeated in the definition of derived classes.

The polymorphic behavior of the function getweight() is shown in the following C++ code fragment.
class Vehicle
{
    public:
        Vehicle();  // constructors
        Vehicle(int wt);  // interface.. now virtuals!
    virtual int getweight() const;
    virtual void setweight(int wt);
    private:
        int weight;  // data
};
// Vehicle's own getweight() function:
int Vehicle::getweight() const
{
    return (weight);
}

class Land: public Vehicle
{
    ...
}
class Auto: public Land
{
    ...
}
class Truck: public Auto
{
    public:
        Truck();  // constructors
        Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt);  // interface: to set two weight fields
        void setweight(int engine_wt, int trailer_wt);  // and to return combined weight
        int getweight() const;
    private:
        int trailer_weight;  // data
};
// Truck's own getweight() function
int Truck::getweight() const
{
    return (Auto::getweight() + trailer_wt);
}

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

Relationships among objects are generally established using pointers. Container classes can be created to express many-to-one relationships.
1.2 The Object Database Model

Object Databases implement the object model by providing an additional layer of persistence.

**Encapsulation**

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

**Inheritance**

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

**Polymorphism**

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

**Object Identity**

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

**References among Objects**

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

**Storing Objects**

An object can be persisted in a database by using a constructor and populating its data members.

**Changing Objects**

Objects can be retrieved from the database (internally using Object identifiers or OID's) and changed in memory before persisting back to the database.

**Deleting Objects**

An object can be deleted from the database by retrieving it and calling its destructor function.

**Queries**

Queries support all object semantics. Object Database Management Systems implement the Object Query Language (OQL) using ODMG 2.0 (Object Database Management Group) specifications.
2. DATABASE OBJECT SPECIFICATIONS

This section discusses two industry standards for housing objects in the database.

The first standard is from ODMG (Object Database Management Group) for ODBMS data definition, data manipulation, and query languages. These languages are ODL, OML, and OQL respectively.

The second standard revolves around the ANSI (X3H2) SQL-99 (originally SQL3) object specifications. SQL-99 variants are being implemented by major ORDBMS vendors.

2.1 ODBMS DATABASE OBJECT SPECIFICATION

2.1.1 ODBMS Object Features [ODMG 2.0]

The main goal of this specification is to provide standards to ensure the portability of applications across different ODBMSs. The standard is built upon the intersection of existing standard domains:

- Database (SQL)
- Objects (OMG)
- OO Programming Languages (C++, Smalltalk, Java)

The goal of an Object DBMS is to add database functionality to OO programming languages in addition to persistence. To achieve this, it is necessary to extend the semantics of the language.

- The ODMG standard consists of various components which include:
  - An Object Model
  - An Object Definition Language (ODL)
  - An Object Query Language (OQL) and late binding support for C++, Smalltalk, and Java

The Object Model is basically built upon the OMG Object Model and extends it with capabilities like:

- Multiple Inheritance
- Database Operations
- Concurrency and Object Locking
- Object Naming, lifetime, and identity

The Object Definition Language (ODL) is an extension of the OMG Interface Definition Language (IDL), and includes support for database schemas. ODL creates an abstraction that allows ODL-generated schemas to be independent of the programming language and the ODMG compliant ODBMS.

The Object Query Language (OQL) is a SQL-like language. Some extensions to OQL are support for object identity, complex objects, and operation invocation in ODMG language bindings used for embedded operations.

2.1.2 Core ODBMS Object Specification [ODMG 2.0]

The main components of the ODMG ODBMS Object specification are:
Object Definition Language (ODL)
ODMG ODL is defined using IDL (Interface Definition Language). IDL provides the type definition. A type is defined as:

```
type definition ::= interface <type name>[::<supertype_list>]
                   |
                   |
                   |
[<type_property_list>]
[<property_list>]
[<operation_list>]
```

Types may contain super types, extent naming, and keys as shown in the following <type_property_list> example.

```
interface Professor: Person
                   |
                   |
                   |
                   |
                   |
                   |
                   |
extent professors;
keys soc_sec_no;
<property_list>
<property_list>
```

Notes:
Each attribute or relationship traversal name should be specified in the property list. Extent naming and key definition may appear in any order. Supertype, extent naming and key definition may be omitted if not applicable.

The <property_list> is defined as:

```
<property_list> ::= <property_spec>;<property_spec><property_list>
<property_spec> ::= <attribute_spec>|<relationship_spec>
```

Notes:
Structured types have bracketed list of field-type pairs associated with them. Enumerated types have bracketed lists of values. Relationships have inverses. An element from one class is indicated by <class>::

<attribute_spec> example is shown below.

```
interface Professor: Person
                   |
                   |
                   |
                   |
extent professors;
keys faculty_id, soc_sec_no;

attribute integer faculty_id;
attribute integer soc_sec_no;
attribute Struct<integer number,string street, Ref<City>city> addr;
attribute Enum {male, female} gender;
```

The City interface in the above example is defined as:
interface City {
    extent cities;
    key city_code;

    attribute integer city_code;
    attribute string name;
}

A <relationship_spec> example follows:

interface Professor: Person {
    extent professors;
    keys faculty_id, soc_sec_no;
    <attribute_list>;
    relationship Set<Student> advises inverse Student::advisor;
    relationship Set<TA> teaching_assistant inverse TA::works_for;
    relationship Department department inverse Department::faculty

    <operation_list>
}

Notes:
A relationship
defines a traversal path,
designates a target type, and
provides information about an inverse traversal path.

An <operation_list> is defined followed by an example.

<operation_list> ::= <operation_spec>; | <operation_spec><operation_list>
<operation_spec> ::= <return_type><operation_name>
    ([<argument_list>])?[<exception_raised>]
...<exception_raised> ::= raises(<exception_list>)

interface Professor {
    <type_property_list>
    <attribute_list>
    <relationship_list>
    grant_tenure() raises(ineligible_for_tenure);
    hire (in Professor);
    fire (in Professor) raises (no_such_employee);
}

Object Query Language (OQL)
OQL is a functional language which:

- consists of high level primitives to deal with sets of objects, structures and lists.
- consists of operators which can freely be composed, as long as the operands respect the type system.
- uses explicit update operators (instead of operations defined on objects)
- follows a declarative nature
A query is a (possibly empty) set of query definition expressions followed by an expression. The result of a query is an object with or without identity.

**OQL Basic Notation**

q: query name  
a: atom  
e: expression  
t: type name  
p: property name  
f: operation name  
x: variable

Query definition expressions are of the form: define q as e

**Example**

Define Tom as element (select p from p in Professors where p.name = 'Tom')

**OQL Expressions**

**Elementary Expressions**  
A variable, an atom, a named object or a query name q.  
e.g. 27,nil,Students,Tom

**Arithmetic Expressions**  
<op>e, not(true)

**Construction Expressions**  
Object: t(pl:el, ..., pn:en)  
Structure: struct(pl:el, ..., pn:en)  
Set: set(el, ...,en)  
Bag: bag(el, ...,en)  
List: list(el, ...,en)  
Array: array(el,...,en)

**Collection Expressions**  
Universal quantification: for all x in el:e2  
Existential quantification: exists x in el:e2  
Membership Testing: e1 in e2  
Select From Where: select e from x1 in el, xn in en where e  
Sort by: sort x in e by el, ...,en  
Unary set operator: <op> (e)  
<op> (E {min, max, count, sum, avg})  
Group by: group x in e by (pl:el, ..., pn:en)  
with (pl:el, ..., pn:en)

**Index Collection Expressions**  
Get the i-th element: e1[e2], list(a,b,c,d) [1]  
Extracting a subcollection: e1[e2:e3], list(a,b,c,d)[1:3]  
Last and First: first(e), last(e)  
Concatenation: e1 + e2, list(1,2) + list(2,3)
OQL Expressions (Continued)

Binary Set Expressions
\[ e_1 \langle\text{op}\rangle e_2 \]
\[ \langle\text{op}\rangle \{ \text{union, except, intersect} \} \]
- bag(2,2,3,3,3) union bag(2,3,3,3) = bag(2,2,3,3,3,2,3,3)
- bag(2,2,3,3,3) intersect bag(2,3,3,3) = bag(2,2,3,3)
- bag(2,2,3,3,3) except bag(2,3,3,3) = bag(2)

