

Tables as Entities

Alexandr Savinov
<http://conceptoriented.com>

First created: 23.06.05
Last update: 27.08.05

1. Introduction

Normally tables are viewed as containers for records and in data modelling they are used to describe a database schema. In this sense tables are interpreted as a structural element of the model while records in tables are used to represent the data semantics. This means that when we manipulate records then we change data semantics but if we manipulate tables then we change data structure. One interesting and important question in this context is how tables can be used to model data semantics and what is the role of tables in representing semantics. In particular, do we need tables at all and can we describe the data by means of records only. On the hand, do we need records at all and may be we can describe data by using only tables? And if we can use both tables and records for representing data semantics then when why should we prefer one mechanism to the other?

In this short paper we try to clarify this issue. We provide several examples which illustrate that tables can be used to represent normal entities just like records. We provide also examples where we show that records are frequently used as tables to represent elements that keep a number of other records. Finally, we describe what the concept-oriented model can offer to solve this problem.

2. Modelling Entities by Means of Tables

Let us assume that there is a table Products which stores a set of product items sold by some shop (Fig. 1). This table columns such as Id (product identifier), Name (product descriptive name), Type (one of several types of product), Price and some special columns for characterizing each individual product stored as a record in this table.

The product type is not arbitrary but is taken from a set of available type which are stored as records in table Types. This table has a column for identifier, a column for type name and possible some other columns characterizing each individual product type stored as a record. For example, table Types might include two records for Cars and for Houses. This means that there exist only two types of products and each individual record table Products can be marked either as a car or as a house. The most important thing at the moment is that *type of product is simply an ordinary record in table Types and nothing more*. Thus each product type represents one real entity or fact from the problem domain. In particular, a product type has an identifier or reference and each one type may have its characteristics stored in record fields. For example, a type might be assigned a name and a delivery type (if we want one type of product to be delivered in one way). We also can remove existing types of products and add them if necessary.

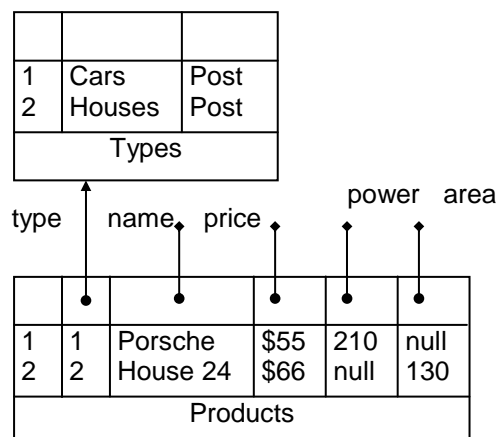


Fig. 1. Each product record has a type which is a record from another table (superconcept).

In this schema there is two tables Products and Types and hence there is two types of records: for representing concrete products and for representing concrete types of products. In such a schema each product is characterized by one and the same set of fields described in the definition of table Products (name, product type, price etc.) Let us now suppose that each product type has some characteristics which are absent or do not make sense for other product types. For example, cars may be characterized by their power and houses are characterized by their area. The simplest way to do so consists in defining these columns in table Products. In this case each one product will be characterized by both power and area even if this property does not makes sense for it. In the case we do not want some product to have a property we simply write NULL. For example, cars will have area equal to NULL while houses will have power assigned to NULL.

An alternative solution consists in introducing one table for each type of products (Fig. 2). In our example, we could create two tables Cars and Houses which will store descriptions of concrete products belonging to these types. Table Products will still store a complete set of products with columns which are common for all types of products such as name and price. Two additional tables Cars and Houses will contain only the products which belong to their type and these tables have columns specific to one concrete type of product. Table Cars will a column Power and table Houses will have a column Area. If there exist two cars and two houses for sale then table Products will have four records and tables Cars and Houses will have each two records.

Let us now study this schema more thoroughly. First of all we see that for each record in table Types there exist one real table. Thus there is one-to-one correspondence between records and tables. If we create a new type then we need to create one record and one new table, and we delete a type we need to delete a record and a table (as well as products with this type).

Another observation is that records can represent tables and play a role of tables. In other words, records can be considered containers for other records just like tables. For example, if we take one product type as a record from Types then obviously records from table Cars belong to this record or they are included into this record.

