Informal Introduction into the Concept-Oriented Data Model

Alexandr Savinov
http://conceptoriented.org/

First created: 24.10.2005
Last update: 24.10.2005

1. Principles and Goals

Currently there exist many full featured data models and specific data modelling techniques which are based on different basic principles. For examples, some models are based on first order logic and predicate calculus. Other models are based on set theory. The third class of models proceeds from the separation between entities and relationships and the fourth direction for data modelling emphasizes the dominating role of elementary facts. The existing models can be distinguished also by their main goals. Some of them are targeted at data retrieval by means of declarative queries while others are mainly dealing with procedural aspects. There are models that are adapted for manipulating hierarchies while other models are more suitable for applying to flat problem domains.

Any new data model that is being developed needs to be somehow distinguished in this highly competing world of data modelling mechanisms. In particular, it is important to understand what is so specific in the proposed concept-oriented data model? What are its main principles and what are its goals? Below in the introduction we describe main principles underlying the concept-oriented model while further in the paper we describe its main constituents.

Hierarchical structure. In the concept-oriented model everything is about hierarchy. This means that COM is defined to be hierarchical from the very beginning while other features are derived from its hierarchical properties. This means that *any element of the model has a number of parent elements and a number of child elements*. In particular, if we define a parent for a concept then this parent (superconcept) is the domain for the corresponding dimension. If we define a parent for an item then this superitem is a value of some attribute. Dually, any item is an attribute value for all its child items. It is important that such a relationship has a hierarchical nature. We order available elements of the model in a hierarchical manner and then derive various properties using appropriate interpretations. For example, a common superitem is interpreted as a (logical) collection for its subitems. Thus COM cannot exist without being a hierarchical and it is one of its main distinguishing properties.

Multidimensional structure. In addition to be hierarchical any concept-oriented model is simultaneously multidimensional. Actually we cannot separate these properties and such a synergy is one of its design goals. This means that we assume that in the real world hierarchical and multidimensional properties cannot be separated and we designed the model in such a way that its structure is also hierarchical and multidimensional. Any element of the model can be viewed as living in a multidimensional space where it has some coordinates. We can vary these coordinates in order to characterize it semantically and this is thoughts of as moving the object in the multidimensional space. However, an amazing feature of the concept-oriented model is that it allows us to develop hierarchical coordinate systems for our objects. In such a coordinate system an object itself can be a coordinate for other objects while its coordinates may have their own coordinates. The task of data modelling in this case is reduced to designing an appropriate multidimensional hierarchical space where objects from the problem domain will live.

Objects and attribute values. An important feature of COM is that *it does not distinguish between objects and their characteristics* (attribute values). Data item in COM play the role of objects and attribute values depending on their interpretation in different situations. In other words, the model itself does not know if some data item is an attribute value or an object. We follow a principle that *a superitem is an attribute value for its subitems*. And dually, *subitems are characterized object for this item regarded as an attribute value*. Thus an item is an attribute value for its subitems and it is an object with attribute values represented by its superitems. For example, an order item is an attribute value for its order part subitems, that is, each order part

item is characterized by one order superitem. Here one order is one attribute value however each order item may have its own characteristics like customer. In the paper we use a convention that attribute values are drawn above objects they characterize. Such a diagram is a directed acyclic graph where all items are ordered.

Item positioning and global semantics. How can we define the meaning of a model if it consists of a number of ordered items? We assume that this order is precisely what determines the model semantics. In other words, since any item in isolation does not have its own meaning we determine it from its relative position among other items. The meaning of the model is determined by the whole structure of elements. A consequence of this principle is that *an item has its semantics distributed all over the model*. Indeed, in order to get more information about an item we need to retrieve some other items (superitems or subitems). Those items also have their properties defined in other items and hence we need to retrieve more items and so on. The items we retrieve given this item depend on what part of the global semantics we need. In the simplest case it is only one superitem (an attribute value) while in more complex cases we might need to retrieve quite distant items. In the most existing model an item meaning is determined by its attribute values which are special elements. In the concept-oriented model a meaning of any element is determined by its relative position with respect to other elements. In other words, *a position of an element in the hierarchical multidimensional space determines its meaning in the model*. Thus in order to change the semantics of the model we need simply to move its elements which are meaningless taken by themselves in isolation.

Primitive elements. If all items are meaningless taken in isolation then how can we get the final semantics? We assume that there exist elements in the model that can be interpreted by themselves without the need to retrieve other elements. Such elements are assumed to be positioned at the top of the hierarchy. In particular, direct superconcepts of the top concept are referred to as primitive concepts. Their items are said to be primitive items which have their own meaning used to interpret other items in the model.

Automated logical navigation. The problem of physical navigation is currently solved however the problem of logical navigation is still highly actual. In other words, in most existing data models we are unbound from the physical environment and hence do not need to specify a physical location of elements we want to get. However, we still need to specify a logical path to our elements which is rather complex for most contemporary practical problems. One of the goals of the concept-oriented model is automating logical navigation so that data is accessed much more easily. For example, instead of specifying numerous join conditions among a dozen of tables we can simply provide an access path as a number of dimension names. The concept-oriented database system then does all the rest itself. Such automation is possible only under certain general assumptions made in the concept-oriented model such as the hierarchical multidimensional structure and globality of its semantics. An amazing property of this model is that in many cases we do not need to specify even an access path because the system can reconstruct it itself. In this case the only thing we need to provide is constraints on the source data items and the target items we want to retrieve. For example, we can say that we want to get all orders related with some customer and some date interval. And that is all! No joins, no paths!

Advanced issues. In this paper we provide very simple introduction adapted to the conventional perception of what data model and data modelling is. The concept-oriented model is however much more ambitious. In particular, in this paper we do not touch the issue of physical organization of data model and layered (physical) structure. The problem is that in the conventional data models it is considered a big achievement that a data modeller can deal with only logical structure. That is definitely true, however, in the concept-oriented model combines these two flavours: the logical organization and the physical organization. This means that the data modeller can work with any separate level however he can also describe a binding between layers and the layers themselves. The result is that the model is hierarchical not only at the logical level but also at the physical level. In this paper for simplicity we do not describe any physical features.

Summary:

- All elements of the concept-oriented model are ordered and have a number of children and a number of parents and the model is a multidimensional hierarchical space.

- Objects and values are not distinguished. Instead a parent item is interpreted as an attribute value for its child items
- Item meaning is determined by its relative position among other items. In order to get its semantics it is necessary to retrieve other appropriate items.
- There exist elements that have their own meaning called primitive.

