Computer Science Journal of Moldova, vol.**20**, no.3(60), 2012

References and arrow notation instead of join operation in query languages

Alexandr Savinov

Abstract

We study properties of the join operation in query languages and describe some of its major drawbacks. We provide strong arguments against using joins as a main construct for retrieving related data elements in general purpose query languages and argue for using references instead. Since conventional references are quite restrictive when applied to data modeling and query languages, we propose to use generalized references as they are defined in the concept-oriented model (COM). These references are used by two new operations, called projection and de-projection, which are denoted by right and left arrows and therefore this access method is referred to as arrow notation. We demonstrate advantages of the arrow notation in comparison to joins and argue that it makes queries simpler, more natural, easier to understand, and the whole query writing process more productive and less error-prone.

Keywords: Data modeling, query languages, concept-oriented model, join, reference, arrow notation, data semantics.

1 Introduction

The main goal of a data model is providing suitable structure for representing things and connections between them. Operations for data access and analysis are performed by means of some kind of query language which reflects and relies on these structural principles. For a general purpose data model and query language, the key problem is in finding the simplest and most natural structure and operations which

©2012 by A. Savinov

cover a wide range of patterns of thought and mechanisms being used in data modeling.

Most data models are very similar in how they represent things but they are quite different in representing connections. There exist several major ways for representing connectivity such as relationships, links, references, keys, joins. A relationship is a thing which may have its own properties and identity. Relationships can connect many things but they do not have a direction. A link is a directed binary relationship, that is, a thing that connects two other things with special roles: an origin and a destination. A reference is also a directed connection between two things but in contrast to links it is not a thing and has neither separate identity nor properties. A key is a number of properties of the thing which are used for identification purposes. Join is an operation which relies on thing properties in order to establish a connection between them at the level of queries.

One of the main motivating factors for developing the relational model [1] was the desire to get rid of (physical) identifiers and to focus on the data itself rather than on how it is represented and accessed. However, removing physical identifiers led to removing connectivity from the model. As a consequence, data was broken into several isolated sets of tuples and the question was how to retrieve related (connected) tuples. The solution was extremely simple: tuples containing the same values were supposed to be related. For example, if both an employee record and a department record have an attribute with the value 'HR' then this employee was supposed to be related to this department. The operation which finds and combines such tuples was called *join*.

Although join was introduced as one of the main operations of the relational algebra, now it is used in almost any data model so it can be characterized as a pillar of data modeling. It is one of the most frequently used words in the literature on query languages and can be found in almost any data related context. The main purpose of join consists in connecting data elements which are modeled as existing separately in different relations. It can be viewed as a means of activating implicit relationship at the level of queries. Since joins are not declared

at the level of the model, they provide almost arbitrary control over the data at query time. This property makes it very powerful operation but at the same time rather difficult to use and even dangerous for inexperienced users. In this sense, join is analogous to the goto (jump) operator in programming languages which is also a powerful low level operator providing high freedom in programming but leading to unstructured code and difficult to find errors [2].

Another wide-spread mechanism of connectivity is *reference*. One of the most important properties of references is that they are not part of the represented thing. For example, a class in object-oriented models does not describe references that will be used for representing its instances. References are not stored as part of the object in any of its fields but rather are provided separately. Another important property is that things cannot be accessed without some kind of reference. Indeed, if a property needs to be accessed then it is not possible to use another property for this purpose just because it is not accessible yet. The pattern "accessing properties using properties" obviously contains a cycle and therefore it cannot be directly implemented. Therefore, it is always necessary to have something that exists separately from and is intended to provide access to object properties. This is precisely what references are intended for. The question is only whether they are described explicitly as integral part of the model, provided by the platform as it is done in object-oriented models, or completely removed and replaced by some other mechanism like primary keys as it is done in the relational model. Essentially, the question is whether references are data and hence the model has to provide adequate means for their modeling or references are not data and should be excluded from the model.

