Thomas Kühn, Kay Bierzynski, Sebastian Richly, Uwe Aßmannn

RSQL - a query language for dynamic data types

Erstveröffentlichung in / First published in:

IDEAS '14: 18th International Database Engineering & Applications Symposium. Porto, 07.–09.07.2016. ACM Digital Library, S. 185–194. ISBN 978-1-4503-2627-8.

DOI: https://doi.org/10.1145/2628194.2628246

RSQL - A Query Language for Dynamic Data Types

Tobias Jäkel[*], Thomas Kühn[†], Hannes Voigt[*], Wolfgang Lehner[*]
[*]Database Technology Group
[†]Software Technologie Group
Technische Universität Dresden
01062 Dresden, Germany
{tobias.jaekel, thomas.kuehn3, hannes.voigt, wolfgang.lehner}@tu-dresden.de

ABSTRACT

Database Management Systems (DBMS) are used by software applications, to store, manipulate, and retrieve large sets of data. However, the requirements of current software systems pose various challenges to established DBMS. First, most software systems organize their data by means of objects rather than relations leading to increased maintenance, redundancy, and transformation overhead when persisting objects to relational databases. Second, complex objects are separated into several objects resulting in *Object Schizophrenia* and hard to persist *Distributed State*. Last but not least, current software systems have to cope with increased **complexity** and **changes**. These challenges have lead to a general paradigm shift in the development of software systems. Unfortunately, classical DBMS will become intractable, if they are not adapted to the new requirements imposed by these software systems. As a result, we propose an extension of DBMS with roles to represent complex objects within a relational database and support the flexibility required by current software systems. To achieve this goal, we introduces RSQL, an extension to SQL with the concept of objects playing roles when interacting with other objects. Additionally, we present a formal model for the logical representation of roles in the extended DBMS.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages

Keywords

Dynamic Data Type, Dynamic Tuple, Role Model, Role Persistency, Conceptual Query Language

1. MOTIVATION

Database Managment Systems (DBMS) are used by software applications, to store, manipulate, and retrieve large sets of data. However, the requirements of current software systems pose various challenges to established DBMS. First, as a result of the success of object oriented languages most software systems organize their data by means of objects rather than relations. This difference of representation is well known as *Impedance Mismatch*. It leads to increased maintenance, redundancy, and transformation overhead as a result of the two divergent views on a conceptual unity [17]. While this overhead is negligible for small and stable applications, it becomes intractable for large and especially dynamic applications. Additionally, objects in an object oriented business application have the tendency to become increasingly complex [29]. This is the case, because they participate in various collaborations and serve various purposes. For each collaboration and purpose the object is enriched by new attributes and behavior. Consequently, one way of dealing with such complex objects is to divide them into smaller parts representing the various purposes [4]. However, this solution suffers from two specific problems: *Object Schizophrenia*, because a single entity is split into several smaller objects with their own identity [14] and *Distributed State*, raising the question how to determine the state of a clustered and possibly distributed object. Last but not least, the development of current and future software systems has at least two challenges: **complexity** and **change** [24].

The former, on the one hand, emphasizes the fact, that software systems tend to become increasingly complex up to a point beyond human comprehension. The latter, on the other hand, highlights negative effects of changes to the requirements, the specification, and implementation of a software system. Such changes become an actual threat, if they have to be applied to a critical system at runtime. Fortunately, researchers provided several approaches to tackle these problems. In summary the development of software systems using a classical DBMS will become intractable, if they are not adapted to these new requirements. Nevertheless, database design can benefit from various approaches proposed in literature coping with these challenges, for example the concept of roles. Please note, that this paper does not address the issue of *Role-based Access Control* (RBAC) within the user management of a DBMS. In contrast to RBAC, we see roles as entities in the domain model of an application.

Our Contributions

We propose the extension of DBMS with roles which enables the representation of object oriented domain models with complex objects within a relational database. This extension should support the flexibility required by current software systems. Thus, the representation of complex objects in relational databases will be tractable, transparent, and independent of a specialized *Object-Relational Mapper* (ORM). Additionally, because the DBMS is now aware of the structure of complex objects, it enables the optimization of storage and query operations. To achieve this goal we introduce RSQL, an extension of SQL with roles, together with a formal model for the logical representation of roles. Both, the query language and the formal model will be part of the proposed DBMS extension.

Outline

Hence, this paper is structured as follows. Section 2 gives a brief introduction to the role concept and its use in software systems. Afterwards, we dive into the representation of complex objects as Dynamic Data Types by providing a formal definition in Section 3 and their implementation in the DBMS in Section 4. Based on these definitions, Section 5 introduces the definition of RSQL. The paper is concluded by a discussion of related work (Section 6), future research directions, and a general conclusion (Section 7).

2. ROLES IN SOFTWARE SYSTEMS

As indicated before, current software systems are characterized by increased complexity and changes on demand. One possibility to cope with these recurring problems is to use the *role concept* when modeling and running the system.

The basic rationale behind the role concept is the insight of entities playing multiple roles during their life time. Roles in general attribute capabilities, obligations, relationships, and constraints to their individual player.

From the perspective of conceptual modeling, roles are ontologically classified as *anti-rigid* and *founded* [28]. The first property indicates, that instances of an anti-rigid type can dynamically start and cease to belong to this type without loosing their identity [10]. The second property emphasizes, that roles depend on the presence of other entities, i.e. their player, other roles, an institution [28, 3]. In contrast to roles, the entities able to play roles, denoted naturals, must be rigid and thus carry an identity criterion [28]. Thus, instances of a Natural Type belong to this type (rigidity) and have a unique and immutable identity (identity criterion) as long as they exist [10]. Most importantly, each natural (instance) can start and later stop playing a certain role (instance) gaining all the features of it without loosing its identity. Despite the fact that every natural can play arbitrarily many roles, each role (instance) is played by exactly one natural (instance) and those naturals whose types fulfill a specific role type of a particular role [28]. In summary, the role concept allows for multi dimensional separation of concerns of complex domain models. As a direct consequence, roles can help to cope with the *complexity* of the design of current software systems and are additionally able to separate complex objects into a stable part (the natural) and several dynamic parts (the roles), as suggested by Sutherland in [29].

