Relationships for Dynamic Data Types in RSQL

Tobias Jäkel[*], Thomas Kühn[+], Stefan Hinkel[*], Hannes Voigt[*] and Wolfgang Lehner[*]

[*] Database Technology Group
{tobias.jaekel, stefan.hinkel, hannes.voigt, wolfgang.lehner}@tu-dresden.de
[+] Software Technology Group
thomas.kuehn3@tu-dresden.de
Technische Universität Dresden
Nöthnitzer Str. 46
01187 Dresden

Abstract: Currently, there is a mismatch between the conceptual model of an information system and its implementation in a database management system (DBMS). Most of the conceptual modeling languages relate their conceptual entities with relationships, but relational database management systems solely rely on the notion of relations to model both, entities and relationships. To make things worse, real world objects are not static as assumed in such modeling languages, but change over time. Thus, modeling languages were enriched to model those scenarios, as well. However, mapping these models onto relational databases requires the use of object-relational mapping engines, which in turn hide the semantics of the conceptual model from the DBMS. Consequently, traditional relational database systems cannot directly ensure specific consistency constraints and thus lose their meaning as single point of truth for highly distributed information systems. To overcome these issues we have proposed RSQL, a data model and query language introducing role-based data structures in DBMSs. Despite the fact that RSQL is able to handle complex objects, it does not support relationships between those objects. Therefore, this work adds relationships to RSQL by augmenting the data model and extending its query language. As a result, this extension allows for the direct representation of conceptual models with complex objects and relationships in the DBMS. Thus, relationships can be directly addressed in queries and the DBMS automatically ensures relationship consistency constraints as well as cardinality. In sum, a DBMS equipped with the extended RSQL is apt for storing and querying conceptual models and thus regains its rightful position as the single point of truth for highly distributed information systems.

1 Introduction

Currently, there is a mismatch between the conceptual model of an information system and its representation in a *Database Management System* (DBMS). For instance, if the conceptual model is mapped to a relational database schema, this results in different views on concepts specified on the conceptual level. Most of the widely used conceptual modeling languages are based on objects and relationships to relate these objects, like ER [Che76] or UML [RJB10]. However, relational databases only rely on the notions of relations to model both objects and relationships. To make things worse, real world objects and their relationships can change over time and usually do. Hence, conceptual modeling languages were enriched by features, like the role concept, to allow for the specification of more complex and dynamic systems. Based on this observation, Bachman has proposed the role concept in 1977 [BD77], where the core of an object is separated from its relationship dependent parts, namely roles. This enables information system designers to model dynamics of objects on the conceptual level by representing all relationship dependent attributes and behavior in several roles. Hence, role-based modeling reduces the complexity and allows for continuous evolution of highly distributed information systems. For instance, imagine a person who can be a consultant and a customer. In object-oriented modeling one would define subclasses using inheritance for each combination, which leads to an exponential explosion of subclasses and to reinstantiation each time the person changes the subclass at run-time[Ste99]. Contrariwise, using role-based modeling would only describe the core object and its extension that can be added and removed dynamically during run-time. Thus, subclass explosion and run-time reinstantiation are avoided. However, the semantics of the role concept is completely unknown to relational DBMSs, thus, the gap between conceptual models and their representation in the database becomes even bigger.

This becomes a problem for highly distributed information systems that rely on a DBMS as **single point of truth**. Because data objects are shared among several subsystems, only the DBMS persists data and ensures their consistency for all applications and users. Hence, the DBMS has to provide global consistency, whereas the subsystems have their local perspective. However, current DBMS are not apt for this task, because they lack the notion of complex objects and relationships. Thus, most of the semantics introduced in the conceptual model is lost during the persisting and transformation process. Consequently, most of the semantics and its validation is implemented in mapping engines running in the various subsystems. Nonetheless, they cannot ensure the consistency of the whole distributed information system. Hence, relational DBMSs must be extended to represent roles and relationships directly to regain their characteristics as **single point of truth**.

One step in this direction was the introduction of RSQL in [JKVL14], which extends a classical DBMS to represent dynamic, complex data structures with roles. This definition comprises dynamic data structures and a query language, but does not consider relationships as first-class citizens. Unfortunately, traditional relational methods are insufficient for directly representing relationships. Data in relational DBMS are stored based on relational semantics and thus relationships must either be mixed into relations of other concepts or mapped to a separate relation. In the former case, the relationship attributes and references become intermingled with relations representing concepts. Whereas in the latter case, the separate relation cannot be distinguished from concept relations within the DBMS. Moreover, both approaches lead to fragile schemas, since every cardinality change on the conceptual level causes extensive database schema changes. In sum, we extend RSQL by first-class relationships that preserve the semantics of the conceptual model.

Hence, this paper has the following contributions. Firstly, we present a formal definition for relationships as an extension to Dynamic Data Types. Secondly, we extend RSQL's *Data Definition Language* (DLL), *Data Manipulation Language* (DML), and *Data Query Language* (DQL) to directly represent relationships. Hence, we introduce new database objects on the type as well as on the instance level. Furthermore, we argue that the direct representation of relationships makes the DBMS more robust against cardinality changes. These contributions ensure a direct representation of role-based objects and relationships in relational DBMS. Finally, DBMS regain the ability to be the single point of truth for highly distributed role-based information systems by providing and ensuring consistent persistence of dynamic complex data objects and their relationships.

This paper is structured as follows: In the following section a running example is introduced. Afterwards, Section 3 presents an extended formal data model. This is followed by the RSQL extension in Section 4, its classification and related work in Section 5. Finally, conclusions and future work are discussed in Section 6.

2 Running Example

The most important feature of the role concept for modeling is to express dynamics in typecasting of objects. Basically, this concept distinguishes between *Natural Types* and *Role Types*. Natural Types represent the core and immutable type of an object that cannot be discarded without ceasing to exist. In contrast, Role Types represent the mutable type of an object, which can be acquired and abandoned dynamically. Over time Naturals, the instances of Natural Types, may start or stop playing several Roles. Thus, they can acquire additional attributes dynamically. To constrain which Natural Types *can play* which Role Types they are restricted by a fills-relation.

A Natural Type and its connected Role Types form a *Dynamic Data Type* (DDT) that is handled as individual type by the DBMS. Thus, DDTs are the main data structure in a role-based DBMS. However, Role Types are not exclusive to a single DDT, rather they can be shared between several Dynamic Data Types. Furthermore, a DDT restricts the types of roles that can be assigned to an instance of that DDT. This definition enables DBMS to guarantee role-specific consistency conditions. Additionally, DDTs can only be related by Relationship Types between Role Types. This enables relationships independently of the DDT, since Role Types can be part of several DDTs.