Structure Expressions
\[ e \rightarrow p \]
\[ e.p \]

Conversion Expressions
- Extracting the element of a singleton: element(e)
- List to set: listtoset(e)
- Flattening: flatten(e)

Typing Expressions: \( t(e) \)

Operation Expressions: \( e \rightarrow f \), \( e.f \)

OQL Examples

```
SELECT DISTINCT x.age FROM x IN Persons
    WHERE x.name = 'Gaarder'
    Returns a literal of type Set<integer>

SELECT DISTINCT STRUCT(a: x.age, s: x.height)
    FROM x IN Persons
    WHERE x.name = 'Gaarder'
    Returns a literal of type Set<struct>

SELECT STRUCT(a: x.age, s: x.height)
    FROM x IN (SELECT y in Employees
                 WHERE y.seniority = '5')
    WHERE x.name = 'Gaarder'
    Returns a literal of type bag<struct>

Staff
    Returns a set of all staff

DEFINE Washingtonians AS
SELECT x FROM x IN STAFF
    WHERE x.address.city = 'Washington'
SELECT x.name FROM x IN Washingtonians
    Returns a set of all staff who live in Washington

Person(name:"Tom", birthdate:"4/28/63" salary:50,000)
The above is an example of creating an object in an ODBMS using a type
name constructor. Strictly speaking this is an OML (Object Manipulation
Language) operation. An OID or Object Identity is generated for this
object during persistence.

struct(a:10, b:"Tom")
The above is an example of creating an object without identity. Again,
this is an OML operation.
```
2.1.3 ODMG ODBMS IMPLEMENTATION EXAMPLE: JASMINE

Jasmine is a full-function Object Database Management System (ODBMS) with a client-server architecture and multi threaded database server. It provides an extensive array of application development tools and languages, including Java. It supports flexible application architectures, including client server, single-tier, and multi-tier applications.

Jasmine provides a powerful Object Data Query Language (ODQL) which is very similar to the ODMG 2.0 standard. Jasmine ODQL can be viewed as the first and only uniform object-oriented database programming language. While in the ODMG 2.0 standard, the Object Definition Language (ODL), the Object Manipulation Language (OML), and the Object Query Language (OQL) have entirely different syntax and semantics, ODQL brings these three components into a single language with a uniform syntax and semantics.

The object definition capabilities of ODQL allow definitions of class hierarchies. Each class may have class and instance properties, and class and instance methods, which can be written entirely in ODQL. Thus, the impedance mismatch between the syntax and semantics of the language facilities for data manipulation and computations is avoided. However, since there are many utilities and libraries already written in other languages such as C and C++, Jasmine also supports C and C++ embedding in ODQL. The data manipulation facilities of ODQL include variable declaration, assignment, if-then-else, while loops, and other programming language constructs expected in computationally complete formal languages. The query facilities of ODQL include SQL-like queries over the extent of one or multiple ODQL classes. The methods of ODQL classes can be invoked in the condition as well as selection clause of queries.

Jasmine has built-in support for multimedia data. Through a multimedia class hierarchy, it provides extensive functionality that simplifies the development of multimedia applications. This class hierarchy supports images, frame animation sequences, as well as other types of audio and video data.

Jasmine is a comprehensive multimedia object database providing a broad spectrum of development environments. This combined with Java and the ability to support multi-tier application architectures, makes Jasmine a good choice for internet application development and deployment.

The following sections, by means of examples, illustrate various facets of the Jasmine ODQL.

**Jasmine ODQL Persistent Class Family Construction**

Class families are roughly similar to tables in relational databases. A Jasmine ODQL class construction example is given below:
Jasmine ODQL Class Definition Example - TRS System

DefaultCF systemCF; /*root system class */

defineClass trscF::trsComposite
   super: systemCF::Composite
   description: "Base (Abstract) class for TRS database."
{
    ...
};

/* Class:Company
This class is at the top of the of the Object Heirarchy (containing
tree) for this example system. It contains the master lists of active
staff, projects and clients.
*/

defineClass trsCF::Company
   super: trsCF::trsComposite
   Description: "Principle container class for application system."
{
    maxInstanceSize: 8;
    instance;
    String companyTitle;
    String address;
    
    /* List of staff currently employed by company */
    List <trsCF::Staff> staffList default: List( );
    
    /* List of active projects being worked on by the company */
    List <trsCF:Project> projectList default: List( );
    
    /* List of company's current clients */
    List <trsCF:Client> clientList default: List( );
}

Note: The above example gives the reader a flavor of Jasmine ODQL class
definitions. We will not go into Staff, Project, Client, etc. class
definitions. The concept is similar to the Company class family
definition.

Jasmine ODQL Database Object Creation
The following examples show how objects are persisted in Jasmine using ODQL.

Jasmine ODQL Object Creation Example - TRS System

DefaultCF trsCF; /*application class family */

/* Define a company record */.
Company oCompany;
oCompany = Company.new( );
oCompany.companyTitle = "National Hydro Power Corporation";
oCompany.address = "11 Connaught Place, New Delhi";

/* Add Project instances to the company object */.
Project oProject1, oProject2;
oProject1 = Project.new( );
oProject1.name = "Uri Hydel Project";
oProject1.completed = TRUE;
Jasmine ODQL Object Creation Example - TRS System (continued)

```
oProject2 = Project.new( );
oProject2.name = "Salal Hydel Project";
oProject1.completed = FALSE;
company.directAdd("projectList", oProject1);
company.directAdd("projectList", oProject2);
/* Create some client objects */.
Client oClient1
oClient1 = Client.new( );
oClient1.name = "J & K Power Development Department";
oClient1.address = "Batmaloo, Srinagar";
company.directAdd("clientList", oClient1);
```

Note: Staff objects can similarly be added to the company object instance. To execute the above script in Jasmine, one would typically call the following command line utility.
```
  cdgqlie -execFile <filename>.odql
```
where <filename>.odql contains the object creation code.

Jasmine ODQL Object Queries
The following examples show how objects are queried in Jasmine using ODQL.

```
Jasmine ODQL Query Example - TRS System

DefaultCF trsCF; /*TRS application class */
/* Get a list of all projects */
Bag<Project> oProjects;
Project oProject;
oProjects = Project from Project;
/* For each Project, Get the staff */
scan(oProjects, oProject); {
    Staff oStaff;
    List<Staff> oStaffs;
oStaffs = oProject.staffList;
}
```

2.2 ORDBMS DATABASE OBJECT SPECIFICATION

There is no standard group which works toward creating an ORDBMS (Object Relational Database Management System) specification except that SQL-99 implementation is considered to be a bare minimum for a database to be considered an ORDBMS.

Michael Stonebraker, a pioneer of the ORDBMS, lays some ground rules for an ORDBMS in the book *Object-Relational DBMSs: The Next Great Wave*.

2.2.1 ORDBMS Object features [Michael Stonebraker]
A fully object Relational- database must have the following characteristics.

1. Base Type Extension
2. Complex Objects
3. Inheritance
Base Type Extension
The base type extension facility must have the following characteristic:

*Dynamic Linking*
It must be possible to adhere new user-defined base types on the fly.

*Client or Server Activation*
A user-defined function for a new base type must have the ability for being executed on either the server or the client.

*Security*
A user defined function for a new base type must not be able to corrupt the database.

*Callback*
User defined functions must be able to use other functions, and stored procedures.

*User Defined Access Methods*
Adding new access methods must be allowed for the definer of a new base type.

*Arbitrary length Data Types*
It must be possible to have user defined data types without length restrictions.

*Complex Types*
There has to be support for rich object types. These include row types, collection types, reference types, distinct, and opaque types.

*Inheritance*

*Inheritance*
A major characteristic of an ORDBMS should be support for inheritance. Both data and function inheritance is necessary.

*Overloading*
It should be possible to specialize the definition of a function to subtypes. Overloading must be supported.

*Inheritance of Types*
Types and tables are two different concepts. A table is a container used to hold instances of a type. In this way there can be multiple tables of a specific type. Each of the tables have the defined inheritance properties. If a table is not constructed from a named type, then the table will be of anonymous type and cannot utilize inheritance.

*Multiple Inheritance*
Multiple inheritance should be supported.

