Object-oriented Database Systems: the Notion and the Issues

(extended abstract)

Klaus R. Dittrich

Forschungszentrum Informatik (FZI) an der Universität Karlsruhe
Haid - und - Neu - Str. 10-14, D-7500 Karlsruhe

A *database system* is a collection of stored data together with their description (the *database*) and a hardware/software system for their reliable and secure management, modification and retrieval (the *database management system*, DBMS).

A database is supposed to represent the interesting semantics of an application (the *miniworld*) as completely and accurately as possible. The *data model* incorporated into a database system defines a framework of concepts that can be used to express the miniworld semantics.
It comprises

- basic data types and constructors for composed data types,
- (generic) operators to insert, manipulate, retrieve and delete instances of the actual data types of a database,
- implicit consistency constraints as well as (eventually) mechanisms for the definition of explicit consistency constraints that further reflect the miniworld semantics as viewed by the database system.

As usual, types have to be defined before instances of them can be created (the collection of defined types – sometimes together with the set of explicit consistency constraints – forms the *database schema*). Every database thus adheres to the schema defined for it, and both together, the schema and the actual data provided by the users (and stored in instances) capture the miniworld semantics.

We can therefore distinguish the following two classes of semantics:

- the semantics of the miniworld itself,
- the semantics of the miniworld as represented within the database.

Let us assume that a database *correctly* reflects the intended miniworld semantics (careful database design!). Due to the rigid framework of data models, there will still remain a semantic gap between the miniworld and its database representation. In other words, it is usually impossible to represent *all* interesting semantics within a database. The "remainder" has to be captured by the application programs using the database and/or it is part of the (hopefully meaningful !) interpretation of the result of database queries by the user himself.

However, the ultimate goal of database systems is to provide for concepts that allow to keep the semantic gap as small as possible and thus permit to represent most of the salient semantics in the database itself.

What are object-oriented database systems ?

This is where object-oriented database systems come in. On a level of abstractions, the semantics of a given miniworld may be modelled as a set of entities and relationships amongst them. While classical business/administration types of database applications tend to deal with rather "simple" entities (i.e. those where only a few properties like name, age, salary are of interest), this is no longer true for applications like VLSI-design, image processing, office automation and the like. In these areas, entities usually show very complex internal structures and may comprise larger numbers of (possibly substructured) properties. Similar observation can be made with respect to relationships.

Today's database systems are mostly based upon one of the now classical data models (hierachical, network, or relational) or one of their derivations. As these models are all tailored to account for the representation of rather simple entities only, the semantic gap tends to become large when complex entities need to be dealt with. This is at least partially due to the fact that in these cases *one* conceptual miniworld entity has to be represented by *a number* of database objects (e.g. records, tuples, ...).

This leads us to a first-cut informal definition of an object-oriented database system: it is based on a data model *that allows to represent one miniworld entity* (whatever its complexity and structure) *by exactly one object* (in terms of the data model concepts) *of the database.* Thus no artificial decomposition into simpler concepts is necessary in any case (unless the database designer decides to do so). Note that as entities might be composed of subentities which are entities in their own right, an object-oriented data model also has to allow for recursively composed objects.

Looking at things a little closer, we can in fact identify several *levels of object-orientation*:

(a) if the data model allows to define data structures to represent entities of any complexity, we call it *structurally object-oriented* (i.e. there are complex objects),

(b) if the data model includes (generic) operators to deal with complex objects in their entirety (in contrast to being forced to decompose the necessary operations into a series of simple object — e.g. tuple or homogeneous set of tuples — operations), we call it *operationally object-oriented*; as it is hardly meaningful without, we require that operational object-orientation includes structural object-orientation;

(c) borrowing types from the object-oriented programming paradigm, a data model may also incorporate features to define object types (again of any complexity) together with a set of specific operators (abstract data types); instances can then only be used by calling these operators, their internal structure may only be exploited by the operator implementations; we call systems based on this approach *behavioralley object-oriented.*

While structural object-orientation is not very useful without operational object-orientation, both (b) *and* (c) are within the range of object-oriented database systems; note that the scope is thus broader than with object-oriented programming languages which are concentrated on the paradigm sketched in (c) only. However, there seems to be agreement that even the (somewhat less advanced) operational object-orientation is a versatile solution for database systems (and might at least be used as a basis for the internals of abstract data types in (c)). By the way, concepts like property inheritance may be added to both (b) and (c).

There are at least two other directions where the notion of object-orientation is used; in our opinion, they do not contribute to the definition of object-oriented database systems, but naturally come along with such systems.

- *object-oriented implementation* : the (database) system as a piece of software is constructed as a set of abstract data type instances, i.e. a specific kind of modularization is applied, even non-object-oriented database systems may have an object-oriented implementation;
- *object-oriented user/programming interface* : the database system interface is presented to the user/application programmer in a fashion inspired by the object - oriented proramming paradigm; while this fits in very smoothly with an object - oriented database system (expecially of type (c)), it may also be provided on top of any other database system.

What are the issues of object-oriented database systems ?

At first glance, one might think that object-oriented database systems just offer a different kind of data model than traditional systems do. However, the more powerful concepts for modelling miniworld semantics result in a number of database issue to be at least reconsidered, if not extended or completely changed. The following list mentions a few of them in random order:

- Like various record-/tuple-oriented data models, many object-oriented data models have been and will be proposed. Some of these differ only slightly in style and/or expressive power, and there is currently no clear tendency towards one or a small number of generally "recognized" models.
- Together with complex objects, applications are often concerned with *object versions* (multiple representations of the same semantic entity, to account for different stages, different times of validity, alternative or hypothetical information etc.). Object-oriented database systems therefore need mechanisms to deal with versions.
- When manipulating objects comprising large bulks of data, transactions may become much longer than usual. New concepts are therefore needed to accommodate long-duration transactions, and in addition the concepts for recovery and consistency control and their relationship to the transaction concept have to be reconsidered.
- Protection mechanisms have to be based on the notion of object which is the natural unit of access control in this framework.
- For databases containing large numbers of data, archiving may become a major issue. Again, objects (and their versions, if any) form the natural unit for this activity.
- To work with an object-oriented database offen consists of first selecting one or a small number of objects and then performing local operations on them for a while. This suggests to provide, among others,
  - specialized access paths for complex objcts,
  - specialized storage structures for complex objects (that e.g. physically cluster logical objects or that use delta storage for versions),
  - object-oriented main memory buffering

  in the implementation of an object-oriented database system.
- High quality database design is a tedious job even for record-oriented database systems. It appears that it is even more difficult within the framework of object-oriented database systems. Appropriate design methodologies and tools that support them have to be developed.