While the concept of roles is well established in conceptual modeling, e.g. in the *Entity-Relationship Model* (ER) [7], in the *Unified Modelling Language* (UML) [25], only few programming languages support the notion of roles directly, e.g. *powerJava* [2] or *Object Teams/Java* [13]. However, because of the ability of roles to change the behavior of their player, many software systems have implemented the role concept by means of object oriented languages. This has led to the *Role Object Pattern* (ROP) identified by Bäumer et.al. in [4]. The Pattern, shown in Figure 1, captures the role concept representing the Natural Type as the `ComponentCore` class, all the Role Types as subclass of `ComponentRole` class. In this way, a `Component` can be varied at runtime by changing the roles it plays which in turn change the objects behavior. As a result, it is possible to change the behavior of a running system by introducing roles to the object's in the system. This not only permits changing the behavior of the system but also the structure and other system properties.

Unfortunately, the ROP has some major issues. First, it may lead to *Object Schizophrenia* [14], because each instance of `ComponentRole` is a normal object, carrying its own identity and is independent of its player. Second, the state of the component is distributed among various objects, making its persistence heavily dependent on the used ORM, leading to opaque and configuration-dependent data representations in the underlying DBMS [8, Chapter 1]. Finally, this representation would lead to a performance overhead during the retrieval in the DBMS and reconstruction within the ORM. In sum, the combination of the ROP and default or customized ORMs is infeasible [8], because neither default nor customized ORMs allow the database to make informed decisions for optimizations and clustering. Hence, the next sections address the various problems and challenges mentioned previously by extending DBMS with the concept of roles.

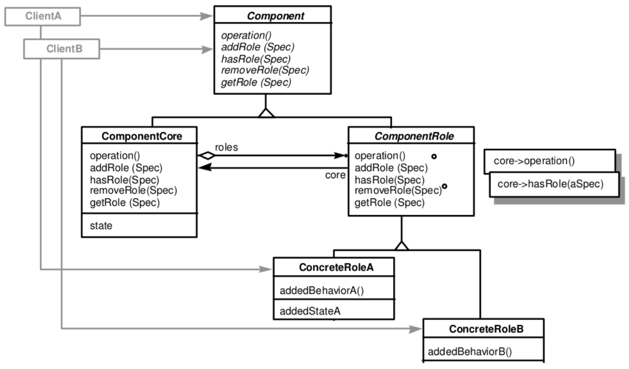

Figure 1: Structure of the *Role Object Patterns*, extracted from [4]

3. DYNAMIC DATA TYPES

Dynamic Data Types (DDT) are a novel data management perspective on role-based data and representing the foundation for enabling role-dynamics in database management systems. To define Dynamic Data Types, we present a formal model consisting of type level and instance level definitions. These definitions are based on the ontological distinction between Natural Types and Role Types presented in [9].

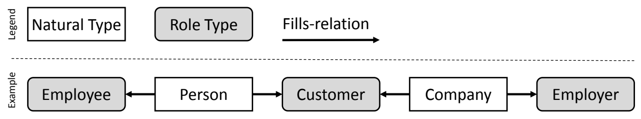

Figure 2: Type Level Example

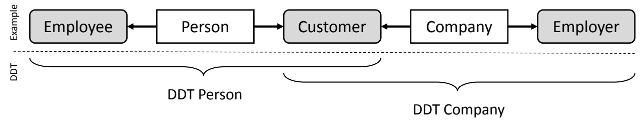

Figure 3: Dynamic Data Type Example

3.1 Type Level

The type level definition is comparable to Friedrich Steimann's type definition for *LODWICK* in [28]. Dynamic Data Types consist of Natural Types and Role Types in general. An ontological distinction between both types can be found in [9]. Roughly, Natural Types are rigid and non-founded. That means an instance of a Natural Type, denoted as Natural, looses its identity by changing the type. Naturals can exist independently of relationships to other individuals in the systems. Role Types are the total opposite to Natural Types, they are non-rigid and founded, which also indicates they are based on relationships [28, 9].

Definition 1. *Let NT be the set of all Natural Types and RT the set of all Role Types. Schema s is then given by the triple $s = (NT, RT, fills)$, where $fills \subseteq NT \times RT$ denotes which Natural Types fulfill which Role Types. Because Role Types cannot exist on their own and to avoid isolated Role Types, we constrain the fills-relation by:*

$$\forall rt \in RT\ \exists nt \in NT\ .\ (nt, rt) \in\ fills$$

Applying this constraint ensures the association of a Role Type to at least one Natural Type by using the fills-relation. Henceforth, we use the infix notation (nt fills rt) for the fills-relation. Figure 2 illustrates an example of the presented definition including the constraint. The example consists of two Natural Types (rectangle shape) *Person* and *Company*. The Role Types (rectangle shape with round edges and shaded gray) in this example are *Employee*, *Customer* and *Employer*. To associate Natural Types and Role Types the fills-relation is populated with *(Person fills Employee)*, *(Person fills Customer)*, *(Company fills Customer)* and *(Company fills Employer)*.

Definition 2. *Let $nt \in NT$ be a Natural Type; a Dynamic Data Type (DDT) is then defined as $ddt = (nt, RT_{nt})$, where $RT_{nt} \subseteq RT$ and is defined as:*

$$RT_{nt} = \{rt \in RT \mid (nt \text{ fills } rt)\}$$

Thus, a Dynamic Data Type is defined as a composition of a Natural Type nt and all Role Types it fills. DDTs are considered as individual types consisting of the subtypes Natural Type and Role Type. Additionally, DDTs are formed indirectly and automatically. In the center of each DDT stands a Natural Type. All Role Types that extend a certain DDT have to be connected regarding to the Natural Type of this DDT in the fills-relation. Figure 3 expands the example shown in Figure 2 with Dynamic Data Types. Two DDTs

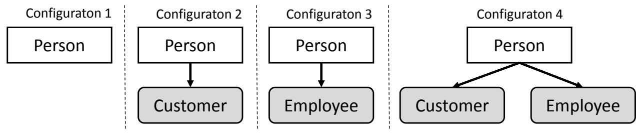

Figure 4: Possible Configurations of DDT Person

are formed, *DDT Person* and *DDT Company*. DDT Person covers the Natural Type *Person* and Role Types *Employee* and *Customer*. The second Dynamic Data Type consists of *Company* as Natural Type and *Customer* and *Employer* as Role Types. As you can see, DDTs are not disjunct, rather they can share Role Types, i.e. Role Types can be shared among several DDTs.

However, the fills-relation indicates that a Natural Type *can play* a certain Role Type. This *can* semantics enables various settings of DDTs, depending on which Role Types are taken into account. These settings are denoted as Configuration. A Configuration specifies by which specific Role Types a Dynamic Data Type is extended.