On the other hand tables may behave like normal records. In particular, tables may have references and properties. For example, although table Cars is considered a table we have its counter-part stored in table Types as one record. That record has an identifier and it has some characteristics. And the properties of this type record characterize table Cars. In other words, we can say that table Cars has some delivery type (which is stored in the corresponding type record in table Types).

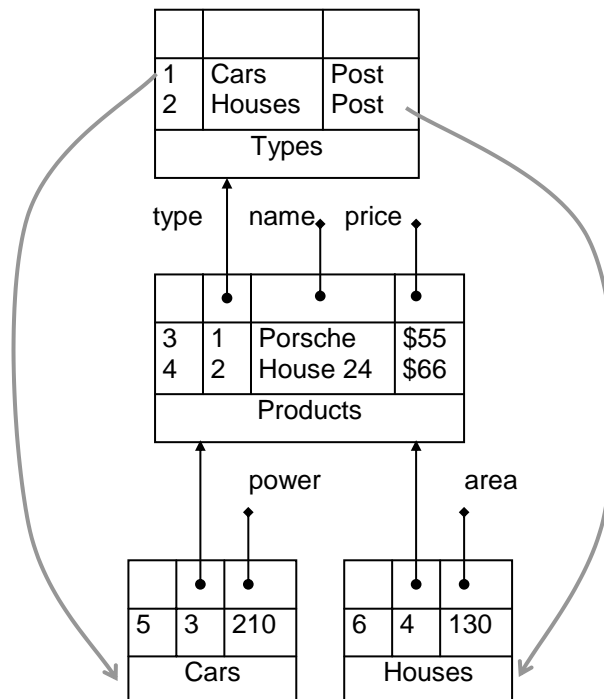


Fig. 2. For each type record one table is created which stores only products of this type.

In this new schema, in contrast to the previous variant, type of product is represented by both a type record and a type table. The most important conclusion here is that *one record and one table in pair may actually represent one and the same entity from the problem domain*. Thus it is only a property of our concrete data representation mechanism (by means of tables and records) that is responsible for such a splitting while in reality product type is one entity which however is able to have properties like records and to include other records like tables.

In the previous schema we retained one common table Products for storing common characteristics of all products along with their type. In that case records from table Types were referenced from the corresponding column of table Products. Now let us suppose that we do not want to store common properties in one table but prefer to have them in each individual table created for each type. In this new case table Products is not needed and does not exist. Then product common columns are defined within each individual type table (Fig. 3). In our case table Cars and table Houses will both have columns Name (of products) and Price (of product). This example is provided in order to demonstrate one interesting property: table Type exists and stores important information but its records are not used anywhere and, particularly, the model does not have any information about correspondence between records from table Types and tables Cars and Houses. In other words, we know that records from Types correspond to types tables like Cars and Houses where this mapping is stored? One simple approach consists in defining a column in table Types which stores a table name for the corresponding type table. Then we can select records from table Types and find a table where the products of this type are stored

Such a solution however has one consequence: we do not know in advance tables we will work with, i.e., we do not know table names which will be used in queries. Since we do not know table names we also do not know their columns. Indeed, at some moment we might add yet another product type as a record in table Types and create the corresponding table. But then we need to store somewhere the structure of this table so that it can be retrieved and used for querying it and getting products.

Another solution is to use table names as some entities or their identifiers. Indeed, information about type is already represented by the fact of table existence. If a table exists then this means that there is the corresponding entity with some characteristics. If we assume that table identifiers can be used as reference to the corresponding entities then this can simplify the representation. This solution is used in the concept-oriented model where any entity may exhibit itself as a record and as a table by having one reference.

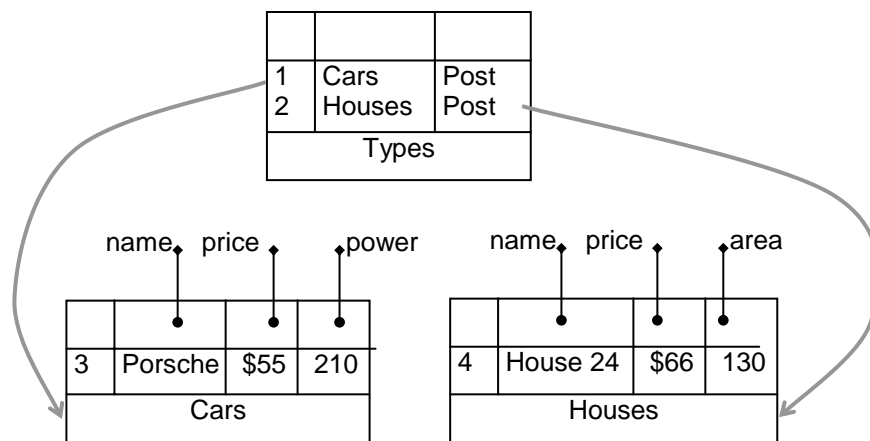


Fig. 3. Each table corresponds to one product type implicitly.