2. Multidimensional Hierarchal Data Schema

2.1. Multidimensional Space

In the concept-oriented data model hierarchical ordering of elements plays a primary role while other properties are derived from this order. In order to illustrate the principle of hierarchical ordering let us consider the conventional star schema where we select a master table linked to different detail tables. Master table is normally drawn in the centre surrounded by detail tables (Fig. 1). If detail tables have their own detail tables, then we get more general snow flake schema but here again a master table is surrounded by its detail tables.

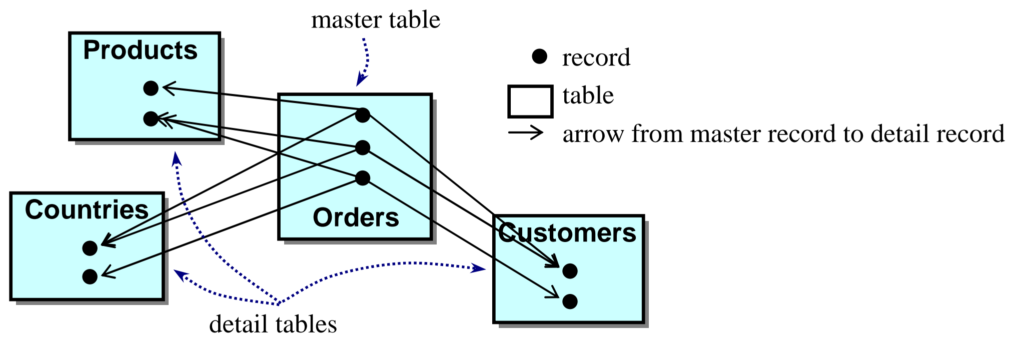

Fig. 1. Star schema in the conventional data modelling.

In the concept-oriented data model we can easily model such a structure however we visualize it by placing a master table below its detail tables (Fig. 2). Thus if we introduce a new detail table it has to be positioned above its master table and if we introduce a new master table it has to be positioned below its detail tables.

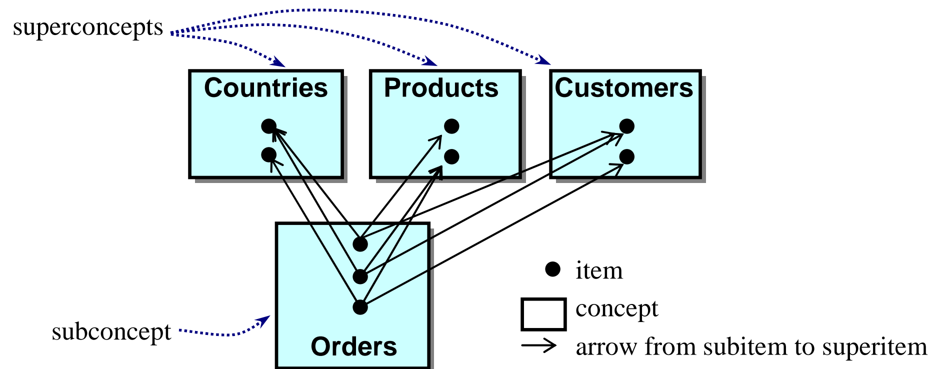

Fig. 2. Star schema in the concept-oriented data model.

This difference may seem unimportant and shallow because it does not influence the result. Indeed, in both cases the schema is one and the same and we change only the ordering of tables. For these simple cases of star and snow flake schemas it is really so and changing the ordering does not change the result model. However, in more complex cases it is not so and the purpose of this example consists in illustrating the change of the paradigm in data modelling. In particular, we want to emphasize that it is

of primary importance for the model to position its elements vertically because the relative position of any element determines its role in the model. In Fig. 1 detail tables can be positioned anywhere in the diagram because we already assigned their role. In contrast, in Fig. 2 we do not assign the role of a table as a master or detail. Instead we fix the mutual order where tables are positioned above and below other tables, and this order then determines their role as a master or detail table. An amazing property of the concept-oriented model is that the order of elements determines also many other mechanisms and features. In other words, proceeding from only one basic mechanism of order we can explain and implement many other conventional and new mechanisms that are needed for data modelling. For example, we can implement data grouping and aggregation, data querying and navigation, one-valued and multi-valued attributes, inference and many others.

In the concept-oriented data model we use the term concept to refer to collections of data items (instead of tables, relations, classes and other analogous terms). A concept that is positioned above another concept with direct or indirect link is referred to as a *superconcept*. Accordingly, a concept that is positioned below some other concept with direct or indirect link is referred to as *subconcept*. In other words, subconcepts are positioned below superconcepts. For example, in Fig. 2 xxx is a superconcept for xxx and xxx is a subconcept for xxx. Concepts that are not linked and hence are not ordered are referred to as orthogonal (or syntactically independent) concepts. Orthogonal concepts are analogous to axes of a coordinate system.

A snow flake schema can be developed either in a bottom-up or a top-down manner. In the former case we start from some subconcept which is analogous to master table and then proceed upward by adding its superconcepts. Since the new superconcepts are positioned above the play the role of detail tables. The second top-down development strategy starts from a set of superconcepts and then proceeds downward by adding new subconcepts for available concepts. Each new subconcept is positioned below its superconcepts and hence plays the role of master table.

There could be mixed strategies but in any case the result is a inversed tree with the root at the bottom and leafs at the top. If there is only one level in this diagram then we obviously get a well known star schema. It is analogous to a flat coordinate system where the detail superconcepts are axes while their common master subconcept is the multidimensional space. Adding data items into detail superconcepts means adding new coordinates that can be then used from the subconcepts. The number of superconcepts is the number of dimensions in this space. Adding data items to the master subconcept means adding new points to the space and defining their coordinates along each of the superconcept axes.

In more general case each of the detail superconcepts may have its own superconcepts for which it play the role of (relative) master table. In this case each coordinate axis is a multidimensional space and each coordinate data item may have its own coordinates. The number of top-level superconcepts is the number of dimensions in this space.

It is important to understand that using such a hierarchy of axes we can effectively restrict possible coordinates of data items from the master subconcept. Theoretically each data item from the master subconcept has its final coordinates in the top-level superconcepts. However, because we introduce intermediate concepts these items may take only intermediate coordinates. In particular, even if the number of combinations of all top-level superitems is large, the number of all possible subitems from the master subconcept can be relatively small if intermediate concepts have a small number of items. This mechanism is referred to as *syntactic constraints* because by introducing intermediate concepts we can prohibit some points in the primitive universe of discourse. In other words, the structure of the model can be used as a mechanism of imposing constraints on possible combinations of data items. In contrast, semantic constraints are described explicitly as some predicate that has to be evaluated to true for any new state of the model.