References have numerous advantages in comparison to joins. They are extremely easy to understand because they are widely used in everyday life where all things have some unique identifiers. They are also very easy in use. It is enough to know a reference in order to get the contents of the represented thing. There is no need in specifying what and how has to be compared and what criteria have to be satisfied to access the represented thing. For example, given an employee record

we could retrieve its department by using the reference stored in one of the employee properties. The use of join operation means that a database is a set of things with common values. To access data, it is necessary to specify a criterion which has to be satisfied by all elements. For example, to get a publisher we need to specify that both the book and the publisher must have the same value in some property (publisher id). Although references are very natural and simple to use, joins are much more powerful when it is necessary to manipulate sets of elements rather than their individual instances. For this reason, it is not that easy to replace joins by references and this is why join is still dominating in the area of data modeling and query languages although it is quite difficult to use.

This paper is devoted to comparing joins and references. We demonstrate that join operation has some significant drawbacks which make it difficult to use and error-prone in comparison to references. Therefore, we ask the question whether it is possible to eliminate joins from data modeling (or at least diminish their use) by retaining most of the possibilities this operation provides. Obviously, it is a highly non-trivial task and one difficulty is that thinking of data in terms of joins is so deeply penetrated into our minds that it is considered more a dogma than one of the alternatives in data modeling and querying. Another difficulty is that join is a set-oriented operation while references are instance-oriented and this is why references are not so popular in data modeling. As a reference-based solution to the problem of joins, we describe a novel approach to data modeling, called the concept-oriented model (COM) [8, 9, 10], which generalizes references. In particular, it allows for modeling domain-specific references which replace primary keys. What is more important, COM provides two novel operations, called projection and de-projection, which can be viewed as set-oriented analogue of the classical dot notation. These two operations are denoted by left and right arrows and therefore this approach is referred to as arrow notion. We demonstrate how typical tasks can be (easier) implemented using COM references and arrow notation without using joins. The paper has the following layout. Section 2 describes the operation of join, references and arrow notation in COM. Section

3 describes drawbacks of joins and how these problems can be solved by means of COM references. Section 4 makes concluding remarks.

2 Joins and references

2.1 Joins and common value approach

In mathematics, a Cartesian product is an operation which allows us to build a new set out of a number of given sets by producing all possible combinations of their members. Given two sets U and V, the Cartesian product $U \times V$ is defined as the set of all possible 2-tuples: $U \times V = \{\langle u, v\rangle | u \in U \wedge v \in V\}$. Each element of the Cartesian product connects two input elements. Including *all* combinations of the input tuples in the result set means that all these tuples are considered related, that is, every element of one set is associated with every element of the other set.

Normally, not all input tuples are related and therefore a mechanism is needed which would allow us to restrict the Cartesian product by specifying which tuples from two sets should match. This task is performed by join operation the basic idea of which is that only those combinations of tuples are included in the result set which both satisfy some common criterion. In most practical cases, the selection of related tuples is performed by using the equality condition (this join is therefore referred to as equijoin). Tuples in the relational model are composed of values which are accessed by means of attribute names. In this case, related (matching) tuples produced by join must contain equal values in the specified attributes: $U \bowtie_{p=q} V = \{\langle u, v\rangle | u \in U \wedge v \in V \wedge u.p = v.q\}$, where p and q are attributes which have to contain the same values in both tuples.

In order to be matched, two data elements have to contain the same value in some attributes and therefore we will refer to this mechanism as a *common value approach*. Thus records which store common values are considered related in the database. For example, records from two tables `Employees` and `Departments` could be defined as related if they have the same value in the `city` attribute.

Note also that the general idea of the common value approach is also present in formal logic and deductive databases [11]. In predicate calculus, if two predicates have the same free variable then they have to match (to be bound to the same value) in order for the resulting proposition to be true. Since relations can be represented as n-place predicates, join can be written in logical form. For example, given two predicates $Employees(\#e, cname, city)$ and $Departments(\#d, ename, city)$ representing relations `Employees` and `Departments`, respectively, we can find all combinations of free variables where the matching variable *city* takes the same value.

The common value approach has the following properties:

- The relationship defined by join (via common values) does not have a direction. We simply say that two records match because they have the same property. Although some variants of join like left and right outer join have a direction, it cannot be easily semantically interpreted and should be viewed as variations of one operation. In particular, we cannot say that one record is referenced or linked to the other. In this sense, the common value approach is similar to relationships in the entity-relationship model which also do not have a direction.