Figure 1 illustrates a small banking application, that serves as running example. It describes *Customers* that are related to *CheckingsAccounts*. Between **Accounts** money can be transferred and a *Customer* can have one or more *Consultants*. The example consists of three DDTs: **Person**, **Company** and **Account**. Both, **Person** and **Company** can be a *Customer*, but only **Person** is allowed to be a *Consultant* as well. *Consultant* and *Customer* are related by a Relationship Type *advices*. The third DDT describes different facets of **Accounts**. An **Account** can be a *CheckingsAccount* and in this role it has to be related to exactly one *Customer*. Additionally, **Accounts** can play the roles of the Role Type *Source* and *Target* to describe transactions of transferring money between **Accounts**. This transaction is represented by the *transfer* Relationship Type. Moreover, a unique *Target* counterpart for each *Source* has to exist.

One possible instance of this model is shown in Figure 2. It comprises two **Persons** *Peter* and *Klaus*, as well as a **Company** *Google*, whereas *Peter* plays the role of a *Consultant*

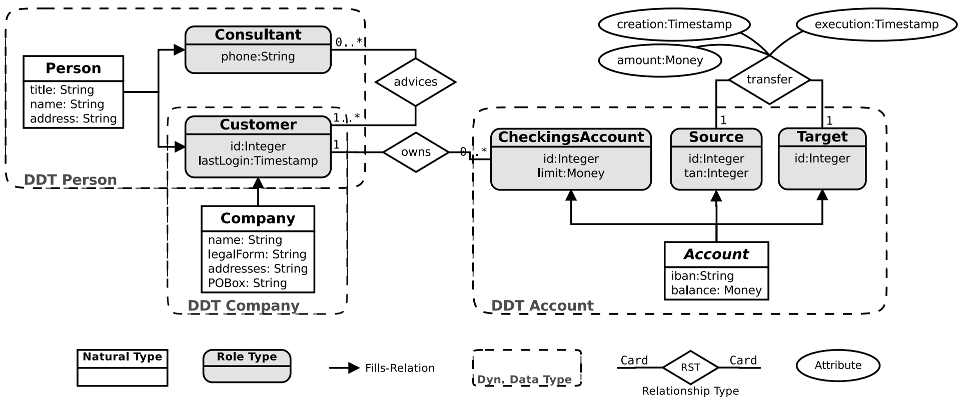

Figure 1: Role Modeling Example - Customers, Bank Accounts, and Transactions

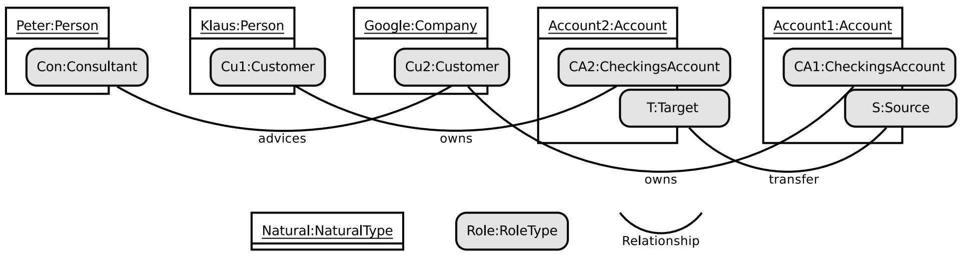

Figure 2: Instance of the Role Modeling Example (Fig. 1)

and the other two are *Customers*. Each of these customers *owns* one CheckingsAccounts, however, only *Google* is *advised* by *Peter*. Additionally, the model contains one transaction from *Account2* to *Account1*, which play the roles *Source* and *Target*, respectively, via the *transfer* relationship. To indicate, which Natural plays which Role, the Roles are placed at the border of their respective players. Throughout this paper, both the role model and its instance are used as a running example.

3 Dynamic Data Types and Relationship Types

This section introduces the notion of Relationship Types to represent dynamic relationships. To do this, we henceforth, present a formal model for both the type and the instance level. In particular, this definition is based on the formal model of *Dynamic Data Types* (DDT) presented in [JKVL14]. Consequently, we augment this model by including Relationship Types and the corresponding semantics. To be coherent, we present complete definitions and examples for both levels, but focus on Relationship Types in the discussion.

3.1 Type Level

In its core, the type level definition is comparable to Friedrich Steimann's type definition for *LODWICK* in [Ste99]. Dynamic Data Types consist of Natural Types and Role Types [JKVL14]. Roughly, Natural Types are rigid and non-founded. That means an instance of a Natural Type, denoted as Natural, loses its identity by changing the type. Naturals can exist independently of relations to other individuals in the systems. Role Types are exactly the opposite of Natural Types, they are non-rigid and founded, whereas the latter indicates their dependence on other individuals [Ste99, Gua92]. Consequently, Relationship Types are defined between two Role Types representing their two ends. For reason of simplicity and in contrast to *LODWICK* [Ste99], we focus solely on binary relationships with a limited set of cardinality constraints.

Definition 1 (Schema)**.** *Let NT be the set of all Natural Types, RT the set of all Role Types, RST be the set of Relationship Types, and $Card = \{0..1,\ 1..1,\ 1..*,\ 0..*\}$ be the set of available cardinalities with $NT \cap RT = \emptyset$. The schema s is then given by the tuple $s = (NT, RT, RST, \mathit{fills}, \mathit{rel})$, where $\mathit{fills} \subseteq NT \times RT$ is a relation for Natural Types fulfilling Role Types and $\mathit{rel} : RST \to RT \times Card \times Card \times RT$ is a function mapping a relationship type to the two Role Types at its end and to the corresponding cardinalities. Because Role Types cannot exist on their own and Relationships are prohibited to link the same Role Type, we constrain the schema s by:*

$$\forall rt \in RT\ \exists nt \in NT\ .\ (nt, rt) \in \mathit{fills} \tag{1}$$

$$\forall rst \in RST\ \nexists rt \in RT\ .\ \mathit{rel}(rst) = (rt, c_1, c_2, rt) \tag{2}$$

Applying these constraints ensures that (1) there exists an association of a Role Type to at least one Natural Type by using the *fills*-relation and that (2) each Relationship Type is defined between two distinct Role Types. Henceforth, we use the infix notation $nt\ \mathit{fills}\ rt$.

The cardinality constraints introduced here differ in their semantics from ER, where cardinality constraints directly describe the relationship between two static entity types. In our model we have to distinguish between two different relations, the *rel*-relation and the *fills*-relations. The former describes the relation between two distinct Role Types and constraints on this relation basically define how often an individual Role can be related to a counter Role. In contrast, the latter defines the relation between a Natural Type and a Role Type to form Dynamic Data Types. However, constraints on this particular relation result in constraining how often a Natural can play a Role of the same Role Type simultaneously, which is an internal Dynamic Data Type constraint. Such constraints, however, are beyond the scope of this paper. Thus, the *fills*-relation is assumed to be unconstrained in our definition. As consequence, one-to-one and one-to-many relationship constraints are only ensured between Roles and not their various players, such that traditional relationship constraints between two Natural Types cannot be enforced.