Summary:

- One common subconcept is an analogue of detail table while its superconcepts are analogous to detail tables.
- One common subconcept is analogous to a multidimensional space while its superconcepts are coordinates axes with items as coordinates.
- Concepts and items at higher levels are more general while concepts and items at lower levels are more specific.

- In the concept-oriented model elements exist in the hierarchy from the very beginning and mutual positioning of elements determines other their properties.
- Subitems are instances of a relation connecting superitems.

2.2. Modelling a Hierarchy

In the previous section we described a concept structure with one bottom subconcept and many superconcepts. A dual structure described in this section consists of one top superconcept and many subconcepts. Such a concept tree with the root at the top is also a wide spread pattern in data modelling. The top concept represents some general notion, class or category while lower-level subconcepts represent more specific notions, classes or categories. Such type of structure is normally used to model hierarchies not only in data modelling but also for other purposes such as class inheritance hierarchy in object-oriented programming. One example of a hierarchy is a product categorization schema (Fig. 3). Top concept `Products` contains items with general properties that are common for all products. Each its subconcept contains more specific items, for example, `Cars` and `Houses` are subconcepts of concept `Products`. Concept `Cars` may have its own more specific subconcepts like `SUV` or `Trucks`. The leaves of the concept tree represent the most specific product categories.

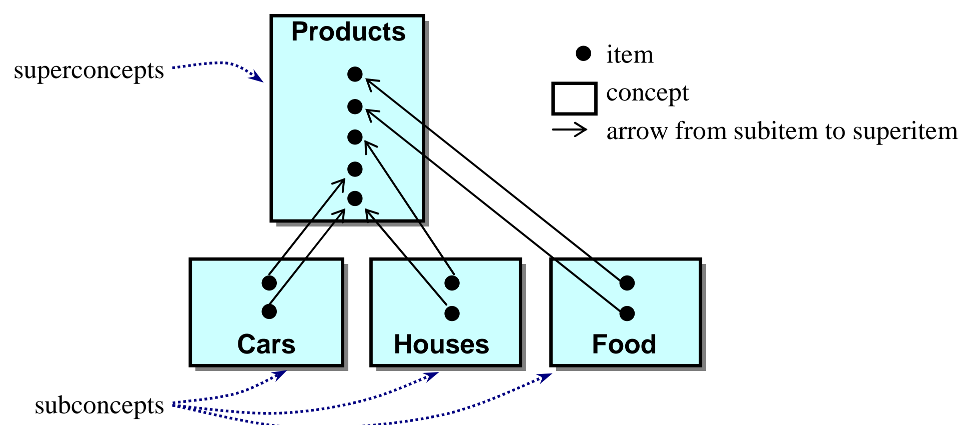

Fig. 3. Product hierarchy.

In this example items represent concrete products and normally for base items from a superconcept are not shared among subitems. In this case items in subconcepts are thought of as extensions of base items in superconcepts. However, in general case superitems can be referenced by many subitems from the same or/and different subconcepts. This means that each superitem may have many subitems from one or more subconcepts.

Concept hierarchy has many different interpretations in addition to categorization. For example, it can be viewed as a means for describing alternative structure or alternative constitution of superconcepts. For example, Fig. 4 shows a structure of company. At the highest level of generality the company is represented by one superconcept `Company`. This concept has a number of properties (dimensions) that are common for all elements of the company data items (this set can be empty). Normally we need more specific information represented in the model which is represented as different subconcepts. Each such subconcept is an alternative specific view on the company and its composition. For example, we can view a company as a collection of its employees and a collection of its products. In our example we can add a concept `Employees` and a concept `Products`. Thus adding new subconcepts can be interpreted as adding *alternative* perspectives to the model. These perspectives can be then used to present and analyze data from different point of view, for example, for preparing reports for human resources department and for sales department.

Using alternative representations we can interpret one and the same superitem from different points of view by choosing one or another subitem (that references it). For example, concept `Customers` might have subconcepts `Orders` and `Surveys`. This allows us to view each individual customer as an entity that orders products or as an entity that answers some questions.

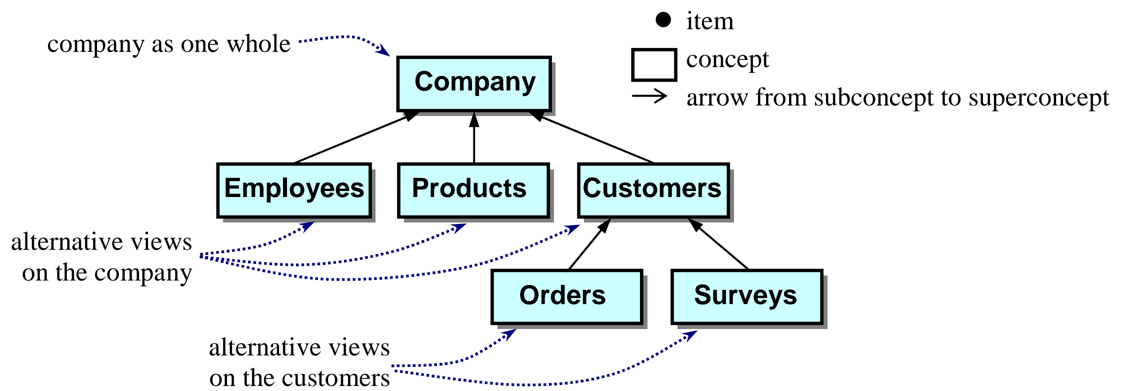

Fig. 4. Company hierarchy.

Summary:

- One superconcept and one superitem can be used (referenced) by many subconcepts and many subitems.
- Subconcepts represent different alternative views of their common superconcept.

2.3. Concept Graph

Multidimensional space (star and snow flake schemas) and hierarchy (tree-like structure) are only particular cases or specific patterns that are frequently used in data modelling. In general case these concrete structures are not used in isolation but rather are parts of a larger data model. In other words, in general case we are not able to select these structures as independent units because they all are interconnected. This means that a concept may have more than one superconcept as in star schema and more than one subconcept as in hierarchical categorization schema.

One of the main ideas of the concept-oriented data model is that *these two structures -- multidimensional space (such as snow flake schema) and hierarchical tree-like structure -- can be combined in one structure called concept graph (Fig. 5).* It is a directed acyclic graph which combines multidimensional and hierarchical properties -- precisely what we want to model. Thus concept graph is the main (syntactic) mechanism of the concept-oriented data model.