- It is defined in terms of values and attribute domains, that is, a connection between two relations is specified via some common domain. There is no direct way to define join in terms of other relations. For example, we cannot directly find `Employees` and `Departments` which have the same `address` attribute which represents a record from the `Addresses` table rather than a domain. The reason is that attributes contain values and cannot contain tuples.

2.2 References and dot notation

Reference is one of the corner stones of the object-oriented paradigm where it is assumed that any object has a unique identity which is

used to represent and access it. References have the following main properties:

- References are values which are passed by-copy. It is enough to store this value in order to represent the object and then access it. When a reference is copied, the contents of the object is not copied but can be accessed later by using this reference. References do not have their own references.

- References are not object properties (not included in the object contents) and not part of the object. They exist separately from the objects they represent.

- References hide the details of object identity so that different objects may have different structure of their references which however are not visible when they are accessed.

- References provide transparent access to objects by hiding its internal mechanics which can be quite complex. They create the illusion of instantaneous access.

- References are used along with a very convenient access pattern, called dot notation, where the result of access is considered a reference which can be used for the next access operation.

References make excellent job in the area of programming but they have a rather limited use in data modeling. So what is the problem in introducing references in query languages and combining features of object-oriented and relational approaches? In fact, it is a rather old idea and almost any new query language tries to use references and dot notion to make data manipulations easier. But the fact is that they all fail in eliminating joins which means that not everything can be done by references in the area of data modeling. The primary reason (for the failure of references in data modeling) is that references and dot notation were designed to manipulate instances rather than sets. In other words, programing is an instance-oriented area while data modeling is a set-oriented area. Indeed, only individual objects can

reference each other, not sets. We cannot easily adopt dot notation for manipulating sets. Another reason is that tuples in the relational model do not have identities because any tuple is unique and identifies itself by its own contents. In the next section we describe an approach to data modeling which does not have these drawbacks.

2.3 References in the concept-oriented model

The concept-oriented data model (COM) is a unified general purpose model the main goal of which is to radically simplify data modeling by reducing a large number of existing data modeling methods to a few novel structural principles. One of its principles is that identities and entities are supposed to be equally important. This distinguishes it from most other models which have a strong bias towards modeling entities while identities (references, addresses, surrogates, OIDs) are considered secondary elements which are either modeled by means of entities or provided by the platform.

COM makes identities and entities equally important parts of a data element both being in the focus of data modeling. An element in COM is defined as consisting of two parts, identity and entity, which are also called reference and object, respectively. Identity is passed by-value while entity is passed by-reference. Both constituents have arbitrary domain-specific structure which is modeled by means of a novel construct, called concept (hence the name of the model). Concept is defined as a pair of two classes: one identity class and one entity class. For example, if employees are identified by their passport number and characterized by name then they are described by the following concept:

```
CONCEPT Employees
   IDENTITY
      CHAR(10) passNo
   ENTITY
      CHAR(64) name
```

Note that objects (entities) of this concept will have only one field and these objects will be represented by a reference (identity) also consisting of one field. However, identity part is passed by-value and

stored in variables while entity part is passed by-reference. A concept can be thought of as a conventional class with an additional class for describing the format of references.

COM provides several benefits which are important in the context of this paper:

- COM does not distinguish between sets of values and sets of objects or, in relational terms, between domains and relations. There is only one type construct, concept, which is used for defining both domains and relations. In particular, relation attributes can be both relation typed and value typed.
- Concepts make it possible to describe arbitrary domain-specific references what is not possible in object-oriented models. In this sense, references in COM are similar to primary keys in the relational model. However, the difference is that they are treated and behave like true references while primary keys are treated as integral part of the entity used for identification purposes (more about these difference can be found in [10], Section 2).