Definition 3. *Let $ddt = (nt, RT_{nt})$ be a Dynamic Data Type; a Configuration of this DDT is then given by $c = (nt, RT^c)$, where $RT^c \subseteq RT_{nt}$.*

Figure 4 illustrates all possible Configurations of DDT Person taken from the example in Figure 3. As you can see, DDT Person can have four distinct Configurations:

$$\begin{aligned} c_1 &= (Person, \emptyset) \\ c_2 &= (Person, \{Customer\}) \\ c_3 &= (Person, \{Employee\}) \\ c_4 &= (Person, \{Customer, Employee\}) \end{aligned}$$

Since Configurations are defined on the type level, playing multiple Roles of the same Role Type simultaneously does not affect the Configuration.

3.2 Instance Level

On the instance level Natural Types are instanciated to Naturals and Role Types to Roles, respectively.

Definition 4. *Let N be the set of all Naturals, R be the set of all Roles and $plays \subseteq N \times R$ defining which Naturals play which Roles; an instance i of the schema $s = (NT, RT, fills)$ is then defined as $i = (N, R, plays, type)$, where $type : (N \to NT) \cup (R \to RT)$ is a polymorphic function assigning a distinct Natural Type or Role Type to each Natural or Role, respectively. As shorthand notation we define two index sets, the first for Naturals and the second for Roles.*

$$\begin{aligned} N_{nt} &= \{n \in N \mid type(n) = nt\} && \text{for } nt \in NT \\ R_{rt} &= \{r \in R \mid type(r) = rt\} && \text{for } rt \in RT \end{aligned}$$

Additionally, the plays-relation is constrained by the following two axioms:

$$\forall r \in R\ \exists!\ n \in N\ .\ (n, r) \in plays$$
$$\forall (n, r) \in plays\ .\ type(n) = nt \wedge type(r) = rt \wedge (nt \text{ fills } rt)$$

The plays-relation is the instance level equivalent to the fills-relation on the type level. For simplicity, we use the infix notation n *plays* r for this relation. So far, Naturals and Roles exist in the system and they may be related by the plays-relation. Additionally, there are two role specific constraints. First, Roles cannot exist without being related to a Natural and a certain Role can be played by exactly one Natural only. Second, not each Natural is allowed to play Roles of each Role Type. Only Roles of Role Types that are related to a certain Natural Type in the fills-relation can be played. Thus, only relations between Roles and Natural that were also defined on the type level are valid. Both observations are covered by the two constraints defined in Definition 4.

As next step, Dynamic Tuples are defined representing the instance level equivalent of Dynamic Data Types. Each Dynamic Tuple is related to exactly one Configuration of its corresponding DDT. Since Dynamic Data Types describe a set of Configurations, different Dynamic Tuples can be an instance of the same DDT, but differ in their Configuration which implies different schemas.

Definition 5. *Let $s = (NT, RT, fills)$ be a schema and $i = (N, R, plays, type)$ an instance of s. Furthermore, $ddt = (nt, RT_{nt})$ is a Dynamic Data Type and $n \in N_{nt}$ a Natural of this type. A Dynamic Tuple d is then defined with respect to the set of Role Types $RT^d_n \subseteq RT_{nt}$ currently played by n:*

$$RT^d_n = \{rt \in RT \mid n \ plays \ r \wedge type(r) = rt\}$$

Without loss of generality, we assume that $RT^d_n = \{rt_1, \ \dots, \ rt_m\}$ for this set of Role Types.
Case 1 *$RT^d_n = \emptyset$: then $d = (n)$*
Case 2 *$RT^d_n \neq \emptyset$: then the Dynamic Tuple is defined as $d = (n, \ R^d_1, \ \dots, \ R^d_m)$ with*

$$R^d_i = \{r \in R_{rt_i} \mid n \ plays \ r\}$$
$$for \ all \ i \in \{1, \dots, m\} \ and \ rt_i \in RT^d_n$$

Such a Dynamic Tuple d has exactly one Configuration $c^d = (nt, RT^d_n)$.

A Dynamic Tuple d is formed indirectly, likewise DDTs are formed on the type level. It is built around a certain Natural n and a set of Role sets, where each Role set holds Roles of a specific Role Type. Each Dynamic Tuple is of a certain Configuration, namely the Configuration c^d that exactly describes the set of Role Types of all Roles played by this particular Natural n. For instance, a Person p plays the Roles e_1 of Role Type Employee, then $d_1 = (p, e_1)$ is the corresponding Dynamic Tuple in Configuration $c^{d_1} = (Person, \{Employee\})$. The Configuration of a Dynamic Tuple will change, if it starts playing a Role of a Role Type that has not been played or if it stops playing the only Role of the corresponding Role Type. If a Role of an already played Role Type is added, so that this Role Type is played multiple times simultaneously, the Configuration will remain the same, since Configurations are defined on the type level.

3.3 Example

Firstly, we define the set of Natural Types and Role Types, respectively and the fills-relation in accordance to Figure 2.

$$\begin{aligned} NT =& \{Person, Company\} \\ RT =& \{Employee, Customer, Employer\} \\ fills =& \{(Person, Employee), (Person, Customer), \\ & (Company, Customer), (Company, Employer)\} \end{aligned}$$

With these sets we can build the schema s of our system with $s = (NT, RT, \text{fills})$. Secondly, we can derive the Dynamic Data Types for each Natural Type. For our example we can build *DDT Person* and *DDT Company*. As the last type level definition, all Configurations from every Dynamic Data Type are derived. DDT Person has four Configurations depicted in Figure 4 and introduced as Configurations c_1, c_2, c_3 and c_4. Additionally, DDT Company also describes four Configurations:

$$\begin{aligned} C_{co1} =& (Company, \emptyset) \\ C_{co2} =& (Company, \{Employer\}) \\ C_{co3} =& (Company, \{Customer\}) \\ C_{co4} =& (Company, \{Employer, Customer\}) \end{aligned}$$

On the instance level the set for Naturals and Roles as well as the plays-relation with respect to Definition 4 are defined firstly.

$$\begin{aligned} N =& \{Peter, Klaus, Google\} \\ R =& \{E_1, E_2, C_1, C_2, C_3, Emp_1\} \\ plays =& \{(Peter, E_1), (Klaus, E_2), (Klaus, C_1), \\ & (Klaus, C_2), (Google, C_3), (Google, Emp_1)\} \end{aligned}$$

The type function provides the type for each Natural and Role, respectively. According to Definition 4 this function provides the following information:

$$\begin{aligned} type =& \{(Peter \rightarrow Person), & (Klaus \rightarrow Person), \\ & (Google \rightarrow Company), & (E_1 \rightarrow Employee), \\ & (E_2 \rightarrow Employee), & (C_1 \rightarrow Customer), \\ & (C_2 \rightarrow Customer), & (C_3 \rightarrow Customer), \\ & (Emp_1 \rightarrow Employer)\} \end{aligned}$$