For convenience we also add one *top concept* which is direct or indirect superconcept for all other concepts and one *bottom concept* which is direct or indirect subconcept for all other concepts. In other words, if a concept does not have a superconcept defined explicitly then its role is played by the top concept. If a concept does not have an explicitly defined subconcept then its role is played by the bottom concept. Direct subconcepts of the top concept are referred to as *primitive concepts*. Top concept is the most general and bottom concept is the most specific in the model.

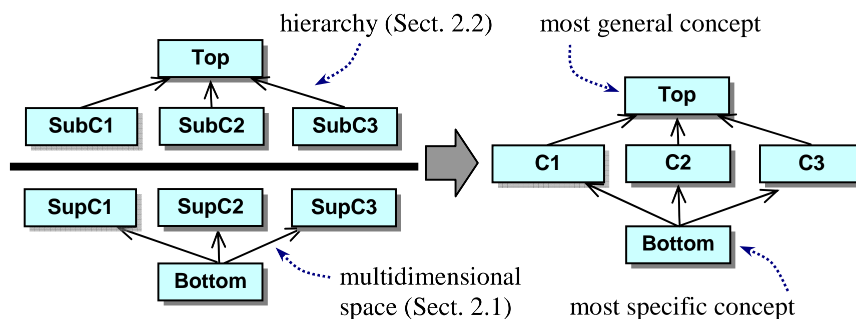

Fig. 5. Concept graph combines hierarchy and multidimensional space.

For example, let us consider a model shown in Fig. 6. At the highest level of abstraction it is represented by the top concept. If we ignore all other concepts then the problem domain is represented by a single element. The next level is where we define primitive concepts like `Prices`, `Users`, `Dates` and `Categories`. It is assumed that auction bids are characterized by price and date, users can create new auctions and participate in them by making bids, and products offered in auctions are assigned some category. Then we define three concepts for `Products`, `Auctions` and `AuctionBids`. Note that concepts in a concept graph may have many parents and many children. Parents (superconcepts) define characteristics while children (subconcepts) contain objects that are characterized by items from this concept. For example, an auction bid item is characterized by one price, one user, one date and one auction. On the other hand, user items are used to characterize auction bid items and auction items. `AuctionBids` is the bottom concept in this concept graph and it contains the most specific items in this model (there is nothing more specific than auction bids because they do not have subitems).

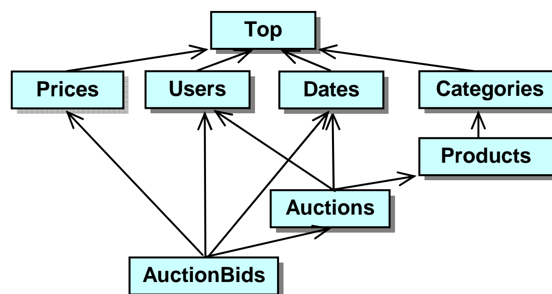

Fig. 6. Auction schema.

Concept graph can be developed by using top-down and bottom-up strategies. If we use top-down strategy then we proceed from the top concept by adding primitive concepts. After that each new subconcept combines a number of available superconcepts. Bottom-up strategy starts from the bottom and adds new superconcepts. Below we provide several principles that are important for the concept-oriented schema (concept graph) design:

- Superconcept exists before its subconcepts and is unaware of its subconcepts. This means that superconcepts represent primary and more general elements of the model and in this sense superconcepts are analogous to base classes in OOP. Superconcepts do not know where and how they will be used and they cannot rely on properties of subconcepts because they will be created later.
- Subconcept is created after its superconcepts and knows its superconcepts. This means that subconcept is more specific to all its superconcepts and relies on their properties.
- Subconcept represents a more specific element and has higher dimensionality. This means that subconcept adds new dimensions to its superconcepts.

Summary:

- Two types of structures (multidimensional space and tree) can be combined into a hierarchical multidimensional space which an acyclic graph of concepts.
- Each concept has a number of superconcepts and a number of subconcepts.
- Top and bottom concepts represent the most general and the most specific concepts.

3. Modelling Relationships

3.1. Many-to-One relationships

Schema can be designed formally by defining a new subconcept as a combination of its superconcepts. For example, concept `Auctions` is a combination of three superconcepts `Users`, `Dates` and `Categories` (Fig. 6). This means that each auction item is a combination of one user, one date and one product category.

A wide spread approach to data modelling consists in considering relationships as a primary construct of the model. This direction to data modelling starts from defining entities which are then connected via relationships. In graph terms entities correspond to nodes while relationships are visualized as edges. The concept-oriented model can be used for data modelling in terms of entities and relationships, which however are roles or interpretations of concepts rather than primary constructs. In other words, the model is still defined via concept graph but the position each concept has in this graph determines its role with respect to other concepts. A concept in this case can be viewed as an entity (type) or a relationship (type) depending on its position in the concept graph.

An important property of the concept-oriented data model is that each arrow in the concept graph represents many-to-one relationship. This relationship connects subitems and superitems interpreted as entities so that many subitems can reference one and the same superitem. Dually, for each one superitem there exist a number of subitems. Fig. 7 shows three different schema fragments each consisting of one subconcept and one superconcept. `OrderParts` is a subconcept for `Orders` because many order parts can belong to one concept and, dually, each order is a collection of many order parts. Thus there is a many-to-one relationship between `OrderParts` and `Orders` which is represented as an upward arrow from subconcept to superconcept. For each auction bid item there is only one auction and, dually, each auction can be viewed as a collection of auction bids. Thus `AuctionBids` and `Auctions` are connected by a many-to-one relationships and in the concept-oriented model this fact is represented by a vertical positioning (`Auctions` is positioned above `AuctionBids`). The third fragment establishes a many-to-one relationship between player items from concept `Players` and team items from concept `Teams`. Here again many-to-one relationship corresponds to subconcept-superconcept relationship in the concept-oriented model, i.e., concept `Players` is positioned below concept `Teams` because each player references only one team it belongs to.

These examples seem almost trivial so the question arises what is original and interesting in them? The main idea here is that we again show that it is all about order and relative position of elements of the model. In other words, we cannot position elements (concepts) arbitrarily, because their relative position determines their role. The approach is that we define relative position for each concept in the concept graph and this position then determines all other properties. In this sense concepts from the three fragments have to have the shown order while in other approaches their order is not important. In other words, if concept `A` is positioned below concept `B` then a many-to-one relationship is automatically established between their items, which means that many items from the first (lower) concept `A` may reference one item from the second (upper) concept `B`.

One consequence of this interpretation of subconcept-superconcept relation is that the conventional entity-relationship data schema can be converted into the concept-oriented data schema by ordering its elements. Such an order subsumes that (primitive) many-to-one relationships are represented by *upward* arrows from the subconcept to the superconcept.