COM introduces an operation of projection which is analogous to dot notation but is applied to sets. In the concept-oriented query language (COQL) it is denoted by right arrow and returns a set of elements which are referenced by the elements from the given set. Sets in COQL are enclosed in parentheses and can also include a condition for constraining its elements. For example, all publishers for a set of books can be obtained by projecting this set of books to the set of publishers:

```
(Books | year > '2005')
  -> publisher -> (Publishers)
```

COM also introduces the opposite operation of de-projection which can be viewed as a set-oriented reversed dot notation. It is denoted by left arrow and returns a set of elements referencing the elements from the given set. For example, given a set of publishers we can get all their books:

```
(Publishers | country = 'MD')
  <- publisher <- (Books)
```

Projection and de-projection can be applied to the result set returned by the previous operation and such an approach is referred to as arrow notation. Arrow notation has the following main properties:

- Operations are applied to sets rather than instances
- It uses domain-specific instances as they are defined in concepts rather than only primitive references
- The structure of references is hidden and is not exposed in the query

In the rest of the paper we describe how these two operations are used for querying instead of joins.

3 References for solving join problems

3.1 Connectivity

Perhaps the main use of joins consists in implementing what references are intended for. A database is thought of as a set of objects referencing each other. However, if the database is unaware of references and manipulates only values then these connections have to be expressed by means of joins. For example, if each employee record references its department then a set of departments for all employees in one country is retrieved by means of the following join-based query:

```
SELECT D.name FROM Departments D, Employees E
WHERE D.dept = E.dept AND E.country = 'MD'
```

Here we immediately see one problem: join is a symmetric construct while references are directed. Indeed, if we look at the above query then it is difficult to understand whether departments reference employees or employees reference departments. It is not surprising because joins have quite different purpose but this fact makes them not very appropriate for implementing references. The mechanism of foreign and primary keys can help here but it is optional and is used at the level of schema rather than in queries.

Another problem of joins is that they expose the structure of references by explicitly specifying all the details which actually do not belong to the domain-specific part of the query. Effectively, the low level mechanics of references becomes integral and explicit part of each and every query that involves more than one table. If the structure of connections changes then all queries where it is used have to be updated. Such program logic or query fragments which are scattered throughout the whole source code are referred to as cross-cutting concern. This problem is well known in programming [4] because it makes programs difficult to maintain and error prone. Such functions as logging, transaction management, persistence and security are typical examples of cross-cutting concerns because they are used in the same form across the whole program. The main goal here is to separate these functions or query fragments from the main business logic.

Join operation is a typical example of a cross-cutting concern because many queries solving different domain specific tasks involve the same fragments in the form of join conditions. The reason is that database schemas always follow certain structure of connections and relationships while joins simply materialize them at query time. In the previous example, the schema contains two tables `Departments` and `Employees` which are connected via the join condition `D.dept=E.dept`. Note however that this join is specified along with the second condition for selecting employees of one country only. The problem is that the first condition is a cross-cutting concern because it depends on the schema structure only and will be repeated in the same form in many queries involving these two tables. The second condition reflects business logic and is unique for each query. In a good query language they should be at least separated and, ideally, the join condition should be modularized so that it does not appear in explicit form in each query. This problem can be partially solved by using a dedicated `JOIN` clause for connectivity and `WHERE` clause for domain-specific conditions. However, this use is optional and the join condition will still be repeated for each and every query.

The mechanism of foreign and primary keys could help in hiding the structure of references at the level of schema. Once a foreign key

has been declared, it is then enough to specify its name instead of enumerating all the columns it (and the corresponding primary key) is composed of. However, foreign keys do not solve the problem of joins at the level of queries because we still have to write them as some condition within `WHERE` or `JOIN` clause along with other domain-specific conditions. Another possible solution consists in defining user-defined types (UDT) in the case of complex primary keys and the corresponding foreign keys. Here again, UDTs allow us to simplify join conditions but do not eliminate them completely so that all queries have to specify how two or more tables have to be joined.

In contrast to joins, the logic of conventional references and referencing is completely hidden so that we see only what has to be retrieved and not how it has to be done. Business logic is effectively separated from the mechanism of implementing references. For example, given an employee we can get the department name by using dot notation:

```
emp.dept.name
```

Here we see neither the structure of references nor the conditions used to match the objects. References can be implemented as 64-bit integers, character strings or more complex structures. Matching related objects could be implemented via look up tables or more complex indexes but these details are also not visible in the access statement. The benefit is that if the structure of references and connections between departments and employees changes then this line of code will still work without any modifications because it does not involve any details of how employees, departments and other objects are connected.

