COMPARISON BETWEEN OBJECT-RELATIONAL AND OBJECT-ORIENTED DATABASES

Daniela GOTSEVA, Loie Naser Mahmud NIMRAWI
Technical University of Sofia, Bulgaria

Abstract. This paper discusses some concepts related to the object-relational and object-oriented database system such as object identity, row types, user-defined types (UDTs), user-defined routines, polymorphism, subtypes and supertypes, persistent stored modules, and large objects. At the end of the paper exists comparison between ORDBMS and OODBMS.

Keywords: object-oriented database, object-relational database, database design, object query language

1. Introduction to object-relational database system

Until recently, the choice of DBMS seemed to be between the relational DBMS and the object-oriented DBMS. However, vendors of RDBMS products are still conscious of the threat and promise of the OODBMS. They agree that their systems are not currently suited to the advanced applications, and that added functionality is required.

The examining of the advanced database applications that are emerging, due to find extensive use of many object-oriented features such as a user-extensible type system, encapsulation, inheritance, polymorphism, dynamic binding of method, complex objects including non-first normal form objects, and object identity. The most obvious way to remedy the shortcomings of the relational model is to extend the model with these types of features. This is the approach that has been taken by many prototype extended relational systems, although each has implemented different combinations of features. Thus, there is no single extended relational model; rather, there are a variety of these models. However all the models do share the same basic relational tables and query language, all incorporate some concept of 'object', and some have the ability to store methods (or procedures or triggers) as well as data in the database.

Various terms have been used for systems that have extended the relational data model. The original term that was used to describe such systems was the extended relational DBMS (ERDBMS). However, in recent years the more descriptive term Object-Relational DBMS has been used to indicate that the system incorporates some notion of 'object', and more recently the term Universal Server or Universal DBMS (UDBMS) has been used. It stands for Object-Relational DBMS (ORDBMS). Three of the leading RDBMS vendors (Oracle, Informix, and IBM) have all extended their systems to become ORDBMSs, although the functionality provided by each is slightly different. The concept of the ORDBMS, as a hybrid of the RDBMS and OODBMS, is very appealing, preserving the wealth of knowledge and experience that has been acquired with the RDBMS. Some analysts predict the ORDBMS will have a 50% larger share of the market than the RDBMS [1, 2].

The main advantages of extending the relational data model come from reuse and sharing. Reuse comes from the ability to extend the DBMS server to perform standard functionality centrally, rather than have it coded in each application. For example, applications may require spatial data type that represents points, lines, and polygons, with associated functions that calculate the distance between two points, the distance between a point and a line, whether a point is contained within a polygon, and whether two polygonal regions overlap, among others. If it is possible to embed this functionality in the server, it saves having to define them in each application that needs them, and consequently allows the functionality to be shared by all applications. These advantages also give rise to increased productivity both for the developer and for the end-user.

2. OODB systems perspectives

Database systems are primaries concerned with the creation and maintenance of large, long-lived collections of data. Modern database systems are characterized by their support of the following features:

- **A data model**: A particular way of describing data, relationships between data, and constraints on the data.

- **Data persistence**: the ability for data to outlive the execution of a program, and possibly the lifetime of the program itself.
- **Data sharing**: The ability for multiple applications (or instances of the same one) to access common data, possibly at the same time.
- **Reliability**: The assurance that the data in the database is protected from hardware and software failures.
- **Scalability**: The ability to operate on large amount of data in simple ways.
- **Security and integrity**: The protection of the data against unauthorized access, and the assurance that the data conforms to specified correctness and consistency rules.
- **Distribution**: The ability to physically distribute a logically interrelated collection of shared data over a computer network, preferably making the distribution transparent to the user.

In contrast, traditional programming languages provide constructs for procedural control and for data and functional abstraction, but lack built-in support for many of the above database features. While each is useful in their respective domains, there exist an increasing number of applications that require functionality from both database system and programming languages. Such applications are characterized by their need to store and retrieve large amounts of shared, structured data.

In the last two decades, there has been considerable effort invested in developing systems that integrate the concepts from these two domains. However, the two domains have slightly different perspectives that have to be considered and the differences addressed [1].

Perhaps two of the most important concerns from the programmers' perspective are performance and ease-of-use, both achieved by having a more seamless integration between the programming language and the DBMS than that provided with traditional database systems. With a traditional DBMS:

- It is the programmer's responsibility to decide when to read and update objects (records)
- The programmer has to write code to translate between the application's object model and the data model of the DBMS (for example, relations) which might be quite different. With an object-oriented programming language, where an object may be composed of many sub-objects represented by pointers, the translation may be particularly complex. In fact, it has been claimed that a significant amount of programming effort and code space is devoted to this type of mapping, possibly as much as 30% as noted above. If this mapping process can be eliminated or at least reduced, the programmer would be freed from this responsibility, the resulting code would be easier to understand and maintain, and performance may increase as a result.