The specified schema instance encompasses the following three Dynamic Tuples:

$$\begin{aligned} d_{Peter} =& (Peter, \{E_1\}) \text{ in Configuration} \\ & c^{Peter} = (Person, \{Employee\}) \\ d_{Klaus} =& (Klaus, \{E_2\}, \{C_1, C_2\}) \text{ in Configuration} \\ & c^{Klaus} = (Person, \{Employee, Customer\}) \\ d_{Google} =& (Google, \{C_3\}, \{Emp_1\}) \text{ in Configuration} \\ & c^{Google} = (Company, \{Customer, Employer\}) \end{aligned}$$

4. SYSTEM OVERVIEW

To persist role-based data structures of object oriented software systems in a DBMS without utilizing highly specialized ORM techniques, the DBMS itself has to be adapted. For this purpose, we introduce Dynamic Tuples as persistent relational representation of dynamic objects in software. Hence, the main task of such a DBMS is to manage Dynamic Tuples and to provide efficient access to this data for users. To query for Dynamic Tuples users must provide a valid Configuration to the DBMS while the DBMS filters all qualified Dynamic Tuples regarding to the provided Configuration. Generally, users and applications interact with DBMSs by utilizing certain query languages. We aim for an implemented role-based data model and for this reason we provide a query language tailored to the data model presented in Section 3, named RSQL. In Figure 5 an adapted DBMS is illustrated in conjunction with RSQL's language

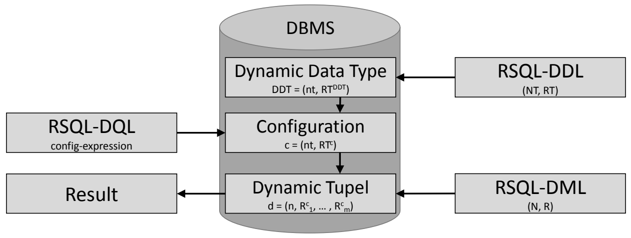

Figure 5: System Overview

components.

While querying, the user has to provide a Configuration to describe the desired Dynamic Tuples. The data query language part of RSQL provides the concept of configuration expressions. Configuration expressions are a novel and powerful tool to specify what the schema of a Dynamic Tuple has to look like to be classified as a qualified tuple. These expressions are not obliged to describe only one Configuration, rather they describe a set of Configrations. This set can be used by the system to filter matching Dynamic Tuples. A set of Configurations as a filter conditions may result in an output that contains Dynamic Tuples of different Configurations. Thus, Dynamic Tuples of different schemata, but of the same type, can be in the same result. During the filtering process the nested Role sets of Dynamic Tuples must be handled relationally. Nested Role sets can be represented in a non-first normal form (NF^2). In [20] *nest* and *unnest* operations has been defined to transform NF^2 into relational tables and in [27] a prototypical DBMS supporting NF^2 relations including a query language extension is explained. By utilizing these approaches, Dynamic Tuples can be defined in a nested fashion and processed in a relational way. After the filtering process, all qualified Dynamic Tuples are returned to the user.

Dynamic Tuples have to be inserted into the system before users can query for them. To manipulate the instances, RSQL provides a tailored data manipulation language. Users must be able to create, extend, and update Dynamic Tuples. This is done indirectly. Dynamic Tuples are manipulated by adding the elements they consist of to the system. Here, users can create, delete, and update Naturals and Roles, respectively. A new Dynamic Tuple is created by inserting a new Natural while an extension is performed by creating a new Role. Extending a Dynamic Tuple results in a Configuration change. If the corresponding Natural starts playing a Role of a Role Type not included in its Configuration the Dynamic Tuple is extended, as well. Otherwise, the Configuration remains the same. In the same way, Contraction of a Dynamic Tuple leads to a Configuration change, whenever the last Role of a certain Role Type is removed from the Dynamic Tuple.

Each Dynamic Tuple is in a certain Configuration which is derived from the corresponding Dynamic Data Type. The elements of Dynamic Data Types are Natural Types and Roles Types. The user indirectly creates Dynamic Data Types by adding Natural Types and Roles to the system. This general procedure is already known from creating and extending Dynamic Tuples. RSQL provides a data definition language, to create Dynamic Data Types. A new Dynamic Data Type is created if a new Natural Type is created in the system. This DDT is extended, if a new Role Type is added

```
⟨select⟩ ::= SELECT ⟨projection-clause⟩
     FROM ⟨from-clause⟩ ( WHERE ⟨where-clause⟩ )?

⟨from-clause⟩ ::= ⟨config-expression⟩
     ( , ⟨config-expression⟩)*

⟨config-expression⟩ ::= ( NATURALTYPE
     ( ⟨ntname⟩ ⟨ntAbbreviation⟩ | _ )
     ( PLAYING ⟨logical-derived-config-expression⟩)? )

⟨logical-derived-config-expression⟩ ::=
     ⟨derived-config-exrpession⟩
   | NOT ⟨logical-derived-config-expression⟩
   | ( ⟨logical-derived-config-expression⟩ ⟨junctor⟩
     ⟨logical-derived-config-expression⟩ )

⟨derived-config-exrpession⟩ ::= ( ROLETYPE ⟨rtname⟩
     ⟨rtAbbreviation⟩ )
```

Figure 6: DQL Statements

to the system and connected to the Natural Type. In contrast, Dynamic Data Types are contracted if the connection to a Role Type is removed or if the Role Type is deleted.

5. RSQL

Users and applications interact with a DBMS by utilizing query languages as interface. By introducing new database objects this interface has to be modified, too. RSQL is a novel interface between DBMSs and users, that manages Dynamic Tuples and Dynamic Data Types to enable users to query for data with role semantics. In addition, RSQL adds role integrity conditions to the DBMS for consistency and to prevent invalid states. Furthermore, it improves interoperability between several applications running the DBMS. RSQL is an acronym for Role SQL, that implies (i) RSQL is designed for role-based data structures, and (ii) is built on SQL [19]. It can be seen as a conceptual query language. The general syntax of RSQL is based on the SQL grammar. RSQL's syntax is defined using the Extended Backus-Naur Form (EBNF) [18]. Note, all non-terminal symbols that are not explained in detail are equal to their SQL equivalent. Like SQL, RSQL consists of a data query language (DQL), a data manipulation language (DML), and a data definition language (DDL). All components are explained in the following subsections, starting by detailing the DQL.