Example 1 *Let $bank = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ be a schema for the bank application shown in Fig. 1, where $NT = \{\mathit{Person}, \mathit{Company}, \mathit{Account}\}$ is the set of Natural Types, $RT = \{\mathit{Customer}, \mathit{Consultant}, \mathit{CheckingsAccount}, \mathit{Source}, \mathit{Target}\}$ are the Role Types,* *and* $RST = \{owns, advices, transfer\}$ *is the set of Relationship Types. Furthermore, the fills-relation is defined as:*

$$\begin{aligned} fills \overset{def}{=} \{&(Person, Customer), &&(Person, Consultant), &&(Company, Customer), \\ &(Account, Source), &&(Account, Target), &&(Account, CheckingsAccount)\} \end{aligned}$$

and the rel-function is defined for each Relationship Type as:

$$\begin{aligned} rel(owns) &\overset{def}{=} (Customer, 1..1, 0..*, CheckingsAccount) \\ rel(advices) &\overset{def}{=} (Consultant, 0..*, 1..*, Customer) \\ rel(transfer) &\overset{def}{=} (Source, 1..1, 1..1, Target) \end{aligned}$$

As it turns out, the $bank$ schema is a direct representation of the graphical model shown in Figure 1 such that each Role Type, Natural Type, and Relationship Type is an element of the corresponding set and each line is mapped to the *fills*-relation or the *rel*-function depending on its kind. As next step, the notion of a DDT is defined as a composition of a Natural Type nt and all Role Types it fills:

Definition 2 (Dynamic Data Type)**.** *Let* $s = (NT, RT, RST, fills, rel)$ *be a schema and* $nt \in NT$ *be a Natural Type. A Dynamic Data Type is then defined as* $ddt = (nt, RT_{nt})$ *with* $RT_{nt} \subseteq RT$ *such that* $RT_{nt} \overset{def}{=} \{rt \in RT \mid nt\ fills\ rt\}$

In addition to that, DDTs are considered as individual types consisting of one Natural Type in its center and several Role Types, which are related in the *fills*-relation. Besides that, each DDT gives rise to a number of Configurations in which instances of that DDT might appear. To put it differently, a Configuration specifies by which specific Role Types a DDT is extended. Thus, it is not defined by the user directly but deduced from the given *fills*-relation.

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

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

Example 2 *Applying these definitions to the schema* $bank = (NT, RT, RST, fills, rel)$ *gives rise to the following three DDTs:*

$$\begin{aligned} ddt_{Person} &= (Person, \{Customer, Consultant\}) \\ ddt_{Company} &= (Company, \{Customer\}) \\ ddt_{Account} &= (Account, \{CheckingsAccount, Source, Target\}) \end{aligned}$$

As a side note, each of these DDTs has a distinct number of possible Configurations, e.g., ddt_{Person} has four, ddt_{Company} has two, and so on. A Configuration characterizes a possible subset of Role Types extending a given DDT. For more information on Configurations the reader can refer to [JKVL14]. In conclusion, the extended schema additionally contains a set of Relationship Types which relate two Role Types and impose cardinality constraints on these relationships.

3.2 Instance Level

In contrast to Steimann's *LODWICK* [Ste99], we do not only instantiate Natural Types to Naturals but also Role Types to Roles. Thus, we are now able to directly represent relationships between two corresponding Roles.

Definition 4 (Instance)**.** *Let $s = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ be a schema, N the set of all Naturals, and R the set of all Roles with $N \cap R = \emptyset$. An instance i of this schema s is then defined as $i = (N, R, \mathit{type}, \mathit{plays}, \mathit{links})$, where $\mathit{type} : (N \to NT) \cup (R \to RT)$ is a polymorphic function assigning a distinct Natural Type or Role Type to each Natural or Role, respectively; $\mathit{plays} \subseteq N \times R$ is a relation defining which Naturals play which Roles; and $\mathit{links} : RST \to 2^{R^\varepsilon \times R^\varepsilon}$ is a function from Relationship Types to their extend set with $R^\varepsilon = R \cup \{\varepsilon\}$ and $\varepsilon \notin R$. As shorthand notation we define two index sets for Naturals and Roles.*

$$N_{nt} \overset{def}{=} \{n \in N \mid \mathit{type}(n) = nt\} \qquad \text{for } nt \in NT$$

$$R_{rt} \overset{def}{=} \{r \in R \mid \mathit{type}(r) = rt\} \qquad \text{for } rt \in RT$$

Moreover, we require the following five axioms to hold for any instance i of the schema s:

$$\forall r \in R\ \exists!\ n \in N\ .\ n\ \mathit{plays}\ r \tag{1}$$

$$\forall (n, r) \in \mathit{plays}\ .\ \mathit{type}(n)\ \mathit{fills}\ \mathit{type}(r) \tag{2}$$

$$\begin{aligned}\forall \mathit{rst} \in RST\ .\ &\mathit{rel}(\mathit{rst}) = (rt_1, c_1, c_2, rt_2) \land \\ &\mathit{links}(\mathit{rst}) \subseteq (R^\varepsilon_{rt_1} \times R^\varepsilon_{rt_2}) \setminus \{(\varepsilon, \varepsilon)\}\end{aligned} \tag{3}$$

$$\begin{aligned}\forall \mathit{rst} \in RST\ .\ &\mathit{rel}(\mathit{rst}) = (rt_1, c_1, c_2, rt_2) \land \\ &(\forall r_1 \in R_{rt_1}\ \exists (r_1, r) \in \mathit{links}(\mathit{rst})) \land \\ &(\forall r_2 \in R_{rt_2}\ \exists (r, r_2) \in \mathit{links}(\mathit{rst}))\end{aligned} \tag{4}$$

$$\begin{aligned}\forall \mathit{rst} \in RST\ \forall (r_1, r_2) \in \mathit{links}(\mathit{rst})\ .\ &\mathit{rel}(\mathit{rst}) = (rt_1, c_1, c_2, rt_2) \land \\ &(c_1 \in \{1..1, 1..*\} \Rightarrow r_1 \neq \varepsilon) \land \\ &(c_1 \in \{0..1, 1..1\} \Rightarrow \nexists (r_1', r_2) \in \mathit{links}(\mathit{rst})\ .\ r_1 \neq r_1') \land \\ &(c_2 \in \{1..1, 1..*\} \Rightarrow r_2 \neq \varepsilon) \land \\ &(c_2 \in \{0..1, 1..1\} \Rightarrow \nexists (r_1, r_2') \in \mathit{links}(\mathit{rst})\ .\ r_2 \neq r_2')\end{aligned} \tag{5}$$

$$\begin{aligned}\forall rst \in RST\ \forall c \in C\ \forall (r_1, r_2) \in (\mathit{links}(rst) \cap R \times R)\ .\ \\ (r_1, \varepsilon), (\varepsilon, r_2) \notin \mathit{links}(rst)\end{aligned} \tag{6}$$

Both, the *plays*-relation and the *links*-function are the instance level equivalent to the *fills*-relation and the *rel*-function on the type level. The former captures which Natural plays which Role at the moment. To simplify things, we use the infix notation $n\ \mathit{plays}\ r$ for this relation. Whereas the latter maps each Relationship Type to its extension, i.e. the set of tuples of Roles which are currently related. Consequently, each Relationship Type gives rise to several relationships represented by the aforementioned tuples.