It is the programmer's responsibility to perform additional type-checking when an object is read back from the database. For example, the programmer may create an object in the strongly-typed object-oriented language java and store it in a traditional DBMS. However, another application written in a different language may modify the object, with no guarantee that the object will conform to its original type.

These difficulties stem from the fact that conventional DBMS have a two-level storage model: the application storage model in main or virtual memory, and the database storage model on disk. In contrast, an OODBMS tries to give the illusion of a single-level storage model, with a similar representation in both memory and in the database stored on disk.

Although the single-level memory model looks intuitively simple, to achieve this illusion the OODBMS has to cleverly manage the representations of objects in memory and on disk objects, and relationships between objects, are identified by object identifiers (OIDs). There are row types of OIDS: logical OIDs that are independent of the physical location of the object on disk, and physical OIDs that encode the location. In the former case, a level of indirection is required to look up the physical address of the object on disk. An OID different in size from a standard in-memory pointer that need only be large enough to address all virtual memory, in both cases. Thus, to achieve the required performance, an OODBMS must be able to convert OIDs to end from in memory pointers. This conversion technique has become known as 'pointer swizzling' or 'object faulting', and the approaches used to implement it have become varied, ranging from software-based residency checks to page faulting schemes used by the underlying hardware [1, 5].

3. SQL3

The object-oriented features proposed in the next SQL standard, SQL3, covering:

- Type constructors for row types and reference type.

- User-defined types (distinct types and structured types) that can participate in supertype/subtype relationships.
- User-defined procedure, functions and operators.
- Type constructors for collection types (arrays, sets, and lists).

Support for large objects Binary Large Objects (BLOBs) and Character Large Objects (CLOBs).

3.1. Object identity

Each relation has an implicitly defined attribute named OID that contains the tuple's unique identifier, where each OID value is created and maintained by postgres. The OID attributes can be accessed but not updated by user queries. Among other users, the OID can be used as a mechanism to simulate attribute types that reference tuples in other relation. The relation name can be used for the type name because relations, types, and procedures have separate name spaces [1].

3.2. Row types

A row type is a sequence of field name/date type pair that provides a data type that can represent the types of rows in tables, so that complete rows can be stored in variables, passed as arguments to routines and returned as return values from function calls. A row type can also be used to allow a column of a table to contain row values [1, 3].

3.3. User-defined types (UDTs)

It refers to user-defined types as Abstract Data Types (ADTs), that may be used in the same way as the built-in types (for example CHAR, INT, FLOAT). UDTs are subdivided into two categories: distinct types and structured types. The simplest type of UDT in SQL3 is the distinct type, which allows differentiation between the same underlying base types. In its more general case, a UDT definition consists of one or more attribute definitions. It has also been proposed that a UDT definition consist additionally of routine declarations. If this proposal is not accepted, these declarations from part of the schema. In what follows, it can be assumed that UDT definition may contain routine declarations. It stands to routines and operators generically as routines. In addition, within the UDT definition it can be also define the equality and ordering relationships for the UDT.

3.4. User-defined routines

User-defined routines (UDRs) define methods for each manipulating data and are an important adjunct to UDTs. An ORDBMS should provide significant flexibility in this area, such as allowing UDRs to return complex values that can be further manipulated (such as tables), and support for overloading of function names to simplify application development. In SQL3, UDRs may be defined as a part of a UDT or separately as part of a schema. An SQL-invoked routine may be a procedure, function, or iterative routine. It may be externally provided in a standard programming language such as C/C++, or defined completely in SQL using extensions that make the language computationally complete. A SQL-invoked procedure is invoked from a SQL CALL statement. It may have zero or more parameters, each of which may be an input parameter(IN), an output parameter (OUT),or both an input and output parameter (INOUT), and it has a body if it is defined fully within SQL. A SQL-invoked function returns a value; any specified parameter [1, 3, 4].

3.5. Relations and inheritance

A relation inherits all attributes from its parents unless an attribute is overridden in the definition. Multiple inheritances are supported, however, if the same attribute can be inherited from more than one parent and the types of the attributes are different, the declaration is disallowed. Key specifications are also inherited [1, 4].

3.6. Polymorphism

Different routines may have the same name, that is routine names may be over-loaded, for example to allow aUDT subtype to redefine a method inherited from a supertype, subject to the following constraints:

- No two functions in the same schema are allowed to have the same signature, that is, the same number of arguments, the same data types for each argument, and the same return data type.
- No two procedures in the same schema are allowed to have the same name and the same number of parameters.

The current draft SQL3 proposal uses a generalized object model, so that the types of all arguments to a routine are taken into consideration when determining which routine to invoke, in order from left to right. Where there is not an exact match between the data type of the argument and the data type of the parameter specified, type precedence list are used to determine the closest match. the exact rules for routine determination for a given invocation are relatively complex [1].