5.1 Data Query Language

The DQL provides a powerful *select* statement to retrieve Dynamic Tuples. Figure 6 shows the syntax of the *select* statement. It starts with the keyword '`SELECT`' followed by the ⟨*projection-clause*⟩, which allows restricting the retrieved attribute set. The projection is followed by the ⟨*from-clause*⟩ and the optional ⟨*where-clause*⟩, which allows filtering Dynamic Tuples by value. Please note that grouping and ordering of tuples is also possible but left out for simplicity of the presentation.

To handle the complexity of Dynamic Data Types, RSQL offers ⟨*config-expressions*⟩ as sophisticated type descriptions. A ⟨*config-expression*⟩ describes a set of Configurations and addresses all Dynamic Tuples that are visible under one of these Configurations. It can be seen as a type level filter condition. A Natural Type is the center element of any ⟨*config-expression*⟩ and is given after the keyword '`NATURALTYPE`'. After the keyword '`PLAYING`' the ⟨*logical-derived-config-expression*⟩ indicates which Roles can be played by the addressed Dynamic Tuples. This logical expression allows connecting several Role Types logically to a predicate match Configurations have to fulfill. A ⟨*logical-derived-config-expression*⟩ may contain a single ⟨*derived-config-expression*⟩, a negated ⟨*logical-derived-config-expression*⟩ or two ⟨*logical-derived-config-expressions*⟩ connected by a logical ⟨*junctor*⟩, e.g., '`AND`'. The *derived-config-expression* describes a Role Type by adding '`ROLETYPE`' and a name including an abbreviation.

A given ⟨*config-expression*⟩ ex is a function $ex(c) \rightarrow \{\top, \bot\}$, which partitions all possible Configurations under the current schema into Configurations c with $ex(c) = \top$ that match the expression and Configurations c with $ex(c) = \bot$ that do not match the expression. It addresses all Dynamic Tuples d having a Configuration c_d with $ex(c_d) = \top$.

For instance, assume the following Dynamic Tuples stored in the database:

$$\begin{aligned} d_{Klaus} &= (Klaus, \{E_2\}, \{C_1, C_2\}) \\ &\quad \text{with } c^{Klaus} = (Person, \{Employee, Customer\}) \\ d_{Peter} &= (Peter, \{E_2\}) \\ &\quad \text{with } c^{Peter} = (Person, \{Employee\}) \end{aligned}$$

Given the expression `(NATURALTYPE Person p PLAYING ((ROLETYPE Employee e) XOR (ROLETYPE Customer cu)))`, $(Person, \{Employee\})$ and $(Person, \{Customer\})$ are matching Configurations, while $(Person, \emptyset)$ and $(Person, \{Employee, Customer\})$ do not match. Accordingly, the expression addresses the Dynamic Tuple d_{Peter} but not d_{Klaus}.

Role Types can be shared among several Dynamic Data Types, hence, RSQL provides a way to query Role Types independently of their Natural Types. In a ⟨*config-expression*⟩ a '`_`' has to be specified instead of naming the corresponding Natural Type. This '`_`' will be interpreted as wildcard for Natural Types. Consequently, only attributes of the specified Role Types are avaliable for projection in the ⟨*<projection-clause>*⟩. The query processing in this case becomes more complex, because the database system has to determine all affected Dynamic Data Types and corresponding Configurations firstly. Afterwards, Dynamic Tuples in the corresponding Configuration will be created for each determined Dynamic Data Type. For instance, two Natural Types can fill the same Role Type, the database system will check all Dynamic Tuples of the addressed Dynamic Data Types for a Configuration match. This results in two sets of qualified Dynamic Tuples, both having the same schema but are of different types. Finally, the different Dynamic Tuples will be united to return only a single result. Exmaple queries are given in Section 5.4.

5.2 Data Manipulation Language

Dynamic Tuples are manipulated indirectly by adding Naturals and Roles to the system. These instances in combination with the plays-relation form Dynamic Tuples. Thus, data manipulation comprises inserting, updating, and deleting Naturals as well as Roles. To manipulate the system instance, the statements listed in Figure 7 are available.

```
⟨insert-nt⟩ ::= INSERT INTO NATURALTYPE ⟨ntname⟩
      ( ⟨attribute-name⟩ ( , ⟨attribute-name⟩ )* )
      VALUES ( ⟨value-expression⟩ ( , ⟨value-expression⟩ )* )

⟨insert-rt⟩ ::= INSERT ROLETYPE ⟨rtname⟩
      ( ⟨attribute-name⟩ ( , ⟨attribute-name⟩ )* )
      VALUES ( ⟨value-expression⟩ ( , ⟨value-expression⟩ )* )
      INTO ⟨config-expression⟩ (WHERE ⟨where-clause⟩)?

⟨update-nt⟩ ::= UPDATE NATURALTYPE ⟨ntname⟩
      SET ⟨assignment-expression⟩ (WHERE ⟨where-clause⟩)?

⟨update-rt⟩ ::= UPDATE ROLETYPE ⟨rtname⟩
      IN ⟨config-expression⟩ SET ⟨assignment-expression⟩
      (WHERE ⟨where-clause⟩)?

⟨delete-nt⟩ ::= DELETE NATURALTYPE
      FROM ( ⟨config-expression⟩ ) ( WHERE ⟨where-clause⟩ )?

⟨delete-rt⟩ ::= DELETE ROLETYPE ⟨rtname⟩ ⟨rtAlias⟩?
      FROM ( ⟨config-expression⟩ ) ( WHERE ⟨where-clause⟩ )?
```

Figure 7: DML Statements

Inserting Naturals into the system results in a new Dynamic Tuple. To create a new Natural the ⟨*insert-nt*⟩ statement has to be executed. The statement starts with '`INSERT INTO NATURALTYPE`' followed by a Natural Type name and a list of attribute names. After the keyword '`VALUES`' a list of ⟨*value-expressions*⟩ provides a value for each of the listed attributes. Inserting a new Role into the system results in an extension of a Dynamic Tuple and may result in a Configuration change. A Configuration change happens, if and only if a Role of a Role Type that is currently not being played is inserted. In this case the Dynamic Tuple's configuration c^d is extended by a new Role Type. The ⟨*insert-rt*⟩ statement starts with '`INSERT ROLETYPE`' followed by a Role Type name. Subsequently, a list of attribute names and a list of *value-expressions* follows. Each Role belongs to exactly one Natural. Accordingly, the ⟨*insert-rt*⟩ statement adds the Role to a single Dynamic Tuple, which is specified after the keyword '`INTO`' with a ⟨*config-expression*⟩ and an optional ⟨*where-clause*⟩. In case the ⟨*config-expression*⟩ and the ⟨*where-clause*⟩ address multiple tuples the insert statement fails.