Another consequence is that the concept-oriented schema can be implemented via foreign keys. In this case each foreign key again corresponds to one *upward* arrow in the concept graph. In contrast to the conventional approaches the structure of foreign keys cannot be arbitrary because all foreign keys must be directed up to a superconcept. It is not only a significant constraint for the model schema. It also plays an important role from the point of view of the model interpretation. As we already mentioned an order of concept is a primary mechanism while all other properties are derived from it.

The third important consequence of such a vertical positioning of two concepts is that it is used as a basis for the mechanism of multi-valued properties. In the concept-oriented model multi-valued properties are not a dedicated mechanism. Rather it is again a consequence of the order. We assume that an arrow from a subconcept to a superconcept represents one-valued property (of the subconcept). This means that items from a subconcept are characterized by single values from a superconcept (or null). This very arrow considered in the opposite direction (for a superconcept to a subconcept) represents a multi-valued property (of the superconcept). In this case items from this superconcept are characterized by collections of values taken from the subconcept. Thus all one-valued properties are upward directed while many-valued properties are downward directed. In contrast, in the conventional data models multiplicity of properties is a kind of special mechanism so that we can take a property and then specify if it should be one-valued or multi-valued. In the concept-oriented model multiplicity of properties is derived from the structure of the concept graph. For example, each order in Fig. 7 is characterized by a *collection* of order parts because `OrderParts` is below `Orders`. Analogously,

concept `Auctions` has a multi-valued property that takes its values from its subconcept `AuctionBids` and each team is characterized by a set of its players.

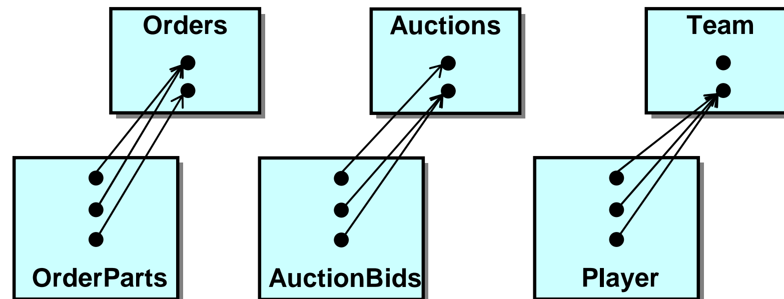

Fig. 7. An arrow represents many-to-one relationship, foreign key or one-valued property.

Summary:

- A link from a subconcept to a superconcept represents one many-to-one relationship while a line from a superconcept to a subconcept represents a one-to-many relationship.
- An upward arrow in the concept graph corresponds to on foreign key between tables.
- An order of concepts defines multiplicity of properties.

3.2. Many-to-Many Relationships

In the previous section we described a primitive many-to-one relationship between concepts. Such relationships are primitive because their instances are not entities (with identity and properties). They are implemented via references and the mechanism of access.

However, in data modelling we normally need many-to-many relationships which connect more than two entities and may have properties. In the concept-oriented data model it is important that arbitrary relationships between entities are implemented by means of their common subentities. This means that if there are some entities described by the corresponding concepts then a relationship among them can be implemented via their common subconcept. Instances of this common subconcept connect instances of the superconcepts. For example (Fig. 8), concept `OrderParts` implements a many-to-many relationship between `Products` and `Orders`. This means that a product item can be used in many orders, and vice versa, one order can include many product items. `AuctionBids` is a common subconcept for superconcepts `Users` and `Auctions`. It also implements a many-to-many relationships where a user can make bids in many auctions and one auction can contain bids from many users.

Note again that for the concept-oriented model it is important that concepts implementing relationships are positioned below concepts corresponding to entities. For example, `OrderParts` is under `Products` and `Orders` while `AuctionBids` is under `Users` and `Auctions`. This order determines the mutual role of these concepts as entities and relationships. In other words, the only information we have in the model about concepts is their order and then this order is used to interpret them as entities or relationships.

Interestingly, relationship concepts can be used as entities in other relationships. For example, if concept `OrderParts` or `AuctionBids` have a subconcept then they will play a role of entities while this subconcept will be interpreted as a relationship. Thus relationships can have a hierarchical form because a subconcept can connect superconcepts of higher order. For example, `Auctions` establishes a relationship between `Users` and `Categories` where the latter superconcept has rank 2 (Fig. 6).

In addition to hierarchy relationships can connect more than two superconcepts. For example, concept `AuctionBids` connects 4 superconcepts.

It is important also to understand that there can be various different interpretations of one concept-oriented schema in terms of entities and relationships. Each such interpretation is an assignment of roles to concepts in the concept graph. One concept may play several roles simultaneously. It can be

interpreted as different entities and different relationships. For example, concept `AuctionBids` can be viewed as a collection of entities with their identifiers and properties. It can be also interpreted as a relationship between `Auctions` and `Users`. Alternatively, it can be interpreted as a relationship between `Prices`, `Dates` and `Categories`.

It should be also noted that subconcept establish semantic dependencies between their superconcepts which are syntactically independent. In other words, if we have a number of concepts and know that their items are not independent then it is necessary introduce their common subconcept with items representing this dependence. For example, concepts `Users`, `Dates` and `Products` are independent without concept `Auctions`. If we want to make them dependent then we define subconcept `Auctions` which is also a relationship between these three superconcepts.

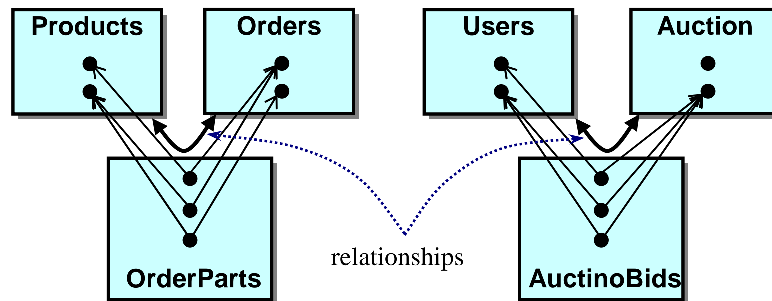

Fig. 8. A common subconcept establishes a many-to-many relationship among its superconcepts.

An interesting question is how primitive relationships can be converted into complex relationships. For example, let us assume that membership of players in a team is modelled via a subconcept-superconcept relation, i.e., by drawing an arrow from subconcept `Players` to superconcept `Teams` (Fig. 9 left). Each player item will reference one team it belongs to.

Suppose now that player may belong to more than one team. In this case this relationship is already many-to-many and hence it cannot be represented by a primitive relationship (an arrow from subconcept to superconcept). In order to describe this new relationship we convert the existing primitive relationship into a new common subconcept `PlayersTeams` (Fig. 9 middle). This new subconcept implements our relationship by indirectly connecting two superconcepts `Players` and `Teams`. If there is an item in concept `PlayersTeams` then it means that some player belongs to some team.