In order to get information about table structure and other properties of tables we can use system catalogue. It is a special system level set of tables where database stores all system information including a set of tables and their columns. An important conclusion is that *the system catalogue is an integral part of our database*. Indeed, we use it to store information about structure of our own tables and this information is then used to build valid queries.

When considering system catalogue we can make another note. It has a table which contains all user tables. This means that when we create tables Cars and Houses then two records are added into this system table. These two records represent the two user tables. Thus our tables always have their counter-part representatives in the form of system records. Here again we confirm our conclusion that *table always represents some entity from the problem domain and, particularly, they have identifiers and their own*

characteristics. However, system stores only characteristics of tables that are needed for functioning the database. The user frequently needs to have his own additional columns describing tables. This is why it is frequently needed to create a table with records representing our own tables like Types in our example. Each record in table Type corresponds to one record in system table of tables.

3. Modelling Classes in OOP

In object-oriented programming we also have a similar problem. Let us assume that we need to model state and behaviour of various figures like squares, triangles and circles. One approach consists in defining one class Figure with fields for all possible types of figures such as radius (of circle), size for square and so on (Fig. 4). In this case we also need one field which stores a type of the object. Then we can instantiate this class and set its field values depending on the type. In such a case the program will consist of a set of objects having one class but distinguished by their type field. The object type in this case still exist but is represented explicitly by values the dedicated field.

```
public class Figure {
    String type; // "Circle", "Square"
    double radius;
    double size;
    double area;
}
```

Fig. 4. All types of figures can be represented by one class and distinguished by type field.

An alternative approach consists in defining several classes of objects each representing one type of figure (Fig. 5). Each such class will have only fields appropriate for its type of figure. The type field in this case is not needed because figure type is represented implicitly by class name. An important conclusion is that now a set of classes represents what in the first example was represented by a set of types as values of type field. Thus we only changed the method of representation of some entities: instead of using normal values for representing figure types we used special class elements for that. In other words, each value of field type in the first example corresponds to one class in the second example.

```
public class Square {
    double size;
    double area;
}
public class Circle {
    double radius;
    double area;
}
```

Fig. 5. All types of figures can be represented by many classes and distinguished by class name.

4. Properties of Representing by Tables

In the previous sections we demonstrated that entities can be represented either by tables or by records. For example, we can define a model where product type is represented by records and we can develop an alternative model where product type is represented by tables. The question is then what are advantages and disadvantages of the two methods and when should we use each of them.

If entities are represented by records only then they are then used to mark other records as their column values. Thus such entities are explicitly visible as record properties and this makes it possible using a powerful mechanism of filtering, querying and aggregating for getting necessary information. In other words, in such an approach where entities are represented by records it is very easy to identify these entities in other records by using properties (columns values). This approach is essentially a manual encoding of object types into their fields. One might say that it is inappropriate use of fields but it is not so. According to the concept-oriented principle any object field or record property specifies its class or group (more about that in the next section). Thus a clear advantage of representing entities as records like in Fig. 1 is that we can apply

powerful built-in data access mechanism to these entities. For example, we might select products of one type by specifying this type in WHERE clause of SQL query.

Another advantage of this approach is that types and membership in classes are actually dynamic and we can easily change an object type if it is necessary. In the case of several tables representing different entities it is not possible to change membership of records. If a record was created in one table then it will exist there forever.

If we were following such an approach then it result in storing all entities in one wide table. This is because as soon as we define table A which references records from table B we at the same moment get the situation where records from B play a role of classes, groups or categories for records from A. Obviously, we do not want to have one wide table even if this approach is extremely convenient from the point of view of information access and processing by means of queries and other facilities? Indeed, we are taught that it is much better to define several tables and distribute information among them. One problem in having one wide table is that such a representation is too inefficient because of a large number of null values which represent inappropriate fields. Another problem is that even if we could store sparse records efficiently it is error prone because the user may set some inappropriate fields to non-null values. For example, a power of house could be 123 instead of null. Thus when we decompose our schema we solve at least two problems: we make it possible to efficiently store our records and we impose constraints, which means that records from some table may have only a limited number of non-null field (those indicated in the table definition).