The question is then why not to use references instead of translating them into the representation via joins? One problem is that references need identities to be explicitly declared in referenced elements and referencing attributes have to be appropriately typed. Only in this case the reference structure can be hidden. This problem can be solved by adopting the mechanism of primary keys for identification and foreign keys for typing referencing attributes. One difficulty with this solution is that primary keys are not true references (they are identifying attributes [10]) and also they are optional. A more serious problem is that references cannot be applied to sets while joins are inherently

set-oriented. Indeed, if we apply dot notation to sets then what kind of result should be returned by such expressions?

The solution is provided by introducing COM concepts. First, they provide a mechanism for defining domain-specific references which are used instead of primary keys. Once a concept has been defined, it is used as a type of attributes in other concepts by replacing the mechanism of foreign keys. Thus COM references combine features of primary keys (which are not references) and object-oriented (true) references. For example, the structure of departments and employees can be declared as follows:

```
CONCEPT Departments
   IDENTITY // True reference
      INT dept
   ENTITY
      CHAR(64) name
CONCEPT Employees
   IDENTITY // True reference
      INT emp
   ENTITY
      Departments dept
```

Note that the last line does not expose the structure of connection, that is, *how* employees are connected to departments. If the department identity changes then all other attributes referencing it will not be changed.

Concepts not only allow us to remove the structure of references from schema but also remove it from set-oriented queries by using arrow notation. For example, all departments for a set of employees in one country can be retrieved as follows:

```
(Employees | country = 'MD')
   -> dept -> (Departments)
```

This roughly corresponds to the following instance-based query using dot notation:

```
employee.dept
```

References can also be followed in the opposite direction by means

of de-projection operation. For example, all employees of a set of departments located in one country is found as follows:

```
(Departments | country = 'MD')
   <- dept <- (Employees)
```

Operations of projection and de-projection can be applied consecutively and many fragments can be omitted because they can be easily reconstructed from the schema. Thus rather complex queries involving many tables with numerous joins can be written in a very simple and natural form [6]. What is more important, these queries are set-oriented and do not expose the structure of connections.

3.2 Semantics

One problem of joins is that they appear only at the level of queries and the database is unaware of possible and meaningful joins at the level of the model. For that reason join can be characterized as an application-specific operation. Every new application can issue its own query with arbitrary joins. On one hand, it is an advantage because applications are not restricted in the use of data and can do whatever they need. However, if the meaning and consistency of results is important, it is a drawback because arbitrary joins lead to arbitrary results. The database is unaware of what operations are meaningful and therefore cannot restrict applications from producing meaningless results. For instance, the database is not able to prevent an application or user from joining integer department ids with the number of product items which is obviously a meaningless operation. From the performance point of view, it is also a disadvantage because the database engine is not able to optimize its operations for executing predefined joins declared at the level of schema.

From this point of view, joins are somewhat analogous to the goto operator in programming which also ignores the program structure and provides the possibility to organize arbitrary control flow. It was clearly shown that such style of programming without any constraints is harmful [2] because goto not only ignores the semantics behind program structure but also the compiler is not able to restrict programmers

from making errors. The freedom in using joins has the same effect: the database is not able to restrict users and applications from issuing meaningless queries and cannot restrict them from making errors. The mechanism of joins essentially assumes that the meaning of data is described at the level of queries rather than in the model structure. In particular, by looking at queries we can get more information about data semantics then by looking at the schema. One way to overcome this problem is to use foreign keys which can be viewed as a way to declare what is meaningful in the database. Yet, this mechanism has significant limitations when used in queries and should be viewed as a workaround.

Since join is a low level operation, it can be used to implement many different patterns which are difficult to reconstruct from the query. For example, the join condition `WHERE A.id=B.id` (where `A` and `B` are two tables) says almost nothing about the real intention of the query. We do not know whether table `A` references table `B` or maybe it is not about referencing at all. We do not know whether the purpose of this query is to build a multidimensional space for OLAP analysis or to find related records connected via some relationships. And if this operation uses a relationship then is it containment or general-specific? Join is not an operation which can be easily semantically interpreted. Given a join we cannot say what kind of semantic relation it represents and how the joined elements are related. On the other hand, assume that we want to use existing relationships in the model. How should we join the tables in order to represent them in the query? The answer is not clear because the translation procedure is ambiguous and does not cover all possible situations. This problem has been studied in semantic data models [3, 5] but these models focus more on conceptual representation issues and less on query languages. Although many operations can be expressed at conceptual level, joins cannot be removed completely just because the lower logical level of the model is supposed to always exist.