2.2.2 Core ORDBMS Object Specification [SQL-99]
ANSI (X3H2) and ISO SQL standardization committees have been working on the SQL-99 (formerly SQL3) object facilities since early to late nineties. This specification has endured the test of time with multiple changes along the way. There is some confusion about what remains from the original core object specification and what has been rendered obsolete. This section details core SQL-99 object facilities which primarily involve extensions to SQL's type facilities. Besides built-in types which were part of SQL-92, there are some interesting temporal and large object types in SQL-99. Large objects are nothing new and temporal types, though powerful, do not exactly exhibit object properties. Figure
2 shows parts of SQL-99 that provide the primary basis for supporting object-oriented structures are:

- Data Types
  - Built-in data types
  - Extended data types
- Complex Data Types
  - User-defined data types
- Row
- Collection

![Diagram](image)

**Figure 2.**

**User-Defined Types**

The traditional RDBMS contains built-in or primitive types. SQL-99 is a giant leap forward with the introduction of user-defined types which provide a specification to create a custom type using a CREATE TYPE statement.

In a standard RDBMS pre-defined type, the kernel provides a mechanism for:

- Storing the data physically in a table (instantiation)
- Comparing the primitive data types (ordering)
- Built-in operators for the data types, e.g., + is used to add numbers together
- Cast operations so that values in one data type can be converted to another (cast methods and transforms)

With SQL-99 UDT's, one has to put all of the above. Once this is done, we're off to the races. The UDT can be used in a similar manner as a primitive type.

The UDT is defined by a descriptor that contains twelve pieces of information:

1. The `<UDT name>` qualified by the `<Schema name>` of the Schema to which it belongs.
2. Whether the UDT is ordered.
3. The UDT’s ordering form; either EQUALS, FULL, or NONE.
4. The UDT’s ordering category; either RELATIVE, HASH, or STATE.
5. The `<specific routine designator>` that identified the UDT’s ordering function.
6. The name of the UDT’s direct supertype, if any.
7. If the UDT is a distinct type, then the descriptor of the <data type> on which it's based; otherwise an Attribute descriptor for each of the UDT's Attributes.
8. The UDT's degree; the number of its Attributes.
9. Whether the UDT is instantiable or not instantiable.
10. Whether the UDT is final or not final.
11. The UDT's Transform descriptor.
12. If the UDT's definition includes a method signature list, a descriptor for each method signature named.

A <UDT name> identifies a UDT. The required syntax for a <UDT name> is as follows:

```
<UDT name> ::= [
  <Schema Name>. ] unqualified name
```

A UDT example definition is provided below:

```
CREATE TYPE bank_rate_udt AS
  name CHAR(40), /* first attribute */
  offer_rate DECIMAL(9,2), /* second attribute */
  compounding_type CHAR(20) /* third attribute */
  NOT FINAL /* mandatory finality clause */
METHOD effective_annual_yield() RETURNS DECIMAL (9,2)
);
```

This CREATE TYPE statement results in a UDT named `bank_rate_udt`. The components of a UDT are three attributes (named `name`, `offer_rate`, and `compounding_type`) and one method (named `effective_annual_yield`).

1. The three name and data pairs `name`, `offer_rate`, and `compounding_type` are the UDT's Attribute definitions.
2. The words NOT FINAL matter only for subtyping. Briefly though, if a UDT definition does not include an UNDER clause, the finality clause must specify NOT FINAL.
3. The clause METHOD `effective_annual_yield` returns DECIMAL (9,2). Like an attribute, a “method” is a component of a UDT, However, this method is actually a declaration that a function named `effective_annual_yield` exists. This function isn't defined further in the UDT definition; there is a separate SQL statement for defining functions — CREATE METHOD. All we can see at this stage is that `effective_annual_yield` had a name and a predefined data type.

Now we can create a table with a UDT column.

```
CREATE TABLE bank (
  bank_seq_no INTEGER,
  bank_rate bank_rate_udt);
```

Inserting a new record into a UDT is non-trivial. Let's continue with our type example as shown below.
BEGIN
    DECLARE u bank_rate_udt;
    SET u = bank_rate_udt();
    SET u = u.name('Bank of America');
    SET u = u.offer_rate(8.00);
    SET u = u.compounding_type('quarterly');
    INSERT INTO BANK VALUES (1000,u);
END;

1. DECLARE u bank_rate_udt is a declaration of a temporary variable u.
2. u = bank_rate_udt() is a constructor function. The SQL-99 DBMS is automatically supposed to create this function when a CREATE TYPE statement is issued.
3. u.name(), u.offer_rate(), and u.compounding_type() are mutator functions. Once again the DBMS's job is to create these mutator functions automatically. You may ask why don't we explicitly enter as string, e.g., u.name = 'Bank of America'. This would violate the encapsulation OO principle.

To select a column associated with a type, the DBMS calls the observer functions which are also created automatically when the UDT is created.

```
SELECT bank_rate.name(),
       bank_rate.offer_rate(),
       bank_rate.compounding_type() /* Observer function */
FROM bank
WHERE bank_seq_no > 0;
```

Besides creating UDT columns, one can also create base or “typed” or “referenceable” tables from UDTs. For example,

```
CREATE TABLE bank_rate OF bank_rate_udt REF IS SELF_REFERENCING_COLUMN;
```

Typed tables are a “two-sided coin”. From one angle, they look like relations but from the other angle they look like OO instantiated classes. A self referencing column is the equivalent of an OO object identifier. All typed tables have a self referencing column which can be given any name on creation. In our example, we call it SELF_REFERENCING_COLUMN. The self referencing column uniquely identifies a single row.

Since BANK_RATE is now a TABLE, all the operations shown below become legal.

```
INSERT INTO bank_rate
VALUES ('Bank of America', 8.00, 'quarterly');
```

Note: We did not specify a value for the SELF_REFERENCING_COLUMN. It is automatically generated.

Let us now create a TABLE which references the bank_rate table.

```
CREATE TABLE account (account_id INTEGER,
                      account_rate_ref REF(bank_rate));
INSERT INTO account
    SELECT 9999, SELF_REFERENCING_COLUMN FROM bank_rate WHERE name = 'First Union National Bank';
```
For more information on the REF value, please refer to the next section on reference types.

The last concept we want to discuss under this section is how inheritance is achieved using the UNDER clause in a UDT as is shown using an EMPLOYEE hierarchy.

```sql
CREATE TYPE employee_udt AS /* create employee data type */
  soc_no VARCHAR2(10),
  emp_name VARCHAR2(40),
  birth_date DATE,
  dept_name VARCHAR2(30) NOT FINAL;

CREATE TYPE hourly_employee_udt AS /* hourly employee */
  hourly_rate DECIMAL(5,2),
  overtime_rate DECIMAL(5,2),
  max_overtime_hours INTEGER
UNDER employee_udt NOT FINAL;

CREATE TYPE salaried_employee_udt AS /* salaried employee */
  monthly_rate DECIMAL(5,2),
  bonus_pct DECIMAL(6,2)
UNDER employee_udt NOT FINAL;

CREATE TABLE employee OF employee_udt;
CREATE TABLE salaried_employee OF salaried_employee_udt;
CREATE TABLE hourly_employee OF hourly_employee_udt;
CREATE TABLE salaried_employee UNDER employee;
CREATE TABLE hourly_employee UNDER employee;
```

Using the above examples, we can think of a “family of types” which has a root super type `employee` and `hourly` and `salaried` subtypes of `employee_udt`. By declaring that `a` is a subtype of `b`, all data and methods are inherited by the sub type. In addition, `a` is automatically linked to `b` and the linking process is practically invisible. Of course, one can define additional data and methods in the sub type. In these examples, UNDER means “a subtype of”.

Note: “subtables” and “supertables” defined are a family whose relationships match the relationships of the “subtypes” and “supertypes” on which they are based.

Reference Types
A referenced type is defined by a descriptor that contains three pieces of information.