Besides the general definition of instances of a schema, it is crucial to discuss some of the axioms required to hold for each instance, because these axioms enforce consistency with respect to the schema. The first two axioms (1-2) ensure that the *plays*-relation assigns exactly one Natural to each Role as its player and is type conform to the *fills*-relation defined in the schema. Similarly, the next three axioms (3-6) constrain the *links*-function to ensure its consistency according to the database schema. In particular, axiom (3) ensures that *links* return only those tuples of roles for a given Relationship Type that have a type matching the Role Types at the end of the Relationship Type's definition.[1] Similarly, axiom (4) ensures that each Role which has a type at the end of a Relationship Type *rst* must be present in that relationship's extension, i.e. it must be in one of the tuples in the set returned by $\mathit{link}(\mathit{rst})$. In addition to that axiom (5) applies the cardinality constraints defined on the schema level to the instance level. In detail, this axiom ensures that ε is only allowed in a tuple if the corresponding cardinality has a lower bound of zero, i.e. $0..1$ or $0..*$, and that each role is related to at most one other role if the corresponding cardinality has an upper bound of one, i.e. $0..1$ or $1..1$. These rules are applied to both ends of the relationship with respect to the specific cardinalities. Consequently, this axiom ensures that each relationship respects the cardinalities defined on the schema level. Last but not least, axiom (6) ensures that whenever a role is linked to the empty counter role ε it is not linked to another role within the same relationship type. In sum, this definition permits the consistency of each instance i of a schema s.

Notably, ε represents an empty counter role, which can be replaced by a counter role later on. This is necessary, because each role participating in a relationship, i.e., its Role Type is at one end of a Relationship Type, must be in the extended set of that relationship. However, this leads to problems for relationships with at least one lower bound of zero indicating that one side of the relationship is not necessarily linked to the other side. To overcome this issue, we introduce ε as an empty placeholder for the missing counter roles for such relationships. This may result in Roles that are not related to any other Role except an empty counter Role, which enables additional flexibility in our model. For instance, the *Customer* Role Type can be related to a *CheckingsAccount* Role Type and a *Consultant* Role Type. For a new *Customer* instance neither a responsible *Consultant* nor a valid *CheckingAccount* may exist, because the account validation process is not finished yet. Enforcing a strict counter Role in the sense of foundation avoids insertion of a new *Customer* Role until the validation process is finished. Despite of that, a Role related to an empty counter Role is still considered founded, because its existence is still dependent on its player. As a result, ε is crucial to model relationships with a lower bound of zero.

Example 3 *Let* $bank = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ *be the schema defined in Example 1. An instance of that schema is then* $i = (N, R, \mathit{type}, \mathit{plays}, \mathit{links})$*, where the components are defined as follows:*

$$N \overset{def}{=} \{\mathit{Peter}, \mathit{Klaus}, \mathit{Google}, \mathit{Account}_1, \mathit{Account}_2\}$$

$$R \overset{def}{=} \{\mathit{Con}, \mathit{Cu}_1, \mathit{Cu}_2, \mathit{CA}_1, \mathit{CA}_2, S, T\}$$

[1] Please note that $R^{\varepsilon}_{rt} = R_{rt} \cup \{\varepsilon\}$ enables the representation of roles to be related to ε.

$$\begin{aligned}
\textit{type} \overset{\textit{def}}{=} \{ & (\textit{Peter} \to \textit{Person}), (\textit{Klaus} \to \textit{Person}),\\
& (\textit{Google} \to \textit{Company}), (\textit{Account}_1 \to \textit{Account}),\\
& (\textit{Account}_2 \to \textit{Account}), (\textit{Con} \to \textit{Consultant}),\\
& (\textit{Cu}_1 \to \textit{Customer}), (\textit{Cu}_2 \to \textit{Customer}),\\
& (\textit{CA}_1 \to \textit{CheckingsAccount}), (\textit{CA}_2 \to \textit{CheckingsAccount}),\\
& (S \to \textit{Source}), (T \to \textit{Target})\}\\
\textit{plays} \overset{\textit{def}}{=} \{ & (\textit{Peter}, \textit{Con}), (\textit{Klaus}, \textit{Cu}_1), (\textit{Google}, \textit{Cu}_2),\\
& (\textit{Account}_1, \textit{CA}_1), (\textit{Account}_2, \textit{CA}_2),\\
& (\textit{Account}_2, S), (\textit{Account}_1, T)\}\\
\textit{links} \overset{\textit{def}}{=} \{ & (\textit{owns} \to \{(\textit{Cu}_1, \textit{CA}_1), (\textit{Cu}_2, \textit{CA}_2)\}), (\textit{transfer} \to \{(S, T)\}),\\
& (\textit{advices} \to \{(\varepsilon, \textit{Cu}_1), (\textit{Con}, \textit{Cu}_2)\})\}
\end{aligned}$$

Like the schema $bank$, the instance i is simply created from Figure 2 by taking all the Naturals and Roles into account, map their types accordingly, link the Roles to their players, and assigning a tuple for each relationship in the figure. In addition to that, the instance must also contain a tuple for the Role $\textit{Cu}_1$, because *Customer* participates in the *advices* Relationship Type. Consequently, this relationship contains a tuple relating $\textit{Cu}_1$ to the empty counter role ε.

Definition 5 (Dynamic Tuple)**.** *Let $s = (NT, RT, RST, \textit{fills}, \textit{rel})$ be a schema and $i = (N, R, \textit{type}, \textit{plays}, \textit{links})$ 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 $RT_n^d \subseteq RT_{nt}$ currently played by n:*

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

Case 1 *$RT_n^d = \emptyset$: then $d = (n)$*
Case 2 *$RT_n^d = \{rt_1, \ldots, rt_m\}$:*
then the Dynamic Tuple is defined as $d = (n,\ R_1^d,\ \ldots,\ R_m^d)$ with

$$R_i^d = \{r \in R_{rt_i} \mid n \textit{ plays } r\} \qquad \textit{for all } i \in \{1, \ldots, m\} \textit{ and } rt_i \in RT_n^d$$

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

A Dynamic Tuple is built around a certain Natural and several Role sets, where each Role set holds Roles of a specific Role Type. Moreover, 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 [JKVL14]. Finally, we can define whether two Dynamic Tuples are related by a certain relationship.