These two advantages very important indeed. The model becomes not only structured and easy to store but it is much more reliable because of implicit constraints. For example, in schema shown in Fig. 1 we can enter any records while in the equivalent schema shown in Fig. 2 some combinations of values are already physically impossible. An important conclusion is that schema plays a role of constraints.

Unfortunately, one needs to pay for these two advantages by having less convenient mechanisms for information access and processing. For example, some information about our products is stored in different tables and we always need to take this into account by making our queries much more complex. If the schema is static and we now in advance (at compile time) what tables exist then the situation is quite acceptable. We simply write somewhat more complex queries. However, if tables may be created and deleted at run time then queries need to be built from information stored in this very database at run time. For complex information systems such a situation is not an exception and it may well happen that the schema allows for creation and deletion of tables as well as their corresponding records. We also showed that such a method of modelling is not a bad style or wrong pattern – it is a very natural approach where records are used to contain other records and hence need to be represented by tables.

The main problem here that existing data models do not support such a method of modelling where tables are used to represent real entities from the problem domain. In particular, it is difficult to directly define properties for tables or to query tables.

In the table below we summarize advantages and disadvantages of two methods of representation entities (by tables and by rows).

<i>Representing by rows</i>	<i>Representing by tables</i>
– Inefficient storage because of wide rows with many nulls	+ Efficient storage because null values corresponding to columns from other tables are not stored
– No constraints because inappropriate fields with null values may still be assigned to have non-null value	+ Constraints on null fields (non-null values are possible only in fields declared in the table where the row exists)
+ Possibility to apply all normal data access and query mechanisms SELECT * FROM Products WHERE type='Cars' Here constraint is dynamic	– Impossibility to apply available data access and query mechanisms to tables SELECT * FROM Cars Here constraint is static
– Inefficient representation because values from one domain need to be duplicated if used in different columns	+ Efficient representation because values from one domain in different columns are represented by reference and are not duplicated

5. Concept-Oriented Model

In the concept-oriented data model (COM) any element of the model by definition can be viewed as a collection other elements of the model and as combination of other elements of the model:

$$e = \{c_1, c_2, \dots, c_m\} \langle o_1, o_2, \dots, o_n \rangle$$

It is a fundamental theoretical assumption which means that any element has two flavours:

- it is a set other elements as its member, and
- it is an object with other elements as its field values

The collectional part has a hierarchical form where each element belongs to only one parent element with one common root element R . This structure is used to produce references which are used to represent elements. This is why it is called physical inclusion of elements. Each element is created, exists and deleted within one parent elements and there are not means to change its parent during the element life time.

Depending on the physical structure and other mechanisms we select the following main types of the concept-oriented model:

- One-level model which has one root where data items exist.
- Two-level model where the root consists of concepts and concepts consist of items.
- Multi-level data model allows for arbitrary depth of inclusion.

Elements in combination are interpreted as object properties, i.e., an element is a combination of its properties. The dual interpretation is that an element is a *logical collection* consisting of all the elements which include it into their combinational part. In contrast to physical collections which are fixed and intended to implement references, logical collections (dual combinations) are intended to represent data itself. Each element logically can be represented as a combination of its parent elements (above it) and, dually, a (logical) collection of all child elements (below it). In conventional terms this means that an object/record is a combination of its properties but at the same time it is a collection/group of all objects/records where it is used as a property. (A property is a group/category for all objects that have it.)

The two-level concept-oriented model consists of the following constituents:

[Root] One root element R is a physical collection of concepts, $R = \{C_1, C_2, \dots, C_N\}$,

[Syntax] Each concept is (i) a combination of other concepts called *superconcepts* (while this concept is a *subconcept*), $C = \langle C_1, C_2, \dots, C_n \rangle \in R$, and (ii) a physical collection of *data items* (or concept instances), $C = \{i_1, i_2, \dots\} \in R$,

[Semantics] Each data item is (i) a combination of other data items called *superitems* (while this item is a *subitem*), $i = \langle i_1, i_2, \dots, i_n \rangle \in C$, and (ii) empty physical collection, $i = \{\}$,

[Special elements] If a concept does not have a superconcept then it is referred to as *primitive* and its superconcept is one common *top concept*; and if a concept does not have a subconcept then it is assumed to be one common *bottom concept*, and an absence of superitem is denoted by one special *null item*.