The update statements allow to update values of a certain type. Naturals are updated with the ⟨*update-nt*⟩ statement. This statement starts with an '`UPDATE NATURALTYPE`', a Natural Type name, and the keyword '`SET`'. The following ⟨*assignment-clause*⟩ defines which attributes have to be updated. Finally, the optional ⟨*where-clause*⟩ restricts the set of Naturals that are affected by the update. Roles can be updated with the ⟨*update-rt*⟩ statement. It starts with '`UPDATE ROLETYPE`' and the name of the Role Type of the Role that should be updated. The following ⟨*config-expression*⟩ specifies the Configuration a Role has to be part of to be affected by the update. This allows, for instance, to update the salary of all employees that also have the customer role. Finally, the ⟨*assignment-expression*⟩ and the ⟨*where-clause*⟩ are similar to the ⟨*update-nt*⟩ statement.

```
⟨create-nt⟩ ::= CREATE NATURALTYPE ⟨ntname⟩
      ( ⟨attribute-definition⟩ ( , ⟨attribute-definition⟩ )* )

⟨create-rt⟩ ::= CREATE ROLETYPE ⟨rtname⟩
      ( ⟨attribute-definition⟩ ( , ⟨attribute-definition⟩ )* )
      ( PLAYED BY ( ⟨ntname⟩ ( , ⟨ntname⟩ )* ) )?

⟨drop-nt⟩ ::= DROP NATURALTYPE ⟨ntname⟩

⟨drop-rt⟩ ::= DROP ROLETYPE ⟨rtname⟩
```

Figure 8: DDL Statements

In contrast to insert statements, updates are set operations.

Delete statements reduce the number of instances in the database. Deleting Naturals removes the corresponding Dynamic Tuples from the database, because all connect Roles cannot exist without their Naturals. Naturals can be deleted with a ⟨*delete-nt*⟩ statement. The statement is indicated by the keywords '`DELETE NATURALTYPE FROM`' and specifies a ⟨*config-expression*⟩ and an optional ⟨*where-clause*⟩. Naming a Natural Type is unnecessary, because the Natural Type will be specified in the ⟨*config-expression*⟩ anyway. Deleting Roles diminishes Dynamic Tuples and may also result in Configuration changes. A Configuration change happens if and only if the last instance of a Role Type is deleted from a Dynamic Tuple. To delete a role, the ⟨*delete-rt*⟩ statement must be used. The statement starts with the keywords '`DELETE ROLETYPE`' followed by a Role Type name and an optional alias. A *config-expression* and a *where-clause* allow to limit the set of Roles to be deleted similar to their use in the ⟨*update-rt*⟩ statement. Like update statements, delete statements are set operations. Every qualified Role and Natural will be deleted from the system. Examples for RSQL's DML statements are given in Section 5.4.

5.3 Data Definition Language

Like Dynamic Tuples on the instance level, Dynamic Data Types (DDT) are created indirectly out of their build blocks. Dynamic Data Types are made of Natural Types and Role Types in combination with the fills-relation. To define a Dynamic Data Type, the statements in Figure 8 are available.

Creating a new Natural Type leads to a new Dynamic Data Type, since each DDT consists of exactly one Natural Type. A ⟨*create-nt*⟩ statement creates a new Natural Type. This statement starts with the keywords '`CREATE NATURALTYPE`'. A unique type name is followed by several ⟨*attribute-definitions*⟩. Each ⟨*attribute-definition*⟩ consists of a unique name and a technical type. Creating a new Role Type extends one or more existing Dynamic Data Types by a new Role Type. Because Role Types have to be connected with a Natural Type, the fills-relation is populated during the creation process. The ⟨*create-rt*⟩ statement starts with '`CREATE ROLETYPE`' and a unique Role Type name. Afterwards, the attributes are specified by ⟨*attribute-definitions*⟩. To populate the fills-relation, Role Types specify their connected Natural Types after the keywords '`PLAYED BY`'. The ⟨*drop-nt*⟩ and ⟨*drop-rt*⟩ statements drop Natural Types or Role Types from the system, respectively. Natural Types are only to be dropped if they are not connected to any Role Types to prevent unconnected Role Types. Examples of this Data Definition Language are presented in the upcoming section.

5.4 RSQL Example Statements

To give a better impression on the RSQL statements, we present and explain some statements in detail. At first, the type layer is created by adding a Natural Type and two Role Types to the system as shown in Figure 9. The Natural Type *Person* has two attributes, a *name* of type *VarChar(100)* and an *age* of type *Int*. Next, a Role type *Employee*, consisting of an *eID* and *salary* both of type *Int*, are added to the system. Additionally, the new Role Type is connected to *Person* and thus, creates the DDT Person. Customer is the second Role Type and consists of a *cID* attribute only. This type is also linked to *Person*. The final type level setup contains a single Dynamic Data Type $Person\ ddt = (Person, \{Employee, Customer\})$.

Next, a Dynamic Tuple is added to the system as presented in Figure 10. The first statement creates a new Natural *Peter* with an age of 37 and thus, a new Dynamic Tuple has been created. The following statements extend the created Dynamic Tuple with three Roles. At first, an *Employee* Role is added to a Person where the Person's name is *Peter*. Here, a Configuration change is performed, because the Role Type *Employee* is currently not being played. Second, another Role of type *Employee* is added. This time, there is no Configuration change, because the Dynamic Tuple *Peter* already plays a Role of type Employee. Finally, a new *Customer* Role extends the Dynamic Tuple *Peter*, since it is the only Dynamic Tuple in the system and the config-expression returns this. The last insert results in a second Configuration change.

Finally, three example queries are explained and listed in Figure 11. Assuming only the statements explained before, the first statement returns the name of all Naturals of type Person that play the Role of type Employee at least once and the salary is higher than 1000. The Dynamic Tuple Peter matches this description and is returned. The second query asks for Persons that are either an Employee or a Customer and the only Dynamic Tuple in the system does not match this description, because both the Employee and Customer Role Type are played. The third example query returns all Persons, that are not a Customer. Here, the Dynamic Tuple *Peter* is not returned because the Role Type Customer is played. Finally, the last example returns all Customer Roles, no matter which Natural Type is playing this Role. Assuming the database also stores a Role Customer that is played by a Natural *Google* of the Natural Type Company, both, the Dynamic Tuple *Peter* and the Dynamic Tuple *Google* will be returned including the Customer's attributes only.