If we want to store a date when a player entered a team then we can add a new superconcept (dimension) to relationship `PlayersTeams` (Fig. 9 right). This subconcept then connects three superconcepts.

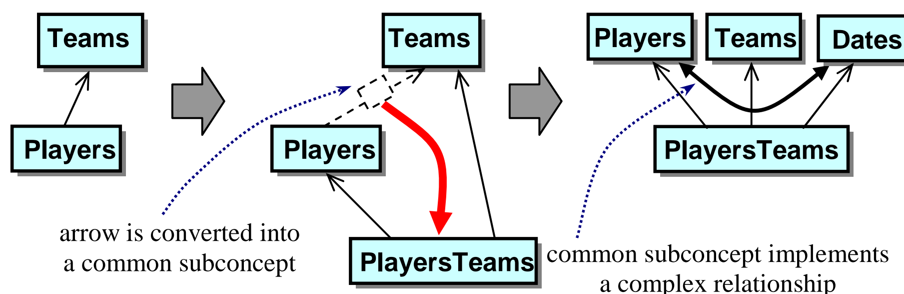

Fig. 9. Converting a many-to-one primitive relationship into a complex many-to-many relationship.

Summary:

- A subconcept is a many-to-many relationship between its superconcepts.

- A concept is interpreted as an entity (type) for its subconcepts and as a relationship for its superconcepts.
- In one and the same model a concept may have different interpretations as entity and relationship.

4. Inverse Dimensions and Access Path

4.1. Model Dimensionality

Dimensionality modelling can viewed as an alternative approach to schema design. This means that theoretically we can define a concept in a concept graph as a combination of its superconcepts, in terms of entities and relationships or as a set of dimensions with their domains. In some cases we would prefer to model the problem domain by using subconcept-superconcept relation while in other cases the use of dimensions is more appropriate. In most cases, especially in complex real world applications, these approaches are mixed.

Each named link in a concept graph from a subconcept to a superconcept is referred to as *dimension* (of rank 1 or local dimension). Superconcept in this link is referred to as *domain* of this dimension. For in concept-oriented schema shown in Fig. 10, concept `Auctions` has three dimensions called `user`, `date` and `product`, with the domains in superconcepts `Users`, `Dates` and `Products`, respectively.

A dimension of higher rank is an upward sequence of local dimensions where each next dimension belongs to the domain of the previous dimension. For example, `auction.product.category` is a dimension of rank 3 of concept `AuctionBids`.

Dimensions play an important role in schema design because they determine a position of superconcept in the definition of the subconcept. Dimensions are also very important for data access and querying because it is the main way to specify what information we want to get from the model.

Another (very general and deep) property of dimensions is that they are dual to references which identify data items. This is a consequence of a general concept-oriented principle that there exist two ways to associate things: collections with references as identifiers and combinations with positions as identifiers. Dimension is a *position* of a superconcept within a subconcept where subconcept is a *combination* of its superconcepts. One property of dimensions is that they are not stored in the model (in storage) and are used explicitly in queries. In contrast, references are not used explicitly but they are precisely what is stored in the model (in storage). The duality between dimensions and references is precisely why tables are so popular. Indeed, table is construct where rows are identified by references and columns are identified by dimensions so we combine these two ways for identifying things in one construct.

Dimension (of any rank) with the domain in a primitive concept is referred to as *primitive dimension*. For example, dimension `auction.date` of concept `AuctionBids` is primitive because it has its domain in `Dates` which is a primitive concept.

The number of paths in the concept graph from the bottom concept to the top concept is referred to as the model dimensionality. (Dimensionality is also referred to as the number of degrees of freedom.) In other words, for each model we can say how many dimensions it has by counting all different paths from the most specific to the most general concept. One consequence of this property is that we know precisely along how many dimensions data items can be positioned. Or, along how many axes we can vary coordinates of our objects. For example, the model in Fig. 10 is 6-dimensional because there are 6 paths from the bottom concept `AuctionBids` to the top (or to primitive concepts).

It is of extreme importance that any item in the model belonging to any concept has its characteristics specified in terms of primitive dimensions. In our auction example any item can varied along 6 primitive dimensions of this model. We say that each item has 6 coordinates. These primitive characteristics or coordinates are determined as follows. We distinguish item's own dimensions and dimensions added by its subconcepts. The item's own dimensions are found obviously by getting its properties or properties of referenced items from superconcepts. Dimensions which are added by subconcepts are always nulls. For example, an auction item has 3 its own dimensions `user`, `date` and `product.category`. All other 3=6-3 dimensions added by subconcept `AuctionBids` (`price`, `user` and `date`) take always value null for all auctions. Informally, we say that items are absent (invisible) along extended dimensions. For example, auctions are absent along price dimension

because this characteristic does not make sense for them (auction items do not have a price). Such an interpretation of null (as absence along dimension) is very important for many mechanisms used in data modelling such grouping and aggregation, logical inference and data transformation.

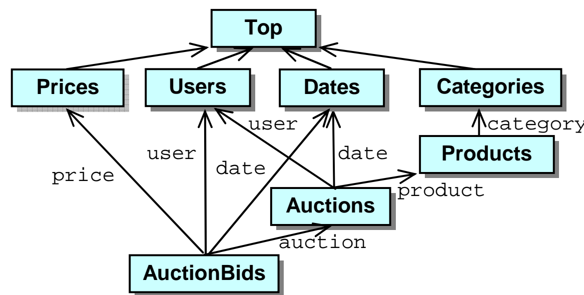

Fig. 10. Dimensions in the auction schema.

Summary:

- Dimension is an upward path in the concept graph.
- The number of paths from the bottom concept to the top is the model primitive dimensionality.
- Each model is characterized by its primitive semantics.

4.2. Inverse Dimensions and Access Path

Inverse dimension is a downward path in the concept graph. In other words, inverse dimension is produced from a normal dimension by inverting its direction. If dimension connects subconcept with some its superconcept then inverse dimension connects superconcept with some its subconcept. Dimension is specified as a sequence of names while inverse dimension is produced by applying an *operator of inversion*. By convention we use curly brackets to invert a dimension. For example, if `AuctionBids.auction.product.category` is a dimension of rank 3 from subconcept `AuctionBids` to superconcept `Categories` then `{AuctionBids.auction.product.category}` is an inverse dimension of rank 3 from superconcept `Categories` to subconcept `AuctionBids`.