1. The data types name: REF
2. The name of the UDT that the reference is based on.
3. The scope of the reference type; the name of the table that makes up the reference type’s scope.

```sql
<reference type> ::= 
  REF (<UDT name>)
  [SCOPE <Table Name> [(<reference scope check>)]]

<reference scope check> ::= 
  REFERENCES ARE [NOT] CHECKED
  [ ON DELETE 
    {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}]
```

A REF value may have a scope; it determines the effect of a dereference operator on that value. A REF value’s scope is a typed table and consists of every row in that table. The
optional SCOPE clause of a reference type specification identifies REF’s scope. The table named in the SCOPE clause must be a referenceable table with a structured type that is the same as the structured type of the UDT the REF is based upon.

SQL-99 provides two scalar operations that operate on or return a reference type: the dereference operation and the reference resolution.

dereference operation

```
<dereference operation> ::= reference_argument \rightarrow <attribute name>
```

reference resolution

```
<reference resolution> ::= DEREF(reference_argument)
```

Row Types

A row type is defined by a descriptor that contains three pieces of information.

1. The data type’s name: ROW
2. The data type’s degree: the number of Fields that belong to the row.
3. A descriptor for each field that belongs to the row. The Field descriptor contains: the name of the Field, the Field’s ordinal position in the row type, the Field’s data type and nullability attribute, the Field’s Character set and default Collation (for character string data types), and the Field’s reference scope check (for reference types).

Please refer to following definition and subsequent example.

```
<row type> ::= ROW (<Field Definition> [ {,<Field definition>} ... ])
<Field definition> ::= <Field name> <data type> [ <reference scope check> ] [COLLATE <Collation name> ]
```

CREATE TABLE job (  
title CHAR(20),  
address ROW(street CHAR(50), city char(30), state(2), zip(9));

INSERT INTO TABLE job  
(title, address)  
VALUE ('President', ROW('1600 Pennsylvania Avenue', 'Washington', 'DC', '20433'));

SELECT title, address FROM job WHERE address.city = 'Washington'

Collection Types

A collection type is defined by a descriptor that contains three pieces of information.

1. The data type’s name: ARRAY
2. The maximum number of elements in the array.
3. The array’s data type specification.

Please refer to following definition and subsequent example.
CREATE TABLE job (  
  title CHAR(20),  
  address CHAR(50) ARRAY[3]);

INSERT INTO TABLE job  

UPDATE job  
SET address[1] = ‘The White House’;

**Distinct Types**  
The main idea of defining distinct types is that they constitute enforceable domains.  
Behold the syntax along with an example.

CREATE TYPE <UDT name> AS <predefined data type> FINAL;  
CREATE TYPE UPI euro AS DECIMAL(8,2) FINAL;  
CREATE TYPE mark AS DECIMAL(8,2) FINAL  

If we now attempt to pass a euro value to a mark target, we will fail. The distinct type  
provides us with a simple form of type checking that we cannot achieve using SQL  
Domains.

**Object Orientation and SQL-99**  
**Classes**  
UDTs are classes. SQL-99 vocabulary may include words like “type family” where most  
languages’ vocabulary would have “class family” but the essential functionality is the same.

**Encapsulation**  
SQL-99 keeps data representation separate from data access, but does not allow for  
PRIVATE and PUBLIC attribute definitions. GRANT and REVOKE handle this.

**Extensibility**  
It is possible to put together packages consisting of new type families, methods, and  
representations. Such packages exist today, although to a large extent the methods are  
external functions written in some other language.

**Inheritance**  
A UDT may be defined under another UDT. Subtypes inherit the methods and attributes  
of supertypes. Inheritance is single as in most pure OO languages.

**Instantiation**  
SQL-99 UDT’s may be used in place of predefined data types in SQL data statements.  
Rows in typed tables may be treated as objects complete with object operations.

**Polymorphism**  
Multiple methods in a type family may have the same name. The DBMS will choose the  
specific method based on the signature.
2.2.3 SQL-99 ORDBMS IMPLEMENTATION EXAMPLES: INFORMIX AND ORACLE 8i

This sections compares Informix and Oracle with respect to implementation of the core SQL-99 object infrastructure detailed in the previous section.

INFORMIX

<table>
<thead>
<tr>
<th>SQL-99 User-Defined Data Types</th>
<th>Informix Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER DEFINED TYPES</td>
<td>Informix implements UDTs as follows:</td>
</tr>
<tr>
<td></td>
<td>1) Named row types (similar to SQL-99 UDTs).</td>
</tr>
<tr>
<td></td>
<td>2) Implements SQL-99 inheritance.</td>
</tr>
<tr>
<td></td>
<td>Inheritance Examples</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE PERSON OF TYPE EMPLOYEE TYPE;</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE EMPLOYEE OF TYPE EMPLOYEE TYPE;</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE EMPLOYEE OF TYPE EMPLOYEE TYPE UNDER PERSON;</td>
</tr>
<tr>
<td>DISTINCT TYPES</td>
<td>Similar to SQL-99 distinct types.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL-99 Miscellaneous Data Types</th>
<th>Informix Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPAQUE DATA TYPE</td>
<td>This is a Informix specific type and not present in the SQL-99 spec. The internal structure of the opaque type is not visible to the database server. The internal structure can only be accessed through user-defined routines.</td>
</tr>
<tr>
<td></td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>CREATE OPAQUE TYPE html AS VARCHAR(200);</td>
</tr>
<tr>
<td>REFERENCE DATA TYPE</td>
<td>Informix does not implement the reference data type.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL-99 Complex Data Types</th>
<th>Informix Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW</td>
<td>SQL99 Examples</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE POEM (poem id NUMBER, ROW(NAME CHAR(30), AUTHOR CHAR(30)));</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO POEM VALUES (1000,ROW('Kubla Khan', 'Samuel Taylor Coleridge'));</td>
</tr>
<tr>
<td>COLLECTION</td>
<td>Informix implements the following collection types (similar to ODMG 2.0 collection types).</td>
</tr>
<tr>
<td></td>
<td>LIST This data type is a collection type which stores ordered, nonunique element values. The elements of a LIST have ordinal positions.</td>
</tr>
</tbody>
</table>
|                           | MULTISET(e) The MULTISET data type is a collection type that
stores nonunique elements: it allows for duplicate element values. The elements in a MULTISSET have no ordinal position. That is, there is no concept of a first, second, or third element in a MULTISSET. All elements in a MULTISSET have the same element type.

**SET (e)**

The SET data type is a collection type that stores unique elements: it does not allow duplicate element values.

**ORACLE 8i**

<table>
<thead>
<tr>
<th>SQL-99 User-Defined Data Types</th>
<th>Oracle 8i Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USER DEFINED TYPES</strong></td>
<td>This is similar to the SQL-99 UDT except Oracle specifically uses the object token.</td>
</tr>
</tbody>
</table>
|                               | CREATE TYPE <typename> AS OBJECT <type-clause>;
|                               | CREATE TABLE <tablename> OF <typename>;
| **Examples**                  | CREATE TYPE document_type AS OBJECT
|                               | (doc-name VARCHAR2(30), business-process VARCHAR2(30));
|                               | CREATE TABLE document_type_table OF document_type;
| Oracle 8i does not implement SQL-99 inheritance. This is a severe drawback in Oracle's current object implementation. | |
| **DISTINCT TYPES**            | Oracle 8i does not implement this type. |

