TECHNISCHE UNIVERSITÄT DRESDEN

Role-based Data Management

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der

Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Wirt.-Inf. Tobias Jäkel
geboren am 2. August 1985 in Altdöbern

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke-Universität Magdeburg
Fakultät Informatik
Institut für Technische und Betriebliche Informationssysteme
Arbeitsgruppe Datenbanken & Software Engineering
39106 Magdeburg

Tag der Verteidigung: 24. März 2017

Dresden, im März 2017

ABSTRACT

Database systems build an integral component of today's software systems and as such they are the central point for storing and sharing a software system's data while ensuring global data consistency at the same time. Introducing the primitives of roles and their accompanied metatype distinction in modeling and programming languages, results in a novel paradigm of designing, extending, and programming modern software systems. In detail, roles as modeling concept enable a separation of concerns within an entity. Along with its rigid core, an entity may acquire various roles in different contexts during its lifetime and thus, adapts its behavior and structure dynamically during runtime.

Unfortunately, database systems, as important component and global consistency provider of such systems, do not keep pace with this trend. The absence of a metatype distinction, in terms of an entity's separation of concerns, in the database system results in various problems for the software system in general, for the application developers, and finally for the database system itself. In case of relational database systems, these problems are concentrated under the term role-relational impedance mismatch. In particular, the whole software system is designed by using different semantics on various layers. In case of role-based software systems in combination with relational database systems this gap in semantics between applications and the database system increases dramatically. Consequently, the database system cannot directly represent the richer semantics of roles as well as the accompanied consistency constraints. These constraints have to be ensured by the applications and the database system loses its single point of truth characteristic in the software system. As the applications are in charge of guaranteeing global consistency, their development requires more effort in data management. Moreover, the software system's data management is distributed over several layers, which results in an unstructured software system architecture.

To overcome the role-relational impedance mismatch and bring the database system back in its rightful position as single point of truth in a software system, this thesis introduces the novel and tripartite RSQL approach. It combines a novel database model that represents the metatype distinction as first class citizen in a database system, an adapted query language on the database model's basis, and finally a proper result representation. Precisely, RSQL's logical database model introduces Dynamic Data Types, to directly represent the separation of concerns within an entity type on the schema level. On the instance level, the database model defines the notion of a Dynamic Tuple that combines an entity with the notion of roles and thus, allows for dynamic structure adaptations during runtime without changing an entity's overall type. These definitions build the main data structures on which the database system operates. Moreover, formal operators connecting the query language statements with the database model data structures, complete the database model. The query language, as external database system interface, features an individual data definition, data manipulation, and data query language. Their statements directly represent the metatype distinction to address Dynamic Data Types and Dynamic Tuples, respectively. As a consequence of the novel data structures, the query processing of Dynamic Tuples is completely redesigned. As last piece for a complete database integration of a role-based notion and its accompanied metatype distinction, we specify the RSQL Result Net as result representation. It provides a novel result structure and features functionalities to navigate through query results. Finally, we evaluate all three RSQL components in comparison to a relational database system. This assessment clearly demonstrates the benefits of the roles concept's full database integration.

ACKNOWLEDGMENTS

At this prominent position, I would like to express my special thanks to my supervisor Wolfgang Lehner, for giving me the opportunity to be part of his amazing research group. Throughout my entire time in his group, he maintained a very special research atmosphere that greatly supported my work and research. Moreover, he initially introduced me to the research field of database systems and gave me the freedom to find my very individual spot in this research area. As supervisor, he was always available for discussions and supported me with many valuable advices, especially on my research paper drafts. I am also very thankful for giving me the opportunity to spend some time at Brown University in the United States. This would not have been possible without his help and his global network of researchers. Additionally, he provided me with the opportunity to attend various interesting conferences all around the world. Thank you for your excellent support.