Definition 6 (Dynamic Relationships)**.** *Let $s = (NT, RT, RST, \textit{fills}, \textit{rel})$ be a schema, $i = (N, R, \textit{types}, \textit{plays}, \textit{links})$ an instance of s and $o, p \in N$ two naturals in i. The Dynamic Tuple $a = (o, R_1^a, \ldots, R_m^a)$ is then related to a Dynamic Tuple $b = (p, R_1^b, \ldots, R_n^b)$ with respect to a given Relationship Type $rst \in RST$, noted as **a** rst **b**, iff there is an $i \in \{1, \ldots, m\}$ and a $j \in \{1, \ldots, n\}$ such that $\exists r_1 \in R_i^a,\ r_2 \in R_j^b\ .\ (r_1, r_2) \in \textit{links}(\textit{rst})$.*

This definition allows to identify whether two Dynamic Tuples are related by a specific Relationship Type. Moreover, because each Natural can play Roles of the same type multiple times, Dynamic Tuples can be related to multiple Dynamic Tuples by the same Relationship Type. Consider, for instance, the instance i of the schema $bank$ containing the following Dynamic Tuples and Dynamic Relationships:

Example 4 *Let* $bank = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ *be the schema (Example 1) and* $i = (N, R, \mathit{type}, \mathit{plays}, \mathit{links})$ *its instance (Example 3); then* i *contains the following Dynamic Tuples:*

$$\begin{aligned} d_{Peter} &= (Peter, \{Con\}) & d_{Klaus} &= (Klaus, \{Cu_1\}) \\ d_{Google} &= (Google, \{Cu_2\}) & & \\ d_{Account_1} &= (Account_1, \{CA_1\}, \{T\}) & d_{Account_2} &= (Account_2, \{CA_2\}, \{S\}) \end{aligned}$$

Besides that, these Dynamic Tuples are related by the following Relationship Types:

$$\begin{aligned} &d_{Klaus}\ \mathit{owns}\ d_{Account_1} & &d_{Google}\ \mathit{owns}\ d_{Account_2} \\ &d_{Peter}\ \mathit{advices}\ d_{Google} & &d_{Account_2}\ \mathit{transfer}\ d_{Account_1} \end{aligned}$$

In sum, this formal model captures not only Dynamic Data Types and Relationship Types but also Dynamic Tuples and Dynamic Relationships.

4 RSQL Extensions for Relationships

The query language is the interface users and applications use to interact with database management systems. Hence, extending the DBMS also requires the interfaces being extended to the new specifications. SQL has been designed to store data in and retrieve them from relations, thus, it directly operates on tables having no notion of relationships and roles. Usually, relationships in relational DBMS are represented as additional columns in tables or as a separate table. Depending on the cardinality one of these options is implemented. In the most general case, a N:M cardinality between the participants forms a separate table that is handled as a normal table having foreign key constraints as references to other tables. The DBMS simply cannot distinguish between tables storing entities and tables that store relationships. The other cases, 1:N and 1:1, lead to additional columns in tables that store entities, which means entities and relationships are mixed in that representation. Attributes of relationships can no longer be distinguished from entity attributes. It becomes worse, if cardinality changes, since the mapping of relationships depends on that cardinality. For instance, a 1:N cardinality is changed to N:M. That means, the additional columns introduced in one of the participant tables will be dropped and the relationship forms a separate table including the relationship attributes. All queries considering this relationship have to be rewritten, which makes database schema and queries fragile. Additionally, the resulting database schema significantly differs from the conceptual model, due to a lot of mappings to store the concepts and relationships relationally. That difference disables users to query for desired concepts without specific database schema knowledge.

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

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

⟨config-expression⟩ ::= ( ⟨nt-name⟩ ⟨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⟩ ::= ⟨rt-name⟩ ⟨rtAbbreviation⟩

⟨relation-clause⟩ ::= ⟨rtAbbreviation⟩ WITH ⟨rtAbbreviation⟩
    USING ⟨rst-name⟩ ⟨rstAbbreviation⟩
```

Figure 3: Data Query Language Syntax

To enable users and applications to store and query for role-based data, we proposed RSQL as extension to SQL [JKVL14]. RSQL enables role specific integrity constraints on the query language level. It relies on Dynamic Data Types on the type level and Dynamic Tuples on the instance level, but without considering relationships between them. As shown before, traditional relational methods are infeasible for representing relationships properly in a relational DBMS. To overcome these limitations of traditional relational DBMS and to enable cardinality-independent first-class relationships, RSQL has to be extended. This extension comprises the creation, manipulation, and retrieval of relationships.

4.1 Data Query Language

The *Data Query Language* (DQL) is used to retrieve stored data from DBMS and in case of RSQL, qualified Dynamic Tuples are returned based on given Configurations. Figure 3 provides an overview on the select statement. In general, the select statement starts with 'SELECT' followed by the ⟨*projection-clause*⟩. This clause filters columns in the result set. After that, the ⟨*from-clause*⟩ describes Configurations based on ⟨*config-expression*⟩ and their relationships by the ⟨*relation-clause*⟩, followed by an optional ⟨*where-clause*⟩.

To handle complexity of Dynamic Data Types, RSQL provides ⟨*config-expression*⟩ as a sophisticated type description. A single ⟨*config-expression*⟩ describes a set of Configurations. Dynamic Tuples matching one of these Configurations will be added to the result. Specifying multiple ⟨*config-expression*⟩ in one select statement produces a Cartesian product. To filter the Cartesian product based on related Dynamic Tuples, the ⟨*relation-clause*⟩