6. RELATED WORK

Object-oriented software systems organize their data by means of objects rather than relations, like relational database management systems do. This results in the well known impedance mismatch and object-oriented database systems try to overcome this mismatch by storing objects as objects instead of relation. Unfortunately, object-oriented databases cause other problems like poor performance. RSQL has not been designed to solve the problem of the impedance mismatch. It rather tackles the problems of Object Schizophrenia [14] and Distributed State by provid-

```
CREATE NATURALTYPE Person (name VarChar(100), age Int);
CREATE ROLETYPE Employee (eId Int, salary Int) PLAYED BY (Person);
CREATE ROLETYPE Customer (cId Int) PLAYED BY (Person);
```

Figure 9: Data Definition Language Examples

```
INSERT INTO NATURALTYPE Person (name, age) VALUES ("'Peter"', 37);
INSERT ROLETYPE Employee (eID, salary) VALUES (1, 2800) INTO (NATURALTYPE Person p)
        WHERE p.name = "'Peter"';
INSERT ROLETYPE Employee (eID, salary) VALUES (2, 1300) INTO (NATURALTYPE Person p)
        WHERE p.name = "'Peter"';
INSERT ROLETYPE Customer (cID) VALUES (1) INTO (NATURALTYPE Person p PLAYING (RT Employee e));
```

Figure 10: Data Manipulation Language Examples

```
SELECT p.name FROM (NATURALTYPE Person p PLAYING (ROLETYPE Employee e)) WHERE e.salary > 1000;
SELECT * FROM (NATURALTYPE Person p PLAYING ((ROLETYPE Employee e) XOR (ROLETYPE Customer cu)));
SELECT p.age FROM (NATURALTYPE Person p PLAYING NOT (ROLETYPE Customer cu));
SELECT * FROM (NATURALTYPE _ PLAYING (ROLETYPE CUSTOMER));
```

Figure 11: Example Queries

ing a consistent perspective of role-based data for both, application and users. Additionally, object-oriented database management systems cannot handle dynamic objects and do not provide role semantics. For these reason, object-oriented databases will not be considered in detail.

The role concept has been proposed the first time in the 1970s in [1] as Role Data Model and as extension of the network model. Their Role Data Model is based on the observations that real world objects interact via roles and this was not covered by state of the art approaches. Roles have been adopted in many research fields, especially conceptual modeling [7, 26, 11] and programming languages [2, 13] to name just a few. The first approach that brings roles into a DBMS has been proposed in DOOR [31]. DOOR is an object-role DBMS that introduces roles at runtime for the first time. It is built on object classes and role classes where role classes can be played by several object classes on the type level. On the instance level objects and roles exist. Their system definition is similar to the system we described in Section 3. Differences exist on the instance level, where we defined a plays-relation, which is not present in *DOOR* [30]. Furthermore, DOOR persistency is based on a file based associative string database, where RSQLs aims to a relational DBMS. Unfortunately, this project is not maintained anymore and DOOR's source code is not available.

RSQL can be seen as a conceptual query language where queries are written designed by the conceptual design instead of the implemented data structures. Applying this approach, both the software and DBMSs get together, because complex types used in software systems can be managed directely by the DBMS. Conceptual query languages have been introduced in the 1990's. In [15, 21] approaches for entity relationship model oriented querying and extended entity relationship model querying can be found. These approaches and RSQL share the same querying paradigm, but differ in their underlying conceptual modeling language. Since entity relationship modeling builds on static types, these approaches are only able to query static semantics rather than dynamic types like RSQL does.

The *Object Role Model* by Halpin [12] is a fact-oriented and attribute free conceptual modeling language. Based on this modeling language, conceptual querying has been introduced [5, 6]. A user queries from a conceptual perspective without any information about the physical database schema, and thus, the distribution of object data over several tables is hidden by the query. RSQL was designed to implement a conceptual and especially a role-based perspective on the data. Users describe Configurations by using config-expressions without any information about the physical schema, since RSQL does not define the physical storage of the system. *ConQuer* [6] and RSQL share the same querying paradigm, but they differ in their underlying role definitions.

Information Networking Model (INM) uses roles in relationships to describe that objects play roles in a certain relationship to other objects [22, 23]. Like RSQL and the underlying formalism, it also supports dynamic and many-faceted object types. Furthermore, complex relationships between objects can be modeled. INM also provides a query language, called *Information Query Language* (IQL) [16]. The IQL utilizes tree expressions, like *XPath*, to hierarchically describe the desired data. Furthermore, they describe a DBMS that persists INM-based data in a key-value-store (Berkeley DB). In contrast, RSQL aims at a relational DBMS to take advantage of the richer role semantics to (i) store role-based data more efficiently and (ii) optimize queries. Key-value-stores cannot take advantage of in-DBMS optimizations, because no information about the stored data is known to the DBMS. Optimizations have to be performed outside of the DBMS which causes overhead in transferring and processing the data.

Numerous implementation techniques for complex types onto tables have been proposed, but all share the same disadvantage in the event of role-based data structures have to be stored. They do not provide support for roles and dynamic types. The semantics of roles will be lost if role-based data are stored using these approaches and has to be restored by each application individually. As mentioned in Section 4 nested relations can be utilized to physically represent sets of Roles of the same Role Type. Additionally, the nest and unnest operators enable data processing in a relational fashion [20]. Nevertheless, nested tables are a variation of static tables and do not provide role semantics, since only the table and nested table semantics are known to the DBMS. In contrast, RSQL provides a consistent perspective of role-based data structures to users, applications, and the database system itself.

7. CONCLUSIONS

This paper provides a novel approach to extend DBMS with roles to incorporate the requirements of current software applications. Thus, as a result of the increased complexity and tendency to change of software applications, we have proposed an extended DBMS able to represent Roles played by Naturals. This not only permits the representation of complex objects but also increases the flexibility of the underlying schema. The latter ultimately enables evolution of applications using our DBMS at runtime.

In detail, we have introduced a novel Dynamic Data Type together with Dynamic Tuples to represent complex objects at the type and at the instance level, respectively. In doing so, our approach directly supports the storage, manipulation, and retrieval of complex objects. Furthermore, because our DBMS extension is based on the presented formal model, we prevent the occurrence of *Object Schizophrenia* as well as the issues associated with *Distributed State*. This is achieved by enforcing the representation of a Natural and its Roles as a Dynamic Tuple with a unique identity. Thus, when querying a complex object only the states of the Naturals and Roles selected by the Configurations contribute to the state of the Dynamic Tuple limiting its size. On top of that, it is possible to validate a given query against the database schema and thus prevent operations with undefined behavior, for instance by forbidding direct joins between Natural and Role Types. Consequently, the proposed DBMS extension can enforce the semantics of the presented formal model. As a result of the direct representation of complex objects with Naturals and Roles, our approach reduces the transformation overhead and increases the interoperability of the data model and ultimately limits the dependence of software applications on specialized ORM.