Special thanks go to Thomas Kissinger, who initially encouraged me to get in touch with Wolfgang to discuss available research opportunities in his group. Without our initial discussion on computer science research and his suggestion to get in touch with Wolfgang, I probably would not have started my doctorate project. I also would like to thank Dirk Habich and Hannes Voigt for acting as my co-mentors and provide me with valuable advices. Especially in the beginning, they were very patient with me and my first steps in writing research papers. They always had time to review my work and provided suggestions to improve it. Additionally, they reviewed this thesis and improved it by providing comments and suggestions. I am very grateful to Gunter Saake for co-refereeing this thesis. Special thanks go to my student Stefan Hinkel, for his significant contributions on the RSQL DB prototype. I also thank my other students, who were involved in my research projects.

Moreover, I like to thank my past and current colleagues for providing a relaxing and entertaining research atmosphere, which made my stay in the group very special. Especially, the off-topic discussions in the coffee kitchen, the Friday afternoon projects, the group retreats, and Mario Kart challenges. In detail, thanks to Ahmad, Alex, Annett, Benjamin, Bernd, Claudio, David, Elena, Elvis, Frank, Gunnar, Ines, Ismail, Johannes, Julian, Juliana, Kai, Kasun, Katrin, both Larses, Laurynas, Maik, Marcus, Matthias, Michael, Patrick, Rihan, Robert, Steffen, Till, Tim, Tomas, both Ulrikes, and Vasileios.

As member of the RoSI research training group, I am very grateful for the fruitful discussions with my fellow researchers. Especially, Thomas Kühn significantly contributed to my work. We had long-lasting discussion on modeling issues, implementation alternatives, and theoretical corner cases. I really enjoyed our fruitful collaboration. Additionally, my thanks go to Friedrich Steimann, who provided additional research material and helpful comments on my work. I thank Sebastian Götz and Sebastian Richly for their support and advices. Furthermore, I would like to thank all RoSI members and involved researchers for the fruitful workshops, especially the late-night discussions. In particular, thanks to İsmail , Jan, Johannes, Mariam, Markus, Martin, Max, Philipp, Steffen, and Stephan for being great roommates as well as fellow researchers.

My special thanks go to Ugur Cetintemel, who mainly hosted me at Brown University. Additionally, these thanks go to Tim Kraska and Carsten Binnig for being great hosts at Brown and providing me a different perspective on my own research.

Very special thanks go to my wife Sindy, who was a great support throughout the entire doctorate project. She always encouraged me to keep on my research topic and finish the doctoral project. Furthermore, she patiently listened to my never-ending monologues about roles, database systems, and roles and database systems in combination, even though computer science is not her favorite topic. She also joined me for our stay in the United States and made it a very special experience. Thank you for going this way with me; for making it our way.

I am also very thankful to my family and friends. Moreover, I would like to thank my parents and my parents-in-law for always supporting me. Additionally, very special thanks to my brother Tommy, who always kept my car running and for the fun moments. Thank you, Jens, Ronny, and Scott for reviewing and improving this thesis with your comments and remarks. Finally, I would like to thank my friends, who made stressful times less stressful. Thank you all for the fun and entertainment.

Tobias Jäkel
Dresden, January 30, 2017

CONTENTS

INTRODUCTION

The digital revolution has been shaping a world in which software dominates our day-to-day life, from tracking our exercise results on a wearable, over smart homes in which the coffee machine is customizable by a smartphone application, to self-driving cars. Hence, nowadays software is ubiquitous [65], but in different kinds of ubiquity. At first, software is physically ubiquitous in space, by running on mobile devices that constantly move between different locations and different users. Secondly, software runs based on the Internet which enables a logical ubiquity in space. As the Internet-based software is available all over the globe, it exposes its end points in many countries with different jurisdictions, or different cultures. For instance, imagine Netflix[1] as a worldwide streaming platform. The movies you are able to watch on Netflix varies dramatically, depending on the location you are accessing this platform. Moreover, you can access Netflix from different devices and depending on your device, some functionalities are available and some not. Finally, today's software is ubiquitous in time, in terms of longevity. Applications are running for decades and face a constant change. This may be caused by varying legal regulations, an adapted business model, or novel technology. As example assume the Amazon online shop[2], that started as online bookstore and evolved to a huge online marketplace where you can buy almost everything.