```
SELECT p2.name, a.balance
FROM    Person p1 PLAYING Consultant co,
        Person p2 PLAYING Customer c,
        Account a PLAYING CheckingsAccount ca,
        RELATING co WITH c USING advices ad,
        RELATING c WITH ca USING owns o
WHERE ca.limit > 1000 AND p1.name="Peter";
```

Figure 4: Example Select Query

inside the ⟨*from-clause*⟩ is used. This is done with the prefix 'RELATING' followed by two Role Type abbreviations combined by a 'WITH'. To define which Relationship Type has to be used, its name and the corresponding alias after 'USING' have to be specified. This ⟨*relation-clause*⟩ defines, via which Role Types Dynamic Tuples have to be connected with each other. Additionally, the Relationship Type and its abbreviation have to be specified to enable projection operations on its attributes. This specification is also necessary, because Role Types can be connected multiple times. The Role Type abbreviations have to be defined in the ⟨*config-expression*⟩ of a Dynamic Data Type in advance. Of course, users can define multiple ⟨*relation-clause*⟩ to filter multiple connected Dynamic Tuples.

Implementing a ⟨*relation-clause*⟩ enables users to produce more robust queries than in SQL. Relationships become explicit to users, thus, they do not have to be mixed with other concepts. Attributes of relationships are addressed via the relationship itself. This enables cardinality free querying, since the user specifies a relationship and not its implementation within the DBMS. If the cardinality of a relationship changes conceptually, the DBMS schema will slightly change, only by altering the cardinality of the corresponding Relationship Type. But this means, no attributes have to be moved to other relations and additional tables are avoided. Thus, changing cardinality does not affect the query statement at all, but its results, so that applications have to handle the query result differently. Moreover, it is more intuitive for users writing queries with explicit relationships to distinguish between entities and their relationships.

Figure 4 illustrates a query for the *balance* of all *Customers* of *Peter* who has an *Account* limit that is higher than 1000. At first, three Configurations are specified, two are based on DDT *Person* and one is based on DDT *Account*. Afterwards, these Configurations are related by Relationship Types *advices* and *owns* within the *from-clause*. This filter aims at selecting only related Dynamic Tuples. Lastly, the result is filtered by the the *checkingsAccount* limit higher than 1000 and *Consultant's* name *Peter*.

4.2 Data Manipulation

The *Data Manipulation Language* (DML) in RSQL creates, deletes, and indirectly modifies Dynamic Tuples. Typically, Roles and Naturals are addressed by this language part and the DBMS automatically creates Dynamic Tuples. Adding Relationships between

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

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

⟨insert-rst⟩ ::= INSERT INTO ⟨rst-name⟩ ( ( ⟨attribute-name⟩ ( , ⟨attribute-name⟩ )* )
    VALUES ( ⟨value-expression⟩ ( , ⟨value-expression⟩ )* ) )?
    INSTANCES ⟨rtAbbreviation⟩ , ⟨rtAbbreviation⟩
    FROM ⟨config-expression⟩ , ⟨config-expression⟩ ( WHERE ⟨where-clause⟩ )?

⟨update-rst⟩ ::= UPDATE ⟨rst-name⟩
    SET ⟨assignment-expression⟩ (WHERE ⟨where-clause⟩)?

⟨delete-rst⟩ ::= DELETE FROM ⟨rst-name⟩ ( WHERE ⟨where-clause⟩ )?
```

Figure 5: Data Manipulation Language Syntax

Roles to the system implies large-scale changes of how Dynamic Tuples are handled. The DBMS, as single point of truth in a distributed software system, has to ensure consistency and in case of RSQL it has to ensure role-specific integrity conditions. This also includes ensuring the cardinality of relationships. For instance, Roles of Role Types that have to be part of a relationship cannot exist on their own. The example illustrated in Figure 1 shows such a Relationship Type. There, a *Source* and *Target* Role Type are connected by a *transfer* Relationship Type having 1..1 and 1..1 cardinality. This means, for each *Source* there has to be a *Target* and neither of them can exist without being in the *transfer* Relationship. That small example requires two Roles to be created at the same time and relating them. To ensure consistency with respect to the model, the insert operations must be embedded in a transaction and at the end of each transaction the validity is checked.

The DML comprises insert, update, and delete statements for Naturals, Roles and Relationships. In Figure 5 the syntax of relationship focused statements is presented. A Natural can be added to the system by using the ⟨*insert-nt*⟩ statement. It is comparable to inserting a tuple into a relational table using SQL. The ⟨*insert-rt*⟩ statement extends a Dynamic Tuple by a certain Role. It also comprises an attribute and value assignment as well as a configuration description to determine the Dynamic Tuple that has to be extended. In addition to these statements, the new ⟨*insert-rst*⟩ statement has been introduced which relates two existing Roles of Dynamic Tuples to each other. The process of relating Dynamic Tuples is the following: Firstly specify which Role Type the Roles belong to, secondly specify the Dynamic Tuples in which the corresponding Roles are present, and thirdly filter the particular Roles. The statement also starts with '`INSERT INTO`' followed by the Relationship Type's name. Afterwards, users can specify attributes and values for those

```
BEGIN TRANSACTION;
INSERT INTO Source (id, tan) VALUES (1, 0321) OF
        Account a WHERE a.iban = "0815";
INSERT INTO Target (id) VALUES (2) OF
        Account a WHERE a.iban = "4711";
INSERT INTO transfer (creation, execution, amount)
        VALUES ("11.09.2014", "12.09.2014", 500)
        INSTANCES s, t
        FROM    Account a1 PLAYING Source s,
                Account a2 PLAYING Target t
        WHERE s.id = 1 AND t.id = 2;
COMMIT;
```

Figure 6: Data Manipulation Language Examples within a Transaction

in an optional clause. The attribute and value assignment is optional, since Relationship Types without attributes may exist. Then, the first process step follows by stating the corresponding Role Type with their abbreviations in the '`INSTANCE OF`' clause by. Afterwards, the '`FROM`' clause follows having exactly two ⟨*config-expression*⟩ to specify the second process step. There, each Role Type stated in the first step has to be present in one ⟨*config-expression*⟩. So far, users defined via which Role Types the resulting Dynamic Tuples have to be connected. In case Roles of the same Role Type are present multiple times in the same Dynamic Tuple, the '`WHERE`' clause has to filter exactly one of those Roles to determine the unique Role that will be related. In general, inserting a Relationship is independent of the Roles that might have been inserted in the same transaction, it connects the Roles that have been described in the '`FROM`' and '`WHERE`' clauses.

The ⟨*update-rst*⟩ statement updates values in Relationships. It is similar to updating Naturals or Roles. Users have to specify '`UPDATE`' and the corresponding Relationship Type name at the statement's very beginning. The system will figure out for itself which type has to be updated by the user provided name, since names have to be unique over all types. To delete Relationships and detach the corresponding Roles, the ⟨*delete-rst*⟩ statement has to be used. The syntax of this statement is similar to traditional delete statements, however, the Relationship Type name has to be specified. If a Relationship is deleted, Roles participating in this Relationship may also be deleted in case they are constrained with a lower bound cardinality of 1 and the last participating Role on one relationship side. This can cause knock-on effects for a series of other Roles and Relationships and the user must be aware of this. Roles participating in this particular Relationship as optional with a lower bound cardinality of 0, are not affected. The other way around, a deletion of a Role may also cause the deletion of a Relationship.