<table>
<thead>
<tr>
<th>SQL-99 Miscellaneous Data Types</th>
<th>Oracle 8i Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REFERENCE DATA TYPE</strong></td>
<td>Oracle supports reference pointers, known as REFS, to point from one object to another (this is similar to the SQL-99 REF type implementation). Column syntax is:</td>
</tr>
<tr>
<td></td>
<td>&lt;attributename&gt; REF &lt;objectdatatype&gt;</td>
</tr>
</tbody>
</table>
| **Examples**                  | CREATE TYPE dept_type AS OBJECT
|                               | (deptno NUMBER, dname VARCHAR2(30));
|                               | CREATE TABLE dept_table OF dept_type;
|                               | CREATE TYPE emp
|                               | (empid NUMBER, empname VARCHAR2(30), dept REF dept_type);
|                               | UPDATE emp SET dept =
|                               | (SELECT REF(d) FROM dept_table WHERE deptno = 1) WHERE empid = 1;
|                               | SELECT dept_table.dname, empname FROM emp
<p>|                               | WHERE JOB = 'ANALYST' |</p>
<table>
<thead>
<tr>
<th>SQL-99 Complex Data Types</th>
<th>Oracle 8i Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROW</strong></td>
<td>Oracle does not implement SQL-99 row types</td>
</tr>
<tr>
<td><strong>COLLECTION</strong></td>
<td>Oracle supports two types of COLLECTIONS: varying arrays (VARRAYS) and nested tables.</td>
</tr>
<tr>
<td></td>
<td>VARRAY</td>
</tr>
<tr>
<td></td>
<td>CREATE TYPE &lt;typename&gt; AS VARRAY(N) OF &lt;type&gt;;</td>
</tr>
<tr>
<td></td>
<td>where N is the maximum size of the array</td>
</tr>
<tr>
<td></td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>CREATE TYPE address_type AS VARRAY(3) OF VARCHAR2(30));</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE person</td>
</tr>
<tr>
<td></td>
<td>(ssno NUMBER PRIMARY KEY, name VARCHAR2(30), paddress address_type);</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO PERSON</td>
</tr>
<tr>
<td></td>
<td>VALUES (99999999,'Peter Pan', address_type('Never','Never','Land'));</td>
</tr>
<tr>
<td></td>
<td>SET SERVEROUTPUT ON</td>
</tr>
<tr>
<td></td>
<td>DECLARE</td>
</tr>
<tr>
<td></td>
<td>CURSOR person_cursor IS SELECT * FROM person;</td>
</tr>
<tr>
<td></td>
<td>Person_rec person_cursor%ROWTYPE;</td>
</tr>
<tr>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>Person_rec IN person_cursor LOOP</td>
</tr>
<tr>
<td></td>
<td>DBMS_OUTPUT.PUT_LINE('Person SS#</td>
</tr>
<tr>
<td></td>
<td>FOR I in 1 .. person_rec.address.COUNT</td>
</tr>
<tr>
<td></td>
<td>DBMS_OUTPUT.PUTLINE(person_rec.address (i));</td>
</tr>
<tr>
<td></td>
<td>END LOOP;</td>
</tr>
<tr>
<td></td>
<td>END LOOP;</td>
</tr>
<tr>
<td></td>
<td>END;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NESTED TABLE</th>
<th>This is best explained by the examples shown below</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COLLECTION</strong></td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>CREATE TYPE address_nested_type AS TABLE OF VARCHAR2(30));</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE person</td>
</tr>
<tr>
<td></td>
<td>(ssno NUMBER PRIMARY KEY, name VARCHAR2(30), paddress address_nested_type)</td>
</tr>
<tr>
<td></td>
<td>NESTED TABLE paddress STORE AS paddress_table;</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO PERSON</td>
</tr>
<tr>
<td></td>
<td>VALUES (99999999,'Peter Pan', address_type('Never','Never','Land'));</td>
</tr>
<tr>
<td></td>
<td>SELECT * FROM THE (SELECT paddress FROM person WHERE ssno = 99999999)</td>
</tr>
</tbody>
</table>
3. ORDBMS EXTENSIBILITY ARCHITECTURE

Though UDT support is a quantum leap for the ORDBMS, it still falls short for applications requiring complex type implementations where content might be stored as BLOBS. The content in this case is opaque to the DBMS. Complex content manipulation is a part of applications required to intelligently access data stored as time series, image, text, geospatial locations, dynamic Web pages, documents, spreadsheets, mail messages, and other data. This requirement has given rise to the ORDBMS extensibility architecture.

Some points to note:

1. SQL-99 UDTs form the cornerstone of an extensible ORDBMS system.
2. There is a rigorous specification for SQL-99. For the extensibility architecture, there is none. The major ORDBMS vendors follow some general design guidelines. Specific implementations differ.

3.1 EXTENSIBILITY BASICS

3.1.1 An Extensible Data Management Architecture

The Extensible Data Management Architecture is being implemented using three major approaches. The different approaches are:

The Universal Server Approach

The “universal server” approach extends RDBMS server capabilities to understand, store, and manage complex data natively in the database itself. Informix, IBM, and Oracle are all implementing this approach.

An “extended universal server” accommodates the fact that there may be very good reasons (such as performance) for not storing all the data in the DBMS. So the DBMS must also be able to efficiently access data stored on external files. Large data values – images, for example – can be stored externally, and a pointer to each image file is stored inside the database as a column value. An additional step is enabling the DBMS to also manage and ensure the integrity of the external data.

The Middleware Approach

Another approach is to use middleware that coordinates and executes requests across multiple, heterogeneous servers (RDBMS, text search engine, image system, and flat files); the data itself is managed within each specialized server. The middleware provides a unified view of the data, executes the global optimization of user queries, and provides global transaction management. There are two types of middleware in an extensible data management architecture. Both types use the SQL API and provide drivers out the back end to access each supported server. One is database middleware, such as IBM’s DataJoiner and Sybase’s OmniConnect, for integrated access to heterogeneous data. Sybase extends this approach to the Adaptive Server Architecture.

Microsoft’s OLE DB and DCOM, and other object request brokers (ORBs), on the other hand, represent another type of middleware: application middleware. OLE DB is an interface that was designed to provide universal access to data. It “componentizes” DBMS functionality, breaking it up into components that can run in the middleware space or in the operating system, such as query processors, optimizers, and transaction managers. One issue with OLE DB in particular and ORBs in general is the ability of third-party software vendors to provide competent database functionality such as global query...
processing and optimization. Given the amount of effort and R&D investment that have gone into optimization algorithms on the part of the major RDBMS vendors, it is not clear that middleware will offer comparable functionality and performance.

The Object Layer Approach
The object layer is an extensible data architecture providing integrated object views and object functionality at the application level. This can encompass client cache management, pointer navigation among objects, local execution of functions, and local query optimization. Object DBMSs are clearly focused here, including persistent storage of objects created by the application. In the case of the RDBMS, the object layer could include the ability to map objects in the database so that relational data can be materialized in the form of native C or C++ objects, Java objects, and so on. The benefits of this approach are a tighter integration between the data manager and the application development language and the potential for better performance. IBM is addressing the object layer through its client object-support development effort. Oracle is implementing this using client-cache management in Oracle 8i.

This section concentrates on the universal server approach and provides a list of desired features for extending the universal server to provide the ability to construct complex applications in a disparate data environment.

3.1.2 Universal Server (ORDBMS) Extensibility Features
As mentioned before, the SQL-99 object types form the core for extending the ORDBMS but not all desired features are part of the SQL-99 specification.

The desired features (including SQL-99) are:

Extensible Type System
An extended RDBMS must support user-defined datatypes (UDTs) at both the column and row level. Prior sections of this paper undergo an in-depth treatment of these distinct or abstract data types.

User-Defined Functions
User-defined functions (UDFs) define methods for manipulating data and are an important adjunct to UDTs. An extended RDBMS should provide significant flexibility in this area, such as allowing UDFs to return complex values that can be further manipulated (such as tables), execution options so that the user can decide whether performance or security is more important when running UDFs, and support the overloading of function names to simplify application development.

Index Structures
Traditional RDBMSs use B-tree (binary tree) indexes to speed access to scalar data. With the ability to define more complex data types in the RDBMS, specialized index structures are required for efficient access to data. Some extended RDBMSs are beginning to support additional index types, such as R-trees (region trees) which form fast access to two and three dimensional data, and the ability to index on the output of a function.

Optimizer
The query optimizer is the heart of the RDBMS performance and must be extended with knowledge about how to execute UDFs efficiently, take advantage of new index structures, transform queries in new ways, and navigate among data using references. Successfully
opening up such a critical and highly tuned DBMS component and educating third parties about optimization techniques is a major challenge for DBMS Vendors.

Other Extensions
Other important extensions are support for large-object storage either inside the database or outside in external files, the ability to apply business rules and integrity constraints to new data types, recursive queries to support complex-data relationships, and extended language support in the server. Extended RDBMSs must support the SQL-99 standard plus additional languages for writing UDFs and stored procedures, such as 3GLs and Java.