3.7. Subtypes and supertypes

SQL3 allows UDTs to participate in a subtype/supertype hierarchy. A type can have more than one supertype (that is, multiple inheritance is supported), and more than one subtype. A subtype inherits all the attributes and behavior of its supertypes and it can define additional attributes and functions like any other UDT and it can override inherited function [1].

3.8. Persistent stored modules

A number of new statement types have been added in SQL3 to make the language computationally complete, so that object behavior (methods) can be stored and executed from within the database as SQL statements. Statements can be grouped together into a compound statement (block), with is own local variables. Some of the additional statements provided in SQL3 are:

- An assignment statement that allows the result of an SQL value expression to be assigned to a local variable, a column, or an attribute of a UDT.
- An IF…THEN…ELSE…END IF statement that allows conditional processing.
- A CASE statement that allows the selection of an execution path based on a set of alternatives.
- A set of statements that allows repeated execution of a block of SQL statements. The iterative statements are FOR, WHILE, and REPEAT.
- A CALL statement that allows procedures to be invoked and RETURN statement that allows an SQL value expression to be used as the return value from an SQL function [2].

3.9. Large Objects

A Large Object is a table field that holds a large a mount of data as a long text file or a graphics file. There are three different types of large object data types defined in SQL3:

- Binary Large Object (BLOB), a binary string that does not have a character set or collation association.
- Character Large Object (CLOB) and National Character Large Object (NCLOB), both character strings.

The SQL large object is slightly different from the original type of DLOB that appears in many current database systems. In such systems, the BLOB is non-interpreted byte stream, and the DBMS does not have any knowledge concerning the content of the BLOB or its internal structure. This prevents the DBMS from performing queries and operations on inherently rich and structured data types, such as images, video, word processing documents, or web pages. Generally, this requires that the entire BLOB be transferred across the network from the DBMS server to the client before any processing can be performed. In contrast, the SQL3 large object does allow some operations to be carried out in the DBMS server.

The standard string operators, which operate on characters strings and return character strings, also operate on character large object string, such as:

- The concatenation operator, (string1|| string2), which returns the character string formed by joining the character string operands in the specified order.
- The character substring function, SUBSTRING (string FROM startops FOR length), which returns a string extracted from a specified string from a start position for a given length.
- The fold function, UPPER (string) and LOWER (string), which convert all characters in a string to upper/lower case.
- The length function, CHAR+LENGTH (string), which return the length of the specified string.
- The position function, POSITION(string1 IN string2), which returns the start position of string1 within string2.

However, CLOB strings are not allowed to participate in most comparison operations, although they can participate in a LIKE predicate, and a comparison or quantified comparison predicate that uses the equal (=) or not equal(<>)operators.

4. Comparison of ORDBMS and OODBMS

It can be conclude the treatment of Object-Relational DBMS and Object-Oriented DBMS with a brief comparison of the two types of system. It can be assumed that future ORDBMSs will be compliant with SQL3 [1]. The results are shown in Table 1.

5. Conclusion

The concept of the ORDBMS is as a hybrid of the RDBMS and OODBMS. The object-oriented features proposed in SQL3 support type constructors for row types and reference types, user-defined types, user-defined procedures, functions and operators, and support for large objects Binary Large Objects (BLOBs) and Character Large Objects (CLOBs).

Table 1. Comparison Between ORDBMS and OODBMS

Feature	ORDBMS	OODBMS
Encapsulation	Supported through UDTs	Supported and broken for queries
Inheritance	Supported (separate hierarchies for UDTs and tables)	Supported
Polymorphism	Supported (UDF invocation based on the generic function)	Supported as in an object oriented programming model language.
Relationships	Strong support with user-defined referential integrity constraints	Supported (for example, using class libraries)
Integrity constraints	Strong support	No support
Recovery	Strong support	Supported but degree of support differs between products
Advanced transaction models	No support	Supported but degree of support differs between products
Security, integrity, and views	Strong support	Limited support

References

1. Connolly, T., Begg, C. (2005) *Database systems of particular approach to Design, Implementation and management*. ISBN 0321210255, Addison Wesley
2. Embley, D. (2003) *Object Database Development: concepts and principles.* Addison Wesley, ISBN 978-0201258295
3. *Database Management Systems Revisited, An Updated DACS State-of-the-Art Report.* Prepared by: Gregory McFarland, Andres Rudmik, and David Lange Modus Operandi, Inc, 1999
4. Cattell, R.G.G. (1997) *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, ISBN 978-0201547481
5. Date, C.J., Darwen, H. (1992) *Into the Great Divide*. Addison-Wesley, ISBN 0-201-82459-0

Received in August 2012