The statements in Figure 6 show an example for inserting two Roles and relating them within a transaction. At first, we create a new transaction, because the following insertions will lead to inconsistent states during the transaction. Afterwards, two Roles are inserted into the database. After each insertion, the stored data is invalid with respect to the schema, that requires that Source and Target have to be connected to exactly one counter Role. To

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

⟨create-rt⟩ ::= CREATE ROLETYPE ⟨rt-name⟩
      ( ⟨attribute-definition⟩ ( , ⟨attribute-definition⟩ )* )
      PLAYED BY ⟨nt-name⟩ ( , ⟨nt-name⟩ )*

⟨create-rst⟩ ::= CREATE RELATIONSHIPTYPE ⟨rst-name⟩
      ( ( ⟨attribute-definition⟩ ( , ⟨attribute-definition⟩ )* ) )?
      CONSISTING OF ( ⟨relation-participation⟩ ) AND ( ⟨relation-participation⟩ )

⟨relation-participation⟩ ::= ⟨rt-name⟩ BEING ( 0 | 1 ) .. ( 1 | * )

⟨drop-rst⟩ ::= DROP RELATIONSHIPTYPE ⟨rst-name⟩
```

Figure 7: Data Definition Language Syntax

restore the validity we insert a new Relationship of the type *transfer*. In the first process step we state the desired Role Types. Afterwards, we define two distinct Dynamic Tuples in the FROM clause, one playing a *Source* Role and one playing a *Target* Role. Usually, this will not be the first transfer on the corresponding accounts, which implies that both Dynamic Tuples of the *Accounts* probably have multiple *Source* and *Target* Roles already. To uniquely identify the desired Roles, we filter the Dynamic Tuples in the '`WHERE`' clause by the *Source id* and *Target id* attribute. Finally, the transaction will be checked and, if valid, committed.

4.3 Data Definition Language

The *Data Definition Language* (DDL) specifies the schema of a database. There, all type information will be provided to the DBMS. Dynamic Data Types are created, altered and dropped indirectly. Creating a new Natural Type creates a new Dynamic Data Type and creating a new Role Type extends existing ones. Role Types, as parts of Dynamic Data Types, can be related by Relationship Types. Also cardinality of Relationship Types is defined using DDL statements.

A Natural Type is created by a ⟨*create-nt*⟩ statement. This statement starts with '`CREATE NATURALTYPE`' followed by a distinct name and a set of attribute definitions. Note, all types form a common name space, such that each defined type has a distinct name. An existing DDT is extended by Role Types using the ⟨*create-rt*⟩ statement. After '`CREATE ROLETYPE`' and a unique name, attributes have to be specified, and at least one Natural Type has to be given after '`PLAYED BY`' clause. This clause avoids isolated Role Types. Alter and drop statements also exist for those types, but are out of scope here.

To relate Dynamic Data Types with each other, Relationship Types have to be used. This