Traditionally, software is well structured in types by embedding its behavior in methods and functions. Once the software is compiled, its behavior is fixed for its whole lifetime. Hence, changing its behavior results in a recompilation and application restart. This characteristic is inherited from the types to the runtime objects, once instantiated it is bound to a certain type and the sets of attributes and methods are fixed and only the values can be changed. However, such software has no explicit notion of a context, hence, runtime objects act identically static, independent of the context [41]. That does not mean software cannot be written to be context-sensitive, rather this means that context adaptation mechanisms have to be implemented along with the regular functional behavior. For instance, the same method call can result in different outputs, depending on an attribute value that is checked during the method execution, but this check is performed explicitly and has to be implemented manually, as shown in [6]. Such code does not explicitly distinguish between a context adaptation mechanism and the functional behavior. In particular, simulating contexts can be manually implemented by conditional statements (usually stated by an if-condition) [6]. That sounds simple, but as a context adaptation may depend on various information, this check can become very complex. For instance, imagine a user of a huge online marketplace in Germany. Once the user is logged into his or her account, the software checks the location first. Even if the marketplace's premium service is equal in all countries, the website language will be set depending on the location. Next, it checks for a premium account and maybe for a discounted premium membership. Depending on this user information, a different front-end is displayed and maybe different discounts or special offers are available. All these checks have to be programmed along with the functional behavior of displaying the landing page. In the ubiquitous scenario in which changes occur very frequently, it is infeasible to manually revise the context checks as well as recompile and restart the whole application.

From Static Types to Adaptive Structures

To cope with the challenges of ubiquitous software, researchers have proposed several approaches, including the concept of roles that has been initially introduced by Bachman and Daya in computer science [8]. The key idea of roles, as design principle, is to split an entity into several metatypes, especially the entity core and the role types an entity may play over its lifetime. This idea is visualized in Figure 1.1 and contrasted to a traditional static typing approach. On the left-hand side the traditional approach models all potential behavior in a single class to avoid entity reinstantiation. In detail, the core entity information and the premium membership specific behavior, like *orderPremium()*, and

[1] `https://www.netflix.com/`

[2] `https://www.amazon.de/`

structure is included in a single type. Thus, all methods and attributes are available for the whole entity lifetime and it has to be manually checked within the methods either an *orderPremium()* method is callable or not. In contrast, the role-based modeling approach on the right-hand side specifies the premium membership specific behavior in a separate role type named *PremMember*. Only if a role of this type is played, the corresponding method and attributes are available within the entity.

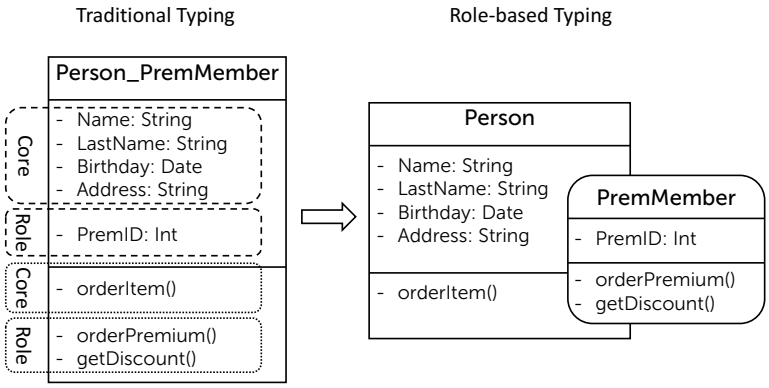


Figure 1.1: All In One Class Definition in Contrast to a Role-based Specification

This split from a single and static metatype to several metatypes with special semantics, enables a separation of concerns on several levels. At first and from a software architecture perspective, the functional behavior is separated from the context-adaptation mechanisms. For instance, assume the order functionality on such an online retailer that varies depending on your user status. The order process does not have to distinguish between a regular or premium user, because this is encapsulated in roles and the method dispatch decides which method to call. Secondly, the use of roles relaxes the situation of the statically described behavior and structure in the types. As role types may be dynamically added to or removed from the system, the entities can vary their available set of behavior and structure. Thus, entity types can evolve over time without recompiling the whole type. As example, imagine a premium functionality launch that defines new behavior during the ordering process. This new behavior is defined in a separate role type which is dynamically added to the system. Thirdly, the actual behavior and structure of an entity can be varied after instantiation by start or stop playing roles in certain contexts. For instance, a regular user signs up for a premium membership and as soon the payment is done, the new functionality is available. This is solved by adding a new role to the user core entity element, giving it additional functionality and structure without entity reinstantiation. Moreover, this enables different lifetimes of various parts of an entity. As soon as the premium membership ends, the additional structure and behavior is gone, but the entity itself survives.