COM allows us to remove the gap between low level join and high level query semantics because it is also a conceptual model with main constructs having some semantics behind them. In particular, references in COM are not simply a means of connectivity but rather a way

to represent semantics. More specifically, references in COM have the following semantic interpretations [9, 10]:

General-specific A referenced element is more general than the referencing (more specific) element. For example, if table `Products` references table `Categories` then products are more specific elements than their categories.

Containment A referenced element is interpreted as a container where the referencing element exists. For example, if an employee record references a department then this employee is supposed to be included in this department as one of its elements.

Relationships An element referencing other elements is interpreted as a relationship between them. For example, if a marriage record references two persons then it is interpreted as a relationship between them.

Multidimensional An element referencing other elements is interpreted as a point while the referenced elements are its coordinates. For example, since sales record references a product item and its price, this sale is considered a point while its characteristics are coordinates along some axes.

According to this interpretations, projection operation applied to a set means getting more general elements, containing elements, dependent elements (connected via this relationship) and coordinates for these elements. And de-projection has the opposite meaning by producing more specific elements, members of a container, relationships and points with these coordinates. As a result, references are used not only for navigating through a graph but rather for semantic navigation. This makes queries much more semantically rich and much easier to write and understand. For example, projecting a set of employees to departments means getting containers for employees because a department is interpreted as a container for a set of employees. At the same time, a department can be treated as a coordinate for employees which are points in a multidimensional space.

3.3 Common value approach

There is one pattern which cannot be modeled by references, namely, the original common value approach directly supported by join operation. This pattern cannot be ignored because in many cases it is precisely what needs to be done. The common value pattern has its own value and the question is how it can be implemented by means of references without joins. For example, if it is necessary to find departments and employees having the same location then it is not clear how it can be done without join operation.

This task can be solved by using product operation which takes two or more collections as input and returns all combinations of their elements as a result collection. In COQL, input collections along with their instance variables are written in parentheses (instance variables are analogous to table aliases in SQL). For example, all combinations of departments and employees are built as follows:

```
(Departments D, Employees E)
```

If we need to return records having some common value then this condition is specified as an additional constraint:

```
(Departments D, Employees E | D.city = E.city)
```

Obviously, it is very similar to how join operation works:

```
SELECT D.*, E.* FROM Departments D, Employees E
WHERE D.city = E.city
```

So the question is why COM is better. The difference is that product in COM is used exclusively to produce combinations of elements. In particular, it is not used for referencing and navigation purposes. Its typical application is in data analysis where it is necessary to produce a multidimensional cube. For that reason, queries in COM much easier to interpret because the purpose of operations is clearer: arrow notation is used for set-based navigation while product is used to build multidimensional space with combinations of records. In other words, COM reflects the real purpose of each operation. Also, product in COM is more general because there is no difference between value domains and relations (see [10], Section 2, for more information). In particular, it is

possible to use any common collection rather than only direct domains of two relations. The following query retrieves all employees who live in the city where their department is located:

```
(Employees E | E.city = E.dept.city)
```

Here we do not use product operation at all although its relational analogue would require joining two tables. The next query finds a set of departments which have at least one employee living in a different city than this department location:

```
(Employees E | E.city != E.dept.city)
  -> dept -> (Departments)
```

Again, here we do not use product operation but still can do what would require joining in SQL.

Since product operation constrained by some common values is a quite frequent pattern, it can be simplified and generalized. Instead of explicitly specifying a condition the combined elements have to satisfy, it is easier to just specify a common greater collection for the input collections. The paths from the input collections to this common collection are then reconstructed automatically from the schema. (In the case of multiple alternative paths, the condition has to be specified explicitly.) For example, the query

```
(Departments, Employees | (Cities) )
```

returns all combinations of departments and employees which have the same city where `Cities` is their common greater collection. Note that `Cities` need not be a direct greater collection and a longer path can lead from the input collections to the `Cities` collection.