```
CREATE RELATIONSHIPTYPE advices CONSISTING OF
        (Consultant BEING 0..∗) AND (Customer BEING 1..∗);
CREATE RELATIONSHIPTYPE transfers
        (creation Timestamp, execution Timestamp, amount Money)
        CONSISTING OF (Source BEING 1..1)
        AND (Target BEING 1..1);
```

Figure 8: Data Definition Language Examples

enables the DBMS to handle relationships explicitly and provides information about relationship attributes and cardinality. This information will be used by the DBMS to check the validity of DML operations. A Relationship Type is created using the ⟨*create-rst*⟩ statement. It starts with '`CREATE RELATIONSHIPTYPE`' and a unique name. After that, it is optional to specify attributes for this Relationship Type. This is optional, since Relationship Types without attributes may exist. To relate Role Types two ⟨*relation-participation*⟩ clauses have to be combined with a prefix '`CONSISTING OF`'. The first ⟨*relation-participation*⟩ denotes the left part of the Relationship Type and the second one the right part. A ⟨*relation-participation*⟩ consists of a Role Type name and its cardinality. The Role Type name specifies the corresponding Role Type that will become part of this relationship. The cardinality is constructed using '`BEING`' followed by the lower bound. There, users can choose between "0" and "1". Afterwards, the upper bound is defined while having two options: "1" or "*". Relationship Types can be altered and dropped as well, but for reasons of brevity we focused on creation only. RSQL also provides a statement to alter Relationship Type attributes, the referenced Role Types, and cardinality, but here we focused on the creation of Relationships.

Figure 8 illustrates syntax and semantics of the DDL. At first, the type level is created assuming that the Dynamic Data Type *Person* and DDT *Account* already exist. The first statement establishes a Relationship Type *advices* between *Consultant* and *Customer* without attributes. Moreover, the *Consultant*'s cardinality is 0 to $*$ and the *Customer*'s is 1 to $*$ denoting that each *Consultant* needs at least one *Customer*. A *Consultant* will be forced to be related to a *Customer* and has to participate in such a relationship. In contrast, the *Customer* can exist without this relationship, but can also have more than one Consultant. The second statement creates a new Relationship Type *transfer* between *Source* and *Target* with three attributes as well as 1 to 1 and 1 to 1 cardinality, respectively.

5 Comparison

The role concept has been proposed by Bachman in 1977 [BD77]. Over the past decades the idea of separating different concerns of data objects has been adapted to many modeling languages, especially in conceptual modeling. Henceforth, we first classify our approach and discuss other related query languages, afterwards.

5.1 Classification

To classify our role-based data model, we employ a previously defined classification scheme developed to evaluate role-based modeling and programming languages [Ste99] and its extension [KLG$^+$14]. Consequently, we can classify RSQL's formal model in accordance with the features of roles: *(1) Roles have properties and behaviors.* Yes, roles have properties. *(2) Roles depend on relationships.* Yes, roles can be defined to depend on relationships. *(3) Objects may play different roles simultaneously.* Yes, by definition. *(4) Objects may play the same role (type) several times.* Yes. *(5) Objects may acquire and abandon roles dynamically.* Yes. *(6) The sequence of role acquisition and removal may be restricted.* Partially, because cardinality constraints can be used to restrict the order of role creation. *(7) Unrelated objects can play the same role.* Yes. *(8) Roles can play roles.* No. Similar to *Lodwick* we disregard roles playing roles. *(9) Roles can be transferred between objects.* Yes, this can be done by altering the player of a role. *(10) The state of an object can be role-specific.* Yes, because a Dynamic Tuple combines the state of the natural and its roles. *(11) Features of an object can be role-specific.* Yes, because an object is defined as a Dynamic Data Type. *(12) Roles restrict access.* This feature is not applicable, because we currently assume that each user has full access to all defined Dynamic Data Types. *(13) Different roles may share structure and behavior.* No, currently, our formal model does not support inheritance between roles. *(14) An object and its roles share identity.* Yes, because they are encapsulated in a Dynamic Tuple inheriting its identity from the object. *(15) An object and its roles have different identities.* Yes, they have different identities in the formal model. *(16) Relationships between roles can be constrained.* Yes, by supporting the definition of cardinality constraints. *(17) There may be constraints between relationships.* No, currently the definition of inter relationship constraints is not allowed. *(18) Roles can be grouped and constrained together.* No. *(19) Roles depend on compartments.* No, there is no notion of Context or Compartment in our model. In consequence, all other features, i.e., Feature 20 to Feature 26, can not be fulfilled at all. As a result, this apporach focuses on the relational nature of roles, but allows for shared as well as own identities for roles by introducing Dynamic Data Types.

5.2 Related Work

Henceforth, we recollect the various related approaches with respect to the underlying data model and role-based query language.

Considering the former, the definition presented here has similarities to *LODWICK*'s definition in [Ste99]. Both are founded on Naturals and Roles on the type level, but on the instance level roles do not exist in *LODWICK*. Furthermore, *LODWICK*'s formal model is able to express arbitrary n-ary relationships, RSQL's data model can only express binary ones. Additionally, Roles and Naturals in RSQL have different identities, but they are combined to Dynamic Data Types, which can be identified solely by a specific Natural. However, Dynamic Tuples in RSQL are identified by value based identification rather than unique object ids, i.e., they inherit their identity directly from the contained Natural.

In direct comparison to *LODWICK*, RSQL is superior because it can resolve the identity issue of roles and includes the definition of cardinality constraints.

The related query languages can be grouped along three different classes: role-based query languages, conceptual query languages, and regular query languages. In the first class, the *Information Networking Model* (INM) uses roles in relationships to describe that objects play roles in a certain relationship to other objects [LH09a, LH09b]. Like RSQL and the underlying formalism, it also supports dynamic and many-faceted object types. Furthermore, INM provides features to model context-dependent information and thus, it introduces contexts to group roles. Relationships also exist in INM, first as normal relationships or role relationships. Via role relationships context information is modeled and normal relationships are the same as in traditional modeling languages. Nevertheless, relationships cannot be constrained by cardinality. Moreover, there exists a query language for INM, called *Information Query Language* (IQL) [HFL10]. The IQL utilizes tree expressions, like *XPath*, to hierarchically describe the desired data. Moreover, 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.

Conceptual query languages abstract implemented database models to conceptual models. Hence, users query based on conceptual levels without any knowledge about the implemented data representation. This approach has become popular in the 1980's and 1990's. At that time, a lot of entity-relationship based conceptual query languages for relational DBMS have been proposed, since data were modeled using ER and stored relationally. Examples of query languages based on ER or extended ER are *SQL/EER* [HE92], the query language presented in [LT94], or *CABLE* [Sho79]. Nevertheless, these query languages are usually mapped to SQL, thus they only abstract for users and relational DBMS remain the same. Moreover, they have no notion of roles but of relationships and RSQL comprises both notions. However, query languages exist that have a notion of roles, for instance *ConQuer* [BH96] that is designed for *Object Role Modeling* (ORM) [Hal98]. In contrast to RSQL, *ConQuer* queries are also mapped to SQL.

In comparison to general query languages, RSQL enables users and applications to query information based on a conceptual model, which relates it to conceptual query languages. Nevertheless, RSQL does not abstract database objects to conceptual entities. RSQL is a regular query language that directly describes database objects that are handled in a DBMS without abstraction. This also requires introducing new database objects in a DBMS and creating new first-class citizen. The only approach that has introduced roles as first-class citizen in a DBMS is *DOOR* [WCL97]. DOOR also has the notion of rigid and non-founded types (Natural Types in RSQL) and founded, non-rigid types (Role Types) on the type level as well as on the instance level. However, DOOR does not support relationships between roles.

6 Conclusions

To lower the mismatch between role-based conceptual models and their DBMS implementation, we extended our data model and query language RSQL by introducing first-class relationships. Hence, role and relationship semantics are preserved in the DBMS and not hidden in mapping engines. This improves the interoperability in highly distributed information systems that run role-based software and allows DBMS to guarantee role specific as well as relationship specific consistency constraints. In detail, we introduced Relationship Types on the type level and Relationships on the instance level in our data model. Additionally, cardinality constraints are directly bound to this newly introduced types. Consequently, RSQL has been adapted to the new data model by introducing special statements to create Relationship Types and insert Relationships, respectively. By making relationships explicit, they do not have to be mapped to tables and queries can be created independently of the relationship implementation. This has the following two effects. Firstly, cardinality changes no longer entail query reconstruction, which produces more robust queries. Secondly, users can create queries without having knowledge about the relational mapping and normalization of relationships. Additionally, the DBMS gains knowledge on the stored data objects that can be used to optimize query processing and enforce relationship consistency constraints. In summary, RSQL enables users and applications to directly represent and query for role-based data objects including their relationships and, at the same time, the implementing DBMS can ensure role-based consistency constraints.

We have presented a formal data model definition and a query language that captures that definition. To constrain Dynamic Data Types and the set of Configuration they describe, we will introduce constraints between Role Type (e.g. prohibition of two Role Types). Additionally, we aim at cardinality constraints on the fills-relation to represent real one-to-one relationships in our system. As a next step, we will implement both the data model and RSQL within an extended relational DBMS. Hence, we will introduce new database objects to teach the DBMS the notion of Dynamic Data Types and Relationship Types. To process Dynamic Tuples, we are going to implement specialized database operators that can be used for query processing as well as for query optimization.

Acknowledgement

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).

References

[BD77] Charles W. Bachman and Manilal 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.

[BH96] A.C. Bloesch and T.A. 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.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, March 1976.

[Gua92] Nicola Guarino. Concepts, Attributes and Arbitrary Relations: Some Linguistic and Ontological Criteria for Structuring Knowledge Bases. *Data & Knowledge Engineering*, 8(3):249 – 261, 1992.

[Hal98] Terry 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.

[HE92] Uwe Hohenstein and Gregor Engels. SQL/EER — Syntax and Semantics of an Entity-relationship-based Query Language. *Information Systems*, 17(3):209–242, May 1992.

[HFL10] Jie Hu, Qingchuan Fu, and Mengchi Liu. Query Processing in INM Database System. In Lei Chen, Changjie Tang, Jun Yang, and Yunjun Gao, editors, *Web-Age Information Management*, volume 6184 of *Lecture Notes in Computer Science*, pages 525–536. Springer Berlin Heidelberg, 2010.

[JKVL14] Tobias Jäkel, Thomas Kühn, Hannes Voigt, and Wolfgang Lehner. RSQL - A Query Language for Dynamic Data Types. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 185–194, New York, USA, 2014. ACM.

[KLG$^+$14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-based Modeling and Programming Languages. In *7th International Conference on Software Language Engineering (SLE)*. Springer, 2014.

[LH09a] Mengchi Liu and Jie 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.

[LH09b] Mengchi Liu and Jie 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.

[LT94] Michael Lawley and Rodney Topor. A Query Language for EER Schemas. In *Proceedings of the 5th Australian Database Conference*. Global Publication Service, 1994.

[RJB10] 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.

[Sho79] Arie Shoshani. CABLE: A Language Based on the Entity-Relationship Model. Technical report, Lawrence Berkeley Laboratory, 1979.

[Ste99] Friedrich Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data & Knowledge Engineering*, 35(1):83 – 106, 1999.

[WCL97] R.K. Wong, H.L. Chau, and F.H. 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.