3.2 SQL-99 SUPPORT FOR THE EXTENSIBLE ORDBMS

<table>
<thead>
<tr>
<th>ORDBMS Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDTs</td>
<td>Y</td>
</tr>
<tr>
<td>Support for strong typing</td>
<td>Y</td>
</tr>
<tr>
<td>Support for hierarchies of type and inheritance</td>
<td>Y</td>
</tr>
<tr>
<td>Data replication support for UDTs</td>
<td>N</td>
</tr>
<tr>
<td>User-defined Functions</td>
<td>Y</td>
</tr>
<tr>
<td>Function Overloading</td>
<td>Y</td>
</tr>
<tr>
<td>Function resolution based on multiple attributes</td>
<td>Y</td>
</tr>
<tr>
<td>Extensible indexing system</td>
<td>N</td>
</tr>
<tr>
<td>Extensible query optimizer</td>
<td>N</td>
</tr>
<tr>
<td>Support for large objects (LOBS)</td>
<td>Y</td>
</tr>
<tr>
<td>Support for external data</td>
<td>N</td>
</tr>
<tr>
<td>Integrated searchable content</td>
<td>Y</td>
</tr>
<tr>
<td>Extended language support</td>
<td>Y</td>
</tr>
<tr>
<td>3GLs</td>
<td>Y</td>
</tr>
<tr>
<td>4GLs</td>
<td>N</td>
</tr>
<tr>
<td>Facilities (API developer's kit for adding extensions</td>
<td>N</td>
</tr>
<tr>
<td>System Management support for extensions</td>
<td>N</td>
</tr>
</tbody>
</table>

3.3 ORDBMS EXTENSIBILITY IMPLEMENTATIONS

Three of the leading DBMS vendors (IBM, Informix, and Oracle) have extended their conventional relational DBMSs to become ORDBMSs. These servers provide the mechanisms to extend the data storage capabilities of the database and the functionality of the DBMS itself. However, each vendor implemented the universal server concept using different mechanisms. In DB2 the mechanisms are called Relational Extenders, in Informix they are called Data Blades, and in Oracle they are called Cartridges.

3.3.1 DB2 Relational Extenders

IBM/DB2 relational extenders are built on the DB2 ORDBMS. It includes UDTs, UDFs, large objects (LOBS), triggers, stored procedures, and checks. As a next step, the DB2 relational extenders are used to define and implement new complex data types. The relational extenders encapsulate the attributes, structures, and behavior of these new data types, storing them in table columns of a DB2 database. The new data types can be accessed through SQL statements in the same manner as the standard DB2 data types. The DBMS treats these data types in a strongly typed manner, ensuring they are only used where data items or columns of that particular data type are anticipated. A DB2 relational extender is therefore a package consisting of a number of UDTs, UDFs, triggers, stored procedures, and constraints.
A complex relational extender may define a UDT with an elaborate internal structure that has multiple internal attributes. In this case, the logical view of the data may not necessarily be the same as the internal physical storage format. The internal attributes and structures may be hidden behind a public functional interface. In such an implementation, the column in the user's table would not contain the actual data of the UDT, but it would contain a handle that refers to an instance of the datatype. The interface would consist of a set of UDFs that take this handle as an argument and perform retrieving, storing, searching, and manipulating of the underlying attributes and structure. For example, an extender can be used to store rich text documents. The data may itself be stored in multiple tables, with keywords stored in separate tables and hidden indexes. Such an extender would typically include a UDF that encapsulates an advanced search engine which searches through the keyword tables. A UDF may even be a powerful function that accesses, retrieves, and manipulates data in another advanced server. The UDFs appear the same to the application as the standard DB2 SQL function.

### 3.3.2 Informix Datablades

Informix DataBlades are named after the special-purpose blades one can insert into a general-purpose knife. The DataBlades are standard software modules that plug into the database and extend its capabilities. A DataBlade is like an object-oriented package, similar to a C++ class library, that encapsulates a data object's class definition. The DataBlades not only let you add new and advanced datatypes to the DBMS, but they also let you specify new, efficient, and optimized access methods and processing options for these data types.

A DataBlade includes the data type definition (or structure) as well as the methods (or operations) through which it can be processed. It also includes the rules (or integrity constraints) that should be enforced, similar to a standard built-in data type.

A DataBlade is composed of a UDT, a number of UDFs, access methods, interfaces, tables, indexes, and client code. A DataBlade consists primarily of a new UDT. These data types are treated by the server in exactly the same way as the built-in types. Their values may be stored, queried, indexed, returned to applications, and passed as parameters. The UDTs can be based on built-in or user-defined types. Through the inheritance properties of the row types, relationships such as joins between related tables can be easily implemented. The distinct data types let you customize existing data types. For example, you can refine the definition of an existing data type to create a new specialized data type. The opaque data types are most flexible. They are implemented using C, C++, or Java code where the code defines how the occurrences of the data type have to be stored, indexed, and processed.

Each DataBlade can have a number of UDFs that operate on its data type. A function can operate on the data type of the DataBlade and on other data types (including other DataBlades). These functions are usually coded in Informix's stored procedure language (SPL), C, C++, or Java. Functions developed in C, C++, or Java are compiled and loaded in a shared object file or a dynamic link library (DLL). When the function is invoked, the shared object is linked into the database server and executes in its space. For the opaque data types, you must code a minimal set of required functions, and you can add a set of additional functions.
The access methods operate on the tables and indexes that are managed by the server. When defining new data types, you can use the existing access methods or implement new ones. An access method is defined to the server as a set of functions that it can call at various times when executing the query. These include functions, for example, to start scanning an index, get a next row, insert a new row, or delete an existing row. The access methods can be used to implement faster search methods for specialized data types such as R-trees, which are more efficient than B-trees for searching through 2D and 3D spatial data types.

An interface enables one DataBlade to share the services of another DataBlade. It is a collection of functions that conforms to a specific standard. Through this facility, various DataBlades can share common functions, such as search and retrieval. A DataBlade can store and manipulate its own definition data in tables and indexes in the database. This makes the whole DataBlade data-driven and easy to change, manage, and extend. Client code is application code that accesses the database through the DataBlade API library. It provides an interface through which the users can query, display, and modify the new data types.

Informix provides a DataBlade Developers Kit (DBDK), a comprehensive development environment for creating new data types and functions.

**3.3.3 Oracle Extensibility Architecture [Cartridges]**

Oracle8i gives application developers control over user-defined data types, not only enabling the capture of domain logic and processes associated with the data, but also enabling customization of the manner in which the server stores, retrieves, or interprets this data. The Oracle 8i database contains a series of database extensibility services, which enable the packaging and integration of domain types and behavior into a server-based, managed components. Such components are called specialized data types.

Specialized data types are the mechanism that extends the capabilities of Oracle 8i. Specialized data types are a safe, solution-oriented means to package domain-specific data and behavior, and integrate such packages with the server. For example, a spatial specialized data type may provide comprehensive functionality for a geography domain, such as being able to store spatial data, perform proximity comparisons on such data, and also integrate spatial data with the server, by providing the ability to index such data.

Specialized data types can have horizontal utility (such as imaging, time series, spatial data), or vertical focus (such as chemical structures, gene sequences or telephone networks). Oracle is building some of the horizontal specialized data types, such as ones for image, spatial, and text data (packaged as InterMedia). At the same time, the specialized data type programming interface is publicly available to Oracle’s partners and customers that are interested in building their own specialized types.

Normally, the database provides a set of services – for example the basic storage service, a query service, services for indexing, query optimization and so on. Applications use these services to avail themselves of database functionality.

The Oracle Extensibility Architecture makes these services customizable for specific application domains. When some aspect of a native service that the database provides is not adequate for specialized processing that a business requires, a specialized data type developer may provide a more specific implementation. For example, if a developer builds a Genomic Specialized Data Type for storing information on gene sequences, they may...
need the capability to create special indexes to query on genomes. Oracle 8i allows the creation of such indexes and directs the server to use these indexes when dealing with special queries.