The concept of roles is well-known in the modeling and programming language communities. For instance, the Object Role Modeling (ORM) [34], the Information Networking Model [61], or the HELENA approach presented in [37] are representatives for modeling languages that utilize roles as first class citizen. On the programming language side, there exist ObjectTeams [39], SCROLL [59], or Rumer [9].

From Adaptive Software to Database Systems

Database systems are an integral component of today's software systems, because they provide standard data management functionalities that are used by the applications. Thus, database systems provide several guarantees on which the applications can rely without taking care of these functionalities by themselves [54]. First, it encapsulates the persistent data management from the applications.

Moreover, as central data storage facility, it ensures global data consistency for the whole application landscape, which gives it the characteristic as single point of truth in a software system. From this point of view, database systems are not mainly concerned with the actual data, because storing data in a certain format is a simple job, but with the schemata and keeping the data consistent with respect to the schema. Hence, database systems care more about the schema and not the actual data. Additionally, database systems ensure an efficient data storage and access within transactions. Furthermore, they have to take care of data recovery and concurrent data access. Finally, database systems provide an expressive query language.

Even in a ubiquitous software world these guarantees have to be ensured by a database system. In detail, we assume such software world to be modeled and implemented by using roles as extension points to entities. Moreover, a role-based metamodel is much more complex, because it encompasses not only a single metatype, in fact there are several metatypes that are interrelated to each other by special semantics and constraints. Unfortunately, there is no database system available that is able to explicitly model roles as extension points of entities in certain contexts, so far. Generally, there are two ways of coping with the absence of role semantics in the database system: (i) hiding this absence as much as possible by implementing mapping engines or (ii) implement a database system that is aware of the role semantics.

The first option does not lift the database model on the semantics of the applications metamodel. Rather, mapping engines are used to hide this semantic gap from the applications and their developers. This solves the semantic gap only partially and from an application perspective. In detail, mapping engines pull data management tasks out of the database system and realize them closer to the applications. Especially, the metamodel constraints, in our case the role-based metatype distinction, their interrelations, and constraints, are managed in the mapping engine, because these constraints are not directly representable in the database. Hence, the mapping engines or applications are in charge to enforce the metamodel and its related constraints. This becomes a problem in case of multi-application scenarios with multiple mapping engines. Each of the mapping engines as well as the applications rely on their local view on the data and their own mapping of the metamodel, hence, the consistency managed in them can only be guaranteed for this local perspective. Moreover, all mapping engines have to be aligned in their mapping processes to be compatible to each other and use the same data other engines write to the database. As a consequence of the metamodel mapping in the mapping engines, the database system loses its characteristics as single point of truth in the software system. Worse, it is not used to take care of the schema, rather as storage engine for a certain storage format. Moreover, is does not encapsulate the data management process, because the applications are required to implement a separate piece of software that maps the semantics. In fact, the persistent data management is distributed over several layers in the software system.

The second option implements the role-based semantics in a database system as a first class citizen, so the semantics can be directly used, without a mapping involved, by the applications. This lifts the database system's metamodel on the same semantic level as the application's one already is. Such an implementation includes not only the metatype distinction, but the metatype interrelations and constraint. Hence, the applications are not in charge to enforce the metamodel in the mapping process, in fact it is ensured in the database system. As the role-based semantics are directly represented in a database system, it is capable to ensures global consistency with respect to the metamodel and without the need to enforce these constraints outside the database system. The mismatch caused by the semantic gap between different, especially role-based, semantics can be overcome by this solution. In this thesis we investigate the options to introduce a novel logical database model that is capable to directly represent a role-based metamodel distinction and ensures the metamodel consistency constraints as inherent database feature.