[Cycles] Cycles in subconcept-superconcept relation and subitem-superitem relation are not allowed,

[Syntactic constraints] Each data item from a concept may combine only items from its superconcepts.

6. Physical and Logical Membership for Two Types of Containers

Data modelling independent of the concrete data model applied is heavily based on the notion of container or set. This means that we normally view our data items as placed and living in some other elements which play a role of groups, sets, categories, containers, tables, etc. Data manipulation then is reduced to adding/deleting items to/from containers and changing item's parent container. For example, if a person belongs to some department then we can place him to a table which represents this department. If an order part belongs to some order then we can specify this order as the order part property.

What is very important for understanding data modelling is that there are two types of sets or containers:

Physical container, which is the only permanent parent element that cannot be changed during life cycle of its child elements, and

Logical container, which is one of many possible parent elements that can be changed for each child element.

In the concept-oriented model this separation means that each element physically belongs to one parent element (Fig. 6a) and logically belongs to several parent elements (Fig. 6b). Such a dual membership is implemented by representing each element as a physical collection of child elements and as a combination of logical parents. For example, a record belongs to only one table and it is physical membership. On the other hand, this very record is a number of fields (a combination of the corresponding elements referenced from these fields). It is very important for the COM that the elements in the record fields are actually logical parents for this record. In other words, we use a principle that

any data item is logically a member of all items referenced by its properties considered groups, categories, sets or containers.

For example, in Fig. 6b element g has three properties represented by elements a , b and c . (All those elements could be in one physical container or in different physical containers somewhere in the physical hierarchy shown in Fig. 6a.) According to the above principle elements a , b and c (superitems) are logical groups or categories for g . Dually, element g is a group or category for elements d , e and f (subitems) because they reference it as one of their properties.

Thus we always have an alternative: either to represent a membership in a group/set/category by means of physical inclusion or a logical inclusion. For example, assume we have a set of departments each consisting of its employees. How to represent those departments? One approach consists in representing each department as a physical container while people will be represented as its physical child elements. For example, two departments will be represented by two tables with records representing its personnel. Alternatively, we can represent a set of all departments by a table while its records will be individual departments. Then any other record can reference one department record as its column value and this will mean that it logically belongs to this department (not only employees but also other types of records such as orders or customers).

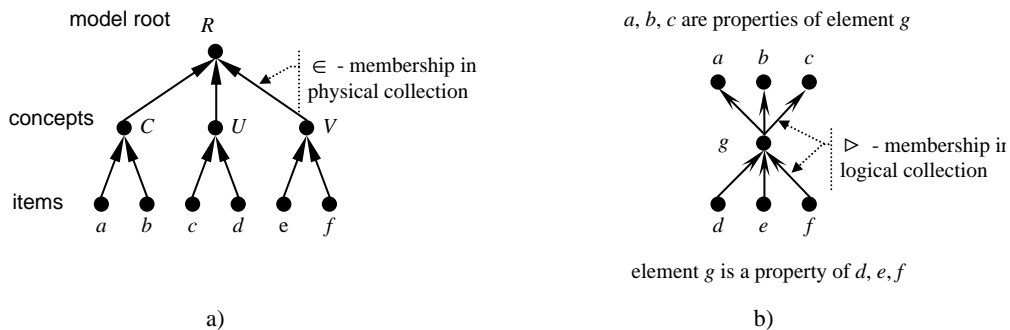


Fig. 6. a) Physical structure b) Logical structure

In section 4 we already described properties of two way of representing entities (by tables and by records). Shortly, the physical approach is more efficient and structured but less flexible. We can create many physical containers and use them to categorize and access data. However, this categorization will be hierarchical and we can change a category (parent container) once an element has been placed into it. Logical container are less efficient, less structured but very flexible. As its extreme we can place all elements into one physical container and then use these elements as logical container for one another. If an element references another element then it specifies its parent logical container. If it references more elements then it has several parent logical containers. In practice we need also to find some compromise between these two extremes.