3.3.4 Extensibility Features Comparison Matrix: Major ORDBMS Vendors

<table>
<thead>
<tr>
<th>Feature</th>
<th>INFORMIX</th>
<th>DB2/UDB</th>
<th>ORACLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDTs</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for hierarchies of types and inheritance</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Data replication support for UDTs</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>User-defined Functions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Function Overloading</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Function resolution based on multiple attributes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Extensible indexing system</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Extensible query optimizer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for large objects (LOBS)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for external data</td>
<td>Yes; access only</td>
<td>Yes; with management of file links</td>
<td>Yes; access only</td>
</tr>
<tr>
<td>Integrated searchable content</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Extended Language Support - 3GLs</td>
<td>Yes; C/C++/Java</td>
<td>Yes; C/C++/Java</td>
<td>Yes; C/C++/Java</td>
</tr>
<tr>
<td>Predefined extensions</td>
<td>Yes; over 20 DataBlades available from Informix and quite a few from partners</td>
<td>Yes; text, image, video, audio, and fingerprints from IBM; spatial from ESRI</td>
<td>Yes; text, spatial, video, image, and time series</td>
</tr>
<tr>
<td>Facilities (API developer’s kit for adding extensions)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3.4 EXTENSIBLE DATA TYPE EXAMPLE: ORACLE 8i

Typical database management systems support a few types of access methods (for example, B-trees, Hash Indexes, Bit Map Indexes) on some set of data types (numbers, strings, etc.). In recent years, databases have been used to store different types of data like text, spatial, image, video, and audio. In these complex domains, there is a need for indexing complex data types and specialized indexing techniques. For simple data types such as integers and small strings, all aspects of indexing can be easily handled by the database system. This is not the case for documents, images, video clips, and other complex data types that require content based retrieval. This requires extensible indexing.

**Extensible Indexing**

With extensible indexing, the application:

- Defines the structure of the domain index as a new indextype.
- Stores the index data either inside the Oracle database (in the form of index-organized tables) or outside the Oracle database.
- Manages, retrieves, and uses the index data to evaluate user queries.
When the database server handles the physical storage of the domain indexes, the extensible infrastructure must be able to:

- Define the format and content of an index. This enables the extensible system (Oracle calls these cartridges) to define an index structure that can accommodate a complex data object.
- Build, delete, and update a domain index. The cartridge handles building and maintaining the index structures.
- Access and interpret the content of an index. This capability enables the data cartridge to become an integral component of query processing. That is, the content-related clauses for database queries are handled by the data cartridge.

**User-Defined Operators**

Data cartridge developers find it useful to define domain-specific operators and integrate them into the Oracle8i server along with extensible indexing schemes such operators take advantage of while accessing data. Oracle8i provides a set of pre-defined operators which include arithmetic operators (+,-,*,/), comparison operators (=,>,<) and logical operators (NOT,AND,OR). These operators take as input one or more arguments (for operands) and return a result. They are represented by special characters (+) or keywords (AND).

The extensible or user-defined operators in Oracle8i are defined by names (i.e., Contains) or special characters. Like built-in operators, they take a set of operands as input and return a result. The implementation of the operator is provided by the user. After a user has defined a new operator, it can be used in SQL statements like any other built-in operator. For example, if the user defines a new operator CONTAINS which takes as input a text document and a keyword and returns TRUE if the document contains the specified keyword, we can write a SQL query as:

```
SELECT * FROM RegionalDocuments WHERE CONTAINS(Abstract, 'Africa and Irrigation');
```

User-defined operators can be invoked anywhere built-in-operators can be used – that is, whenever expressions can occur. For example, user-defined operators can be used in the following:

- Select List of a SELECT command
- Condition of a WHERE clause
- ORDER BY and GROUP By clauses

**Defining a Text Indexing Scheme**

The sequence of steps required to define a text indexing scheme using a text indextype are:

**Step 1**

Define code functions to support functional implementation of operators which eventually would be supported by the text indextype.

Suppose our text indexing scheme is in the context of a text data cartridge that intends to support an operator CONTAINS. The operator CONTAINS takes as parameter a text
value and a key and returns a boolean value indicating whether the text contained the key. The functional implementation of the operator is a regular function defined as:

```sql
CREATE FUNCTION TextContains (Text IN VARCHAR2, Key IN VARCHAR2) RETURN BOOLEAN AS
BEGIN
<code_block>
END TextContains;
```

**Step 2**
Create a new operator and define its specification, namely the argument and return data types, and the functional implementation.

```sql
CREATE OPERATOR Contains BINDING (VARCHAR2, VARCHAR2) RETURNS BOOLEAN USING TextContains;
```

**Step 3**
Define a type or package that implements the index interface `ODCIIndex`. This involves implementing routines for index definition, index maintenance, and index scan operations. The `ODCIIndex` interface is described in more detail in the section on extensibility interfaces below.

The index definition routines (insert, delete, update) maintain the text index when the table rows are inserted, deleted or updated.

The index scan routines (start, fetch, close) implement access to the text index to retrieve rows of the base table that satisfy the operator predicate. In this case the `CONTAINS(....)` forms a boolean predicate whose arguments are passed in to the index scan routines. The index scan routines scan the text index and return the qualifying rows to the system.

```sql
CREATE TYPE TextIndexMethods ( PROCEDURE create(....) .... );
CREATE TYPE BODY TextIndexMethods ( .... );
```

**Step 4**
Create the text index type schema object. The index type definition also specifies all the operators supported by the new index type and specifies the type that implements the index interface,
CREATE INDEXTE Type TextIndexType FOR CONTAINS(VARCHAR2, VARCHAR2)
USING TextIndexMethods;

Suppose that the text indextype presented in the previous section has been defined in the
system. The user can define text indexes on text columns and use the associated
CONTAINS operator to query text data. Further, suppose a DOCUMENTS table is
defined as follows:

CREATE TABLE DOCUMENTS (doc-name VARCHAR2(64), doc-id INTEGER, Abstract
VARCHAR2(2000);

A text domain index can be built on the abstract column as follows:

CREATE INDEX AbstractIndex ON DOCUMENTS(Abstract) INDEXTYPE IS
TextIndexType;

The text data in the abstract column can be queried as:

SELECT * FROM DOCUMENTS WHERE CONTAINS(Abstract, 'Disbursement');

The query execution will use the text index on abstract to efficiently evaluate the
CONTAINS predicate.

Extensible Optimizer
The extensible optimizer functionality enables authors of user-defined functions to create
statistics collection, selectivity, and cost functions. This information is used by the
optimizer in choosing a query plan. The cost-based optimizer is thus extended to use the
user-supplied information; the rule-based optimizer is not being changed.

Note: The trained eye will notice that the above example forms the core implementation infrastructure of
Oracle Text (Prior to the Oracle 9i implementation, Oracle Text was called Intermedia Text).

ODCIIIndex Interface
The ODCIIIndex (Oracle Data Cartridge Index) interface consists of the following
categories of methods: index definition methods, index maintenance methods, and index
scan methods.

Index Definition Methods
These methods allow specification of create, alter, drop and truncate behaviors.

- ODCICreate() is called when a CREATE INDEX statement is issued that
  references the indextype. Upon invocation, any physical parameters specified as
  part of the CREATE INDEX ... PARAMETERS (...) statement are passed in
  along the description of the index. A typical action of this routine is to create
tables/files to store the index data. Further, if the base table is not empty, this
routine should build the index for the existing data in the indexed columns.

- ODCIAlter() is invoked when a domain index is altered using an ALTER INDEX
  statement. The description of the domain index to be altered is passed in along
with any specified parameters. In addition, this routine handles ALTER with REBUILD option which supports rebuilding the domain index. The precise behavior in these two cases are defined by the developer of the indextype.

- **ODCItruncate()** is called when a TRUNCATE statement is issued against a table that contains a column or OBJECT type attribute indexed by the indextype. After this routine executes, the domain index should be empty.

- **ODCIDrop ()** is invoked when a domain index is destroyed using a DROP INDEX statement.

### Index Maintenance Methods
These methods allow specification of index insert, update, and delete behaviors.

- **ODCInsert()** is called when a record is inserted in a table that contains columns or object attributes indexed by the indextype. The new values in the indexed columns are passed in as arguments along with the corresponding row identifier.

- **ODCIDelete()** is called when a record is deleted from the table that contains the column or object type attributes indexed by the indextype. The old values in the columns are passed in as arguments along with the corresponding row identifier.

- **ODCICreate()** is called when a record is updated in a table that contains columns or OBJECT attributes indexed by the indextype. The old and new values in the indexed columns are passed in as arguments along with the row identifier.

### Index Scan Methods
These methods allow specification of index based implementation for evaluating predicates containing operators. An index scan is specified through three routines (start, fetch, and close) which can perform initialization, fetch rows satisfying the predicate, and clean up once all rows satisfying the predicate are returned.