Introducing Novel Semantics in a Database System

To adapt a database system, we firstly have to look at the architecture of common database systems. These are clearly structured in their abstraction layers. In detail, traditional database systems feature five layers of these as depicted in Figure 1.2 [71, pp. 37–40]. Topmost, the set-oriented interface represents the query language and represents a set-oriented access to the database system. In case of a relational database system, this is ensured by implementing the SQL standard. This interface accesses the data system, which is probably the most complex layer. It ensures the translation between the set-oriented and record-oriented interface by algebraically optimizing the queries, choosing the best access paths on the basis of statistics, and ensuring the integrity. Moreover, the logical processing model is implemented in this layer. In relational database system, the relational algebra, the optimizer, as well as integrity checks are implemented in the data system. Thus, the operator plan is created, optimized, and joins are performed there. The record-oriented interface represents the navigational access on the data access system, like indexes or table scans. The data access system is in charge of the transformation on the internal record structure by providing efficient data access structures. However, as it can be seen by these two example layers, the semantic richness decreases from one layer down to another one. From relations and tables on the set-oriented interface to byte streams that are stored as files on the file system.

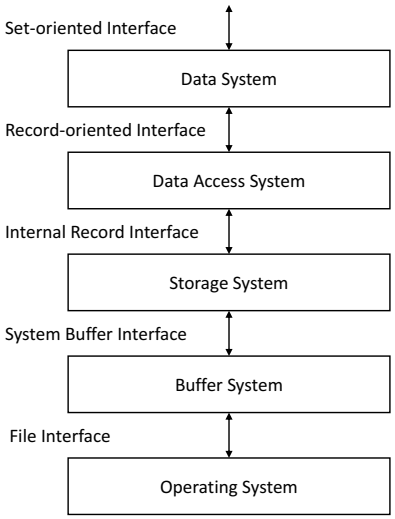


Figure 1.2: 5 Layers of a Database System Architecture; According to [71]

Consequently, introducing a novel database model with additional role-based metamodel semantics, requires to adapt a database system's data system as well as the set-oriented interface first. From the data system perspective, it is required to introduce data structures that feature role-based semantics. Additionally, novel logical operators are needed to enable a processing of these structures. Moreover, the corresponding processing model for role-based data structures must translate them into external records and access paths to achieve a connection to the data access layer. The set-oriented interface is in the need to reflect the data structures in a language interface and in the query results. Precisely, it is required to adapt the query language by introducing role-based semantics in its statements. Additionally, the query results must reflect the implemented metamodel semantics as well and expose them to the clients.

Thesis Contributions

In this thesis we argue for an integration of roles and their related semantics in a database system, to overcome the semantic gap between role-based software and traditional non-role-based database system. Thus, we propose an adaptation of the data system layer to a novel database model and an adjusted processing model in combination with a redesigned set-oriented interface. In sum, these adaptations are combined in the RSQL approach, which is the key contribution of this thesis. In detail, this thesis provides the following contributions.

(1) At first, we investigate the problems of traditional, especially relational, database systems in combination with role-based software systems in detail. These problems are concentrated under the term **role-relational impedance mismatch**, which describes the problems from a database system's, application's and application developers', and software system's perspective. On this basis, we define several **requirements to overcome this role-relational impedance mismatch** and evaluate them against four possible architecture layout solutions. These layouts range from using relational techniques only, over mapping engines, to a full database integration. As Figure 1.3 shows, these contributions are the main concern of Chapter 3.

(2) To meet these requirements, we introduce a role-based database model as logical foundation for an adapted data system layer. This database model is called **RSQL database model** and defines logical data structures that encapsulate the role-based semantics. In detail, we define **Dynamic Data Types** as data structure on the schema level and **Dynamic Tuples** as instance representation. These definitions feature roles as first class citizen and embed them into contexts. Moreover, we specify various **logical operators** that enable the processing of Dynamic Tuples while preserving the role-based semantics. These contributions build the main body of Chapter 4.