Dimensions and inverse dimensions are used to logically navigate over the concept-oriented model (over the concept graph). The idea is that we can specify an *access path* as a sequence of segments where each segment is either dimensions or an inverse dimension. Using dimension we can move up to a superconcept and using inverse dimensions we can move down to a subconcept. One advantage is that we do not need to specify complex numerous join conditions for that.

In order to get related items two operations are used: projection and de-projection. *Projection* of a set of items *I* along dimension *d*, denoted as *I->d*, is a subset of items from the domain (superconcept) of *d* referenced by items from *I*. Each referenced item is included only one time in the projection. For example, a projection of a set of auctions along dimension `user` is a set of all users that at least once created an auction (each user is included only once). If we need to get all referenced superitems then we can use dot operation instead of arrow.

De-projection of a set of items *I* along inverse dimension *{d}*, denoted as *I->{d}*, is a subset of items from the domain (subconcept) of *{d}*, which reference items from *I*. Thus inverse dimension indicate some subconcept and if an item from this subconcept references some source item from *I* then it is included into the de-projection. For example, a de-projection of a set of users along inverse dimension `{Auctions.user}` is a set of auctions created by these users.

Note that we use one and the same sign (arrow ->) for projection and de-projection. In this case the type of the operation is determined by the direction of the path. If the arrow sign is followed by dimension (if we move up) then it is projection. If the arrow sign is followed by inverse dimension (if we move down) then it is de-projection. Access path allows us to find related items by consecutively

applying projection and de-projection operations. Each next operation is applied to the result of the previous operation.

The mechanism of access paths using projection and de-projection is used as a means of logical navigation and data retrieval. In particular, it can be used to retrieve related items given some already known item. For example, given an auction we can get all auction bids using the following simple access path:

```
Auctions.allBids = this->{ AuctionBids.auction }
```

Here we actually defined a derived property of concept `Auctions` called `allBids`. This property can be used as normal dimension however it will return a set of all bids for one auction item. The next access path uses projection after de-projection and returns all users that made bids for an auction:

```
Auctions.allBidUsers = this->{ AuctionBids.auction }->user
```

We can also impose constraints on related items returned by de-projection:

```
Cateogories.todaysAuctions =
  this->{ a : Auction.product.category | a.date==today }
```

This access path will return all today's auctions for a given category item. In order to use constraints (after the bar) we declared an instance variable. More complex derived properties can take parameters:

```
Cateogories.auctionsForDate(Date date) =
  this->{ a : Auction.product.category | a.date==date }
```

It is possible to apply an aggregation function to a collection of items. For example, the next access path returns the maximum bid for all auctions and defines it as a property of concept `Auctions`:

```
Auctions.maxBid = max( this.allBids.price )
```

Note that in this derived property we used a previously defined derived property and then applied a dimension to it in order to get a price of each bid.

Summary:

- Access path is a sequence of segments where each segment is either dimension or inverse dimension.
- Inverse dimension is a downward path in the concept graph.
- Access path automates logical navigation in the model by automating projection and de-projection operations and eliminating the need for manual joins.

5. Query Language

5.1. Source space

In this section we describe one possible approach to query language design. Any query in this approach operates over a elements of the source space of items and selects some of them as its output items. In other words, a query takes some input space and uses its elements to produce an output collection. In general case the source space is a multidimensional space where each item is a combination of other items. In order to describe such a space in the query we simply enumerate a number of source collections by providing their names. This name can be a static collection (a concept in the database) or a dynamic collection (produced from some other query). For each source collection in the list we normally need to specify an instance variable that will store a reference to the current item.

We will use a convention that the source space is distinguished by keyword `FORALL` followed by a list of source collections. Let us consider the following query:

```
NewCollection = FORALL(o Orders, p Products) { RETURN(o, p); }
```

Here the source space consists of all combinations of items from concepts `Orders` and `Products`. If there is 10 orders and 5 products then the source space consists of 50 items. These items are completely new and have their own references.

The composition of new items is specified via keyword `'RETURN'` follows by a list of variables. Thus it is the return statement that determines the structure of the output collection. In the case of no

parameters the output includes all instance variables from the source collections. It is also possible to provide new names for the returned values. In our example each new item has two dimensions which reference the corresponding items from the source collections. The output collection is stored in the variable `NewCollection`. It can be then used just like static concepts in other queries. For example, we might write the following query:

```
AnotherCollection =
  FORALL(nc NewCollection, p Products) { RETURN(nc, p); }
```

Here the input space is again two-dimensional and the body is empty (we can omit the return statement). For the query itself it does not matter if an input collection is a static concept or dynamic collection produced before. It is important however, that input collections are never changed as a result of query processing. If we need to produce some new items then we built new collection and construct our elements from the input items.

Instance variables can be used in the body of the query. They store a reference to the current item. Keyword `this` is used to access the current item from the output collection. In other words, `this` is a reference to the item that is currently being built.

Input collections are frequently specified inline within a query.

```
A = FORALL(p Products, b FORALL(o Orders){...} ) { RETURN(c, b); }
```

Here the second input collection consists of some order items which are selected from the internal query.

Summary:

- Source collections are specified in `FORALL` statement.
- Source collections are computed before this query can be executed.
- Source collections are not changed as a result of query execution.
- Output collection structure is specified via RETURN keyword.

5.2. Selection Criteria

In order to return only a subset of all possible items it is necessary to provide selection criteria via `IF` statement. In `IF` part some condition is specified. If this condition is satisfied for the current item from the source space then the next segment of code (`THEN` part) is applied. For example, the following query returns all combinations of today's orders and customers with the first name 'John':

```
NewCollection = FORALL(o Orders, p Products) {
  IF( o.date = today && p.category == 'Cars' ) { RETURN(o, p); }
}
```

Frequently we need to compute the values returned as an output collection. The intermediate values can be stored as local variables in the body of the query. Let us consider the following query:

```
NewCollection = FORALL(o Orders, p Products) {
  int orderCount = p.getOrderCount; // Derived property
  IF( orderCount > 10 ) {
    RETURN(o, p, p.getOrderSum/orderCount AS mean_order_price);
  }
}
```

Here we retrieve the number of orders for the current product. It is stored in a local variable `orderCount` because it will be used more than once. Then in `IF` part we select only items with products that have more than 10 orders. The output collection will store mean price as the third dimension. Additionally, we provide custom name for this new dimension using `AS` keyword.