An interesting use of product operation restricted by common values consists in implementing inference which is a procedure where constraints can be automatically propagated through the model [7]. For example, assume that we want to relate departments and employees by the city they are located in. The final goal is to impose constraints on departments and then automatically find employees living in these cities (by ignoring departments people work in). Inference is always performed via some common lesser collection. In our example it is defined as a product of employees and departments with the condition

that they have to belong to the same city. Inference consists of two steps: first de-project to the common lesser collection and then project to the target collection:

```
(Departments | name = 'HR')
   <- (Departments D, Employees E | (Cities) )
   -> (Employees)
```

Note how simple and natural this query is. It specifies only collection names and has no indication how they have to be joined. Even if it is necessary to specify connections, they are specified as paths rather than explicit joins. If the schema changes and the collections will be connected differently then in many cases this query will still work.

4 Conclusion

In this paper we have provided a critical analysis of join operation and its use for data querying and retrieving related elements. Although join is an extremely powerful operation which makes it possible to dynamically (at the level of query) relate arbitrary tuples and retrieve quite complex result sets it has several major problems:

- Join is not appropriate for implementing references which is one of its main uses and one of the main data modeling mechanisms. Join exposes the details of reference implementation and is a cross-cutting concern of query languages which cannot be easily modularized.
- Join is not appropriate for representing semantics behind the higher level operation or pattern it implements. From join structure, it is quite difficult to understand what kind of relationship is used in this query. Joins do not reflect their purpose and cannot be unambiguously interpreted from the point of view of business purpose of the query.

Of course, these are not absolute flaws but rather consequences of the low level character of this operation which makes it inappropriate for domain-specific queries in general purpose query languages where

the criteria of simplicity, closeness to the domain concepts, structural and semantic consistency are of primary importance. Therefore, joins not only require high expertise but also can easily result in semantic bugs which are very difficult to find.

Data access via references and dot notation does not have the problems of join – it is more intuitive, much easier to use and more reliable. Yet, this approach is intended for manipulating instances rather than sets and therefore its benefits in the context of query languages are very limited. To overcome these limitations, we proposed to use generalized references and arrow notation as they are defined in the concept-oriented model. This new representation and access method allows us to combine set-orientation of joins with the simplicity and naturalness of references. The use of generalized references and arrow notation instead of join will result in simpler queries, more natural and structured model design, less errors and higher productivity in query writing.

References

[1] E.Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6): 377–387, 1970.

[2] E.W.Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3): 147–148, 1968.

[3] R.Hull, R.King. Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys (CSUR)*, 19(3): 201–260, 1987.

[4] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.Lopes, J.-M.Loingtier, J.Irwin. Aspect-Oriented Programming. *ECOOP'97*, LNCS 1241: 220–242, 1997.

[5] J.Peckham, F.Maryanski. Semantic data models. *ACM Computing Surveys (CSUR)*, 20(3): 153–189, 1988.

[6] A.Savinov. Logical Navigation in the Concept-Oriented Data Model. *Journal of Conceptual Modeling*, Issue 36, 2005.

[7] A.Savinov. Query by Constraint Propagation in the Concept-Oriented Data Model. *Computer Science Journal of Moldova*, 14(2): 219–238, 2006.

[8] A.Savinov. Concept-Oriented Query Language for Data Modeling and Analysis. *Advanced Database Query Systems: Techniques, Applications and Technologies*, L.Yan, Z.Ma (Eds.), IGI Global, 2010, 85–101.

[9] A.Savinov. Concept-Oriented Model: Extending Objects with Identity, Hierarchies and Semantics. *Computer Science Journal of Moldova*, 19(3): 254–287, 2011.

[10] A.Savinov. Concept-Oriented Model: Classes, Hierarchies and References Revisited. *Journal of Emerging Trends in Computing and Information Sciences*, 3(4): 456–470, 2012.

[11] J.D.Ullman, C.Zaniolo. Deductive databases: achievements and future directions. *ACM SIGMOD Record*, 19(4): 75–82. 1990.

Alexandr Savinov,

Received June 28, 2012

SAP Research Dresden,
SAP AG
Chemnitzer Str. 48,
01187 Dresden, Germany
E–mail: *alexandr.savinov@sap.com*
Home page: *http://conceptoriented.org/savinov*