(3) To propose a solution for an adaptation of the set-oriented interface with respect to role-based semantics, we specify the **RSQL query language** and the **RSQL Result Net**. In detail, the query language definitions feature formal syntax descriptions for the data definition, data manipulation, and data query language. The first creates a role-based database schema, the second populates the corresponding database with role-based data, and the latter retrieves data from the database. Moreover, we connect the data query language elements with the logical database model operators to generate logical operator plans. These plans build the basis for a **query processing** of Dynamic Tuples. Moreover, we propose a role-based result representation, named RSQL Result Net, which preserves the role-based semantics in query results by returning multiple sets of Dynamic Tuples to the clients. In addition, we specify several functionalities to navigate and iterate through the result, on the basis of Dynamic Tuples. As Figure 1.3 illustrates, Chapter 5 includes these contributions.

(4) To **demonstrate the benefits** of the logical database model, the query language specification, and the result representation, we evaluate them in contrast to a relation representation in Chapter 6. As the database model evaluation shows, the proposed RSQL database model requires up to 15 times less statements to set up a role-based database schema. The query language evaluation demonstrates shorter RSQL queries that additionally do not mix mapping details with entity information. To evaluate the result representation, we compared the processing effort of an RSQL Result Net based result in contrast to two relational result representations of role-based data. This shows, that the RSQL Result Net combines the positive aspects of both relational representations, in writing lines of code, transferred data, and size of the result.

Thesis Structure

The remainder of this thesis is structured as visualized in Figure 1.3. The following Chapter2 introduces the concept of roles as modeling primitive and provides an overview of various role notions. As the perception of a role is very diverse, a classification of different role notions is provided on the basis of the surveys published by Steimann [79] and Kühn et al. [58]. Moreover, this chapter introduces a university domain that is utilized as running example throughout the thesis.

As the roles are accepted in modeling and programming languages, but not as data model in database systems, Chapter 3 discusses the problems arising out of the role absence in the database system under the term role-relational impedance mismatch. To overcome this mismatch, several requirements for a database integration are defined. Moreover, related approaches are detailed, classified by their targeted software system architecture, and rated by their capability to overcome the mismatch.

On these foundations and the conclusion that no related approach fully integrates the notion of roles in a database system, the RSQL approach is introduced. At first, Chapter 4 defines the database model on the schema and instance level by establishing the Dynamic Data Types and Dynamic Tuples along with Relationships and formal operators. Secondly, the query language consisting of a data definition language, data manipulation language, and a data query language is discussed in Chapter 5. Moreover, this chapter specifies the query processing and RSQL's result representation, the RSQL Result Net.

To evaluate RSQL, it is compared to a relational representation of itself in Chapter 6. In detail, this evaluation comprises a database model comparison with respect to instance consistency, a comparison of the RSQL's query language with SQL, and an assessment of the client side costs to process the query results. Finally, this thesis provides a conclusion, and a perspective on future works in Chapter 7.