- **ODCISstart()** is invoked to initialize any data structures and start an index scan. The index related information and the operator related information are passed in as arguments. A typical action performed when ODCISstart () is invoked is to parse and execute SQL statements that query the tables storing the index data. It could also generate some set of result rows to be returned later when ODCIFetch() is invoked.

- **ODCIFetch()** returns the next row identifier of the row that satisfies the operator predicate.

- **ODCIClose()** is invoked when the cursor is closed or reused. In this call the index implementor can perform any clean-up.

### ODCIStats Interface
The ODCIStats (Oracle Data Cartridge Statistics) interface consists of the following categories of methods: user-defined statistics functions, user-defined selectivity functions, user-defined cost functions and type methods, and user-defined cost for domain indexes.

#### User-Defined Statistics Functions

- **Collect_Stats** and **Delete_Stats**

These are invoked when collecting or deleting statistics using the ANALYSE call.
User-Defined Selectivity Functions
A user-defined selectivity function can be specified for a user-defined Boolean function or operator. The function returns a value for the computed selectivity as a percent (whole number between 0 and 100 inclusive).

User-Defined Cost for Functions and Type Methods
The cost of a function or type method is computed by a call to `func_cost()`.

User-Defined Cost for Domain Indexes
The cost for using a domain index is computed by a call to `index_cost()`.

4. IMPLEMENTING OBJECT DATABASE TECHNOLOGY: LOOKING AHEAD

Before embarking on any ambitious object database implementation, an enterprise needs to look at various facets of the business a few of which are application areas, toolsets, skills mix etc.

4.1 Application Areas and Database Objects

There are five main types of application areas in a mid to high end enterprise. They are:

**ERP Systems (OLTP)**
These core apps include HR, Travel, Financial Management, etc.

For ERP systems, the vendors are by definition constrained to the ERP database structure. Although ERPs sit on top of major ORDBMSs, the move to SQL-99 objects will be daunting given the mammoth engineering work required to support all ORDBMSs. The problem is further exacerbated by the fact that ORDBMSs are building variants of SQL-99 which makes it difficult for the ERP vendor to support the SQL-99 object model.

**Core Business Systems and Reference Data (OLTP)**
These applications form the basic bread and butter of the enterprise. Examples of core applications include a gene sequencing and publishing system (Genomic Company), a bond portfolio rebalancing system (Investment Bank) and a Loan System (Real Estate Lending Company).

The systems described above are typically RDBMS implementations with considerable complexity which includes hundreds of thousands of tables and other stored objects (functions, procedures, triggers, etc.)

SQL-99 objects can cut down the complexity in these systems for two major reasons. First, the RDBMS decomposition issue gets addressed. For example, a logical object gets decomposed into several 3rd Normal Form tables in a relational table whereas it would translate to a single or very few types in an ORDBMS. Second, there is a substantial portion of code (reference) data which manifests itself in form of numerous tables in the relational database. The reference table proliferation can be addressed by SQL-99 collection types.
Note: The move to the SQL-99 ORDBMS is easier said than done. Please refer to the “Tools and Standard Compliance” sub-sections for issues acting as impediments to a full blown SQL object implementation.

Messaging Systems (Enterprise Application Integration)
Given traditional messaging formats (EDI) are based on complex structures, the ORDBMS SQL-99 type system becomes the ideal vehicle for implementing these applications. Messaging systems can be fully encapsulated in server code hiding the type complexity from invoking routines. Front-end GUI object impedance mismatch issues disappear as the majority of these systems are predominantly server side implementations.

Knowledge Management Systems (Enterprise Intelligence)
Some examples of this infrastructure include document management systems, work group and collaboration system, data warehousing and data mining systems.

Document Management Systems and Search Engines
ORDBMS vendors provide canned Document Management implementations on top of the SQL-99 object type system using the ORDBMS extensibility services. Oracle Text is an example of these canned implementations.

It is also possible to build a fully functional Document Management System based on the extensibility infrastructure various ORDBMS vendors provide (Relational Extenders in DB2/UDB, DataBlades in Informix, Extensible Type System or Cartridges in Oracle 8i). However, building one from ground up is not desirable from a cost and viability perspective.

Workgroup and Collaboration Systems
Although it is possible to build these specialized types using the ORDBMS extensibility suites, the author would suggest using either evaluating canned ORDBMS types or encapsulate existing non-RDBMS based specialized systems like Lotus Notes or web based groupware tools.

Depending on the requirements, these applications may also benefit from pure ODBMS implementations because of robust object locking and object lifetime support in the ODBMS.

Data Warehousing Systems (OLAP)
The standard RDBMS has evolved in terms of both structure and performance to house VLDB data. The structures (star /constellation schemas), the indexes (bitmap), and general performance (parallel loads, parallel queries, etc) together provide the required infrastructure to develop and maintain a typical Data Warehouse. As such, the author does not see any major benefits of moving these applications to SQL-99 objects.

Data Mining Systems
The author would not recommend building these systems from ground up. While evaluating data mining tools, an ideal system would be an extensible or specialized canned type fully integrated within the ORDBMS (similar to Oracle's Intermedia Text). The second choice would be a canned system based on the relational model.

Specialized Systems
These systems include medical records systems, distance learning, and multimedia (audio, video, animation) applications. Some of these systems are so specialized that it may be
worthwhile for enterprises to look at pure object databases (ODBMS) implementations (performance may be one reason). The choice would boil down to price, skills mix and integration requirements for these ODBMSs.

4.2 Tools and Database Objects

Traditionally tools have lagged supporting the functionality of the ORDBMS. Most vendors first provide support for 3GL languages (C, C++) and their own proprietary engines (e.g., Oracle's PL/SQL). Since Java is gaining momentum in the enterprise, the vendors have also starting building java extensions (or translators) for the SQL-99 tools. The 4GL development space is seeing a flurry of activity as SQL-99 objects are steadily creeping into GUI development environments (Oracle's Developer 6i, Sybase's Power Designer, etc.). This will help solve the translation or impedance mismatch problem.

In the design space, object database modeling finds its root in ORM (a conceptual Object-role modeling tool). The formal object-role modeling language (FORML) encapsulates ORM, taking a systematic, rigorous approach to capturing business concepts. The new object modeling standard is UML. UML is based on the Unified model that includes conceptualization and requirements analysis covering conceptual modeling with mappings into classes and components. UML's classes are roughly equivalent to ORDBMS types and methods. Though non-UML ORDBMS design tools are available (Erwin OR/Compass, Infomodeler), UML is the de facto choice and highly recommended. Rational Rose and Designer 6i tools both provide UML design capabilities. Rational Rose is the undisputed industry leader. For pure Oracle based implementations, however, Designer 6i may provide tighter integration.

Note: It is highly recommended that an enterprise take a deep look at the toolsets and their capabilities as part of the evaluation phase of an ORDBMS implementation.

4.3 SQL-99 and Database Objects

Vendors are implementing their own variants of the SQL-99 standards making it difficult for enterprises to go forward with SQL object based applications. Some core object features are missing in vendor implementations. For example, Oracle 8i does not implement SQL-99 type inheritance capability. An ORDBMS should ideally implement SQL-99 and extensibility services. Some desirable features are:

- Sub-typing
- Inheritance
- Dynamic Polymorphism
- Declarative Constraints on attributes of Object Types
- Type Replication
- Ability to specify default constructor functions

Note: It is highly recommended that an enterprise take a deep look at the type and object model features as part of the evaluation phase of an ORDBMS implementation.
CLOSING NOTE

As a closing note, the relational database is undergoing a slow metamorphosis. The objects are creeping in, slowly but surely. This paper provides a tutorial and some guidelines for an enterprise to be ready for this new kid on the block.

For questions, please email me at:

pbashir@worldbank.org

REFERENCES

1. Modelling Object/Relational Databases by Seth Grimes, DBMS, April 1998
2. Object-Relational Databases: Their Time has Come…Almost, Dr. Paul Dorsey, Select Magazine, July 1998
4. Object Oriented Databases, Doss R. Keese, Bowie State University, INSS 690, July 31, 1999
5. Is a Hybrid Database in your future?, Rick Cook, SunWorld, February 1997