In sum, we made the following four major contributions to DBMS. First, we provided a formal model for the logical representation of roles in DBMS. Second, Dynamic Data Types and Dynamic Tuples were introduced as a new approach to represent complex, dynamic objects. Third, we have devised an RSQL to store, manipulate, and retrieve these Dynamic Data Objects. Fourth, a DBMS extension was proposed which increases the interoperability of the schema as well as enforces the semantics of the role concept. Altogether, these contributions represent a step towards DBMS suitable for future software applications.

However, there are many more steps to make in pursuit of this goal. One major step is the development of an efficient representation of the formal model within the logical level of the DBMS together with specialized database operators for combining Naturals and Roles to Dynamic Tuples. Afterwards, we can evaluate the actual performance overhead in contrast to standard and specialized ORM. Another step is, to further extend RSQL and the underlying formal model, to capture both the *relational nature* [28] and *contextual dependence* [13] of roles. Among others, these steps will lead to a new kind of DBMS with increased knowledge of the domain model of future software systems.

8. ACKNOWLEDGMENTS

This work is funded by the German Research Foundation (DFG) within the Research Training Group ”Role-based Software Infrastructures for continuous-context-sensitive Systems” (GRK 1907). Special thanks go to Stefan Hinkel, Johannes Fett, and Christian Kabelitz for their help in developing RSQL and to Ulrike Schöbel for improving this paper.

9. REFERENCES

[1] C. W. Bachman and M. Daya. The Role Concept in Data Models. In *Proceedings of the third International Conference on Very Large Data Bases*, pages 464–476. VLDB Endowment, 1977.

[2] M. Baldoni, G. Boella, and L. V. D. Torre. Roles as a coordination construct: Introducing powerJava. In *Procs. of MTCoord '05 workshop at COORDINATION '05*, page 2006. Electronic, 2005.

[3] M. Baldoni, G. Boella, and L. van der Torre. powerJava: Ontologically Founded Roles in Object Oriented Programming Languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1414–1418, New York, NY, USA, 2006. ACM.

[4] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. The Role Object Pattern. Technical report, Washington University Dept. of Computer Science, 1997.

[5] A. Bloesch and T. Halpin. Conquer: A conceptual query language. In *Proceedings of the International Conference on Conceptual Modeling – ER '96*, volume 1157 of *Lecture Notes in Computer Science*, pages 121–133. Springer Berlin Heidelberg, 1996.

[6] A. C. Bloesch and T. A. Halpin. Conceptual Queries using ConQuer-II. In *Proceedings of the International Conference on Conceptual Modeling – ER'97*, pages 113–126. Springer, 1997.

[7] P. P.-S. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, Mar. 1976.

[8] S. Götz, S. Richly, and U. Aßmann. Role-based object-relational co-evolution. In *Proceedings of 8th*

Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2011), 2011.

[9] N. Guarino. Concepts, Attributes and Arbitrary Relations: Some Linguistic and Ontological Criteria for Structuring Knowledge Bases. *Data & Knowledge Engineering*, 8(3):249 – 261, 1992.

[10] G. Guizzardi and G. Wagner. Conceptual Simulation Modeling with onto-UML. In *Proceedings of the Winter Simulation Conference*, WSC '12, pages 5:1–5:15. Winter Simulation Conference, 2012.

[11] T. Halpin. ORM/NIAM Object-Role Modeling. In *Handbook on Architectures of Information Systems*, International Handbooks on Information Systems, pages 81–101. Springer Berlin Heidelberg, 1998.

[12] T. A. Halpin. ORM 2. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 676–687. Springer, 2005.

[13] S. Herrmann. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[14] S. Herrmann. Demystifying Object Schizophrenia. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '10, pages 2:1–2:5, New York, NY, USA, 2010. ACM.

[15] U. Hohenstein and G. Engels. SQL/EER — Syntax and Semantics of an Entity-relationship-based Query Language. *Information Systems*, 17(3):209–242, May 1992.

[16] J. Hu, Q. Fu, and M. Liu. Query Processing in INM Database System. In L. Chen, C. Tang, J. Yang, and Y. Gao, editors, *Web-Age Information Management*, volume 6184 of *Lecture Notes in Computer Science*, pages 525–536. Springer Berlin Heidelberg, 2010.

[17] C. Ireland, D. Bowers, M. Newton, and K. Waugh. A Classification of Object-relational Impedance Mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09*, pages 36–43. IEEE, 2009.

[18] ISO/IEC. ISO/IEC 14977:1996(E), 1996.

[19] ISO/IEC. ISO/IEC 9075-2:2011 (Information technology – Database languages – SQL), 2011.

[20] G. Jaeschke and H.-J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *PODS'82, Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California*, pages 124–138. ACM, 1982.

[21] M. Lawley and R. Topor. A Query Language for EER Schemas. In *Proceedings of the 5th Australian Database Conference*. Global Publication Service, 1994.

[22] M. Liu and J. Hu. Information Networking Model. In *International Conference on Conceptual Modeling – ER 2009*, volume 5829 of *Lecture Notes in Computer Science*, pages 131–144. Springer Berlin Heidelberg, 2009.

[23] M. Liu and J. Hu. Modeling Complex Relationships. In *Database and Expert Systems Applications*, volume 5690 of *Lecture Notes in Computer Science*, pages 719–726. Springer Berlin Heidelberg, 2009.

[24] S. Murer, C. Worms, and F. J. Furrer. Managed Evolution. *Informatik-Spektrum*, 31(6):537–547, 2008.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual (Paperback)*. The Addison-Wesley object technology series. ADDISON WESLEY Publishing Company Incorporated, 2010.

[26] J. Rumbaugh, R. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1st edition, Jan. 1999.

[27] H.-J. Schek and P. Pistor. Data Structures for an Integrated Data Base Management and Information Retrieval System. In *Proceedings of the 8th International Conference on Very Large Data Bases*, pages 197–207, San Francisco, CA, USA, 1982. Morgan Kaufmann Publishers Inc.

[28] F. Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data & Knowledge Engineering*, 35(1):83 – 106, 1999.

[29] J. Sutherland. Business Objects in Corporate Information Systems. *ACM Computing Surveys (CSUR)*, 27(2):274–276, 1995.

[30] R. Wong, H. Chau, and F. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 402–411, Apr 1997.

[31] R. Wong, H. Chau, and F. Lochovsky. Dynamic Knowledge Representation in DOOR. In *Proceedings of Knowledge and Data Engineering Exchange Workshop, 1997*, pages 89–96, Nov 1997.