Let us consider the following query:

```
FORALL(d Dates, c Categories) {
  IF( isLastWeek(d) )
    RETURN avg(
      this->{ Auctions.date, Auctions.product.category }.maxBid
```

```
        );
}
```

Here we define two-dimensional source space from two concepts `Dates` and `Categories`. For each such pair of one date item and one category item we want to find a set of auctions. In other words, these auctions will be characterized by one concrete date and one concrete category. This operation is referred to as *multidimensional de-projection*. In this case we de-project two-dimensional point consisting of one date and one category to subconcept `Auctions` along two dimensions `Auctions.date` and `Auctions.product.category`, respectively. These dimensions are referred to as *bounding dimensions*. In the above query we write keyword this in order to reference the current point from the input space and then write arrow sign followed by two bounding dimensions in curly brackets. This two-dimensional de-projection along two bounding dimensions is ended by derived property `maxBid` because we want to get a numeric property for our aggregation function.

More verbosely the same query could be written using nested query as follows:

```
FORALL(d Dates, c Categories) {
  IF( isLastWeek(d) )
    RETURN avg(
      FORALL(a Auctions)
        IF( a.date == d && a.product.category == c ) RETURN
a.maxBid;
        );
}
```

Here two-dimensional de-projection is computed manually as a nested query that retrieves items from subconcept `Auctions` that reference both the current date and the current category. For each auction selected for given date and category the nested query returns maximum bid using derived property. The returned set of maximum bid prices is passed as a parameter to the aggregation function.

Summary:

- Selection criteria are specified in IF statement
- Body of the query may use local variable and other statements of the programming language

5.3. Query Optimization

It is important that concept-oriented queries are very close to conventional programming languages and can be directly integrated into a programming language. In other words, we do not write queries as text string that are processed at run-time. Instead, each query is a part of the source code and is compiled by the compiler just like other statements. The compiler then has to understand the logic of queries and optimization rules.

It is easy to see that concept-oriented query is very similar to loops. The difference is that `FORALL` query loop can return a new output collection via `RETURN` keyword. However, it is not necessary and query loop can be used without return statement. Such an approach is normally used for processing elements of source collections. For example, we can update data items.

Although concept-oriented query looks like a procedural query it is actually so. Such queries are mixed and can mix procedural elements and declarative elements. It is precisely the design goal of our approach because in complex queries we normally want to combine advantages of both approaches. In some cases it is enough only to provide selection criteria for the source collections and the compiler then will generate the necessary code for the database engine. In other case it may be important to have some intermediate code including conditional computations or normal loops. In both cases it is the task of the compiler to find appropriate optimizations.

Summary:

- Concept-oriented queries are an integral part of the program code and are compiled by the program compiler.
- It is the task of compiler/interpreter to find appropriate optimizations.
- Concept-oriented queries effectively combine procedural and declarative features.

6. Conclusions

In this paper we described a new concept-oriented data model. This model a number of advantages which allow us to simplify data modelling process and this use of data models. Below we summarize some of these advantages and main properties of the described concept-oriented model.

Hierarchical multidimensional organization. The concept-oriented model dictates the structure of concepts and does not permit to have arbitrary graphs. This is a strong constraint but it is precisely what allows us to derive many useful mechanisms and properties of the model. This structure of concepts assumes that any concept has a number of superconcepts and a number of subconcepts without cycles and complemented with the most general top concept and the most specific bottom concept. The model itself then is multidimensional and hierarchical simultaneously and we do not need to model these properties separately.

Canonical semantics. The concept-oriented model is characterized by canonical semantics. We can precisely say what is the meaning of data in any concrete model and compare different model semantically. For example, we can say that two models with different structure have the same semantics or one of them is more specific than another. Any data item in this case has all dimensions that the model has even if the concept where it exists has only a subset of dimensions. In such an approach any local operation with data can be expressed as a change of the canonical semantics.

Globality of the semantics. This property means that any item has its semantics distributed all over the model. In traditional data models it is assumed normally that one data item is characterized by its attribute values. We also follow this principle except that an attribute value is just another data item, i.e., we do not distinguish between objects and their attribute values. Thus getting attribute values for an item is reduced to retrieving some other items which also have their own attribute values. In this case data items in the model are characterized by all other items in this very model and we say that the item semantics is globally distributed. The item meaning depends on what other related items we want to retrieve. For example, the meaning of order items could be expressed by the related order parts or, alternatively, by retrieving related customer items.

Automating logical navigation. The relational model of data automates physical navigation however it fails to automate logical navigation. This results in the necessity to specify numerous joins in its queries in order to get related records because the data model cannot help us in logical navigation. The concept-oriented model makes it very simple to navigate over its structure of concepts and retrieve related data items by specifying only access path as a sequence dimensions and inverse dimensions. Note that this facility is not simply a convenient query language – it is a consequence of the main model properties such as multidimensional hierarchical structure and global canonical semantics.

Automatic retrieval of related items. An amazing property of the concept-oriented model is that in many cases an access path for retrieving related items can be built automatically. In this case in order to retrieve related items we can only specify constraints imposed on source concepts and then indicate the target concept. The concept-oriented database will be able to build access path and get the target items that are related to the source items under the specified constraints. For example, we could specify some date and some customer and then ask the system to get all related products. No other information is necessary in order for the system to get the related products. This is not possible in any other model (except for maybe in URM) because this mechanism is based on the concept-oriented model main principles such as multidimensional hierarchical organization and global canonical semantics. Of course, in the case of many possible paths it is necessary to provide some hint to the system. For example, products could be related to customers via orders and via surveys and the concept-oriented database system is not able to choose among them. Such a hint could be given by specifying some intermediate concept in the path.

Multi-valued properties via inverse dimensions. Most existing data models have multi-valued properties as a dedicated separate mechanism. This means that we can declare a property either as a one-valued or as a multi-valued without any consequences for the model syntax and semantics. In the concept-oriented model multi-valued properties are dual to one-valued. In particular, multi-valued properties are implemented via inverse dimensions.

Simple query language. The concept-oriented query language combines properties of procedural languages and declarative languages. Simple queries can be expressed in a declarative manner while for more complex queries we can use procedural features. It is the task of compiler how to optimize such queries. Concept-oriented queries are based on the mechanism of access paths and hence make it very easy logical navigation over the model.

Simple grouping and aggregation. Grouping and aggregation is made especially simple because of the hierarchical structure of the model. We proceed from the assumption that any superitem is a group or category for its subitems.

Generalizing various existing data modelling approaches and techniques. The concept-oriented model is compatible with many existing models in the sense that many existing data modelling techniques and mechanisms can be easily implemented or simulated in COM: For example, entity-relationship modelling is made easy because concepts can play both roles. Dimensionality modelling is also easy because the structure of COM is intrinsically multidimensional. Hierarchical modelling is also simple so that such techniques as online analytical processing look very natural when wrapped into the concept-oriented environment.