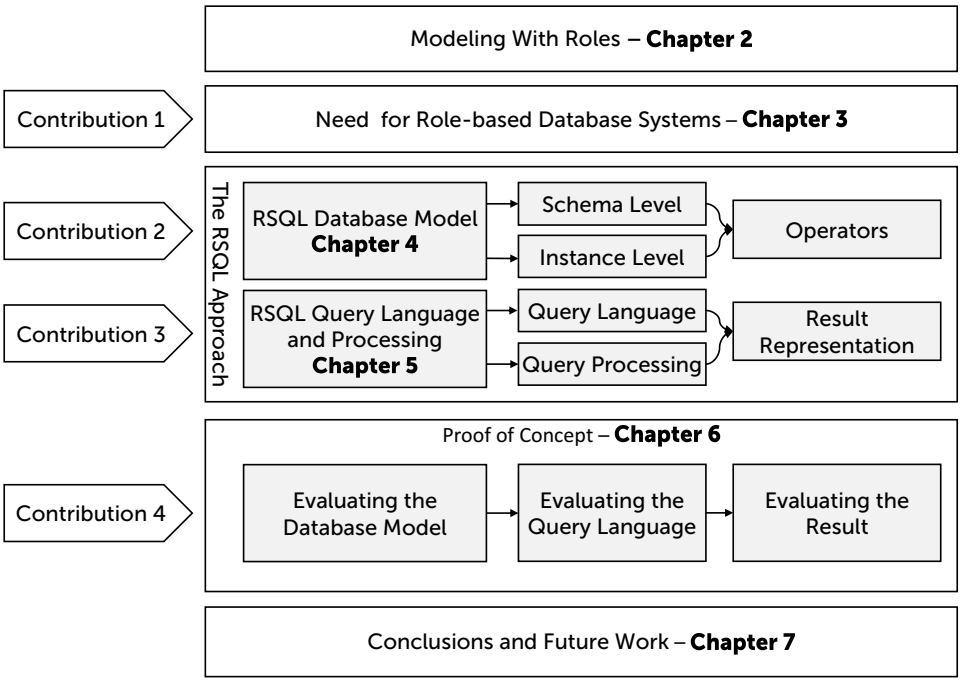


Figure 1.3: Thesis Structure and Contributions

MODELING WITH ROLES

Roles as modeling primitives are nothing novel, in fact this concept has been introduced by Bachman and Daya in 1977 [8] as extension to the network data model. However, roles provide advantages compared to traditional modeling languages, like the Unified Modeling Language (UML), and programming languages, like Java; structure and behavior can be added to an existing object during runtime. Hence, we firstly outline design time and runtime issues of widely used traditional modeling approaches to make our way toward the benefits of roles and a detailed discussion of these. Over the recent decades, a lot of role-based approaches have been proposed, all having a different notion of roles. This zoo of role notions has been surveyed by Steimann as well as Kühn et al. resulting in 26 different features conceded to roles. Moreover, a classification of this thesis within that particular zoo is presented. To rely on a sophisticated role notion as base for this thesis, the Compartment Role Object Model (CROM) is introduced and explained in detail. Furthermore, a university management scenario is introduced, presenting the advantages of roles as well as running example throughout this thesis. Finally, the chapter is concluded in a short summary.

2.1 WAY TOWARDS ROLES

Generally, a model in computer science is an abstracted and simplified image of the real world for a given purpose. Hence, uninteresting parts of the captured real world are omitted, objects are abstracted to classes or types, and complex relations are boiled down to their essential parts with respect to the given purpose. In computer science models are used for various purposes, for instance during the analysis or design phase in software development. In general, Steimann identifies two main goals for modeling in computer science [78]. At first, capturing real situations of a corresponding domain and for communication purposes during the analysis phase. Secondly, a model can represent the design for a software system, which describes the elements of such a system in detail. This may also be a guideline for the implementation of a piece of software. However, traditional and widely used mature modeling methodologies, like the Unified Modeling Language [70] or Entity-Relationship Modeling (ERM) [18], are based on the fundamental concept of entities and relationships between these entities. Moreover, these traditional modeling languages consider each entity to be an instance of a fixed certain type (static typing), having a defined structure (the entity's attributes) and a given behavior (the entity's methods). Unfortunately, changing an entity's type is not covered by those modeling languages and results in circuitous instance replacing, whereas the object's identity is typically lost. This static representation results in problems for modern software systems that act in a large variety of environments and contexts.

While designing, and modeling a software system using these traditional modeling methodologies, designers can run into a lot of problems, like fat types, subtype explosion, or redundant subtypes. At first, a modeler needs to integrate every aspect of an entity in its type during design time, even if some attributes or methods are of interest only in some situations for few entities. Such fat types give every instance the full behavior, even when the instance is not supposed to have it. For example, imagine a student management system and along with the students there exist student representatives and student assistants. Including every behavior of the student representative and assistant into the student type is unintended, because every student would get the methods and attributes of a student representative and assistant as well. To avoid such problems, designer may use inheritance for generalization and specialization of types. This leads to the second problem occurring during design time, subtyping does not scale and results in numerous subtypes. For example, a student may be a student representative and student assistant at the same time, i.e. the modeler needs to design each combination of subtypes as subtype as well. In the example the subtype of student assistant and student representative is required. As you can imagine, introducing another subtype, like a student tutor, results in a combinatorial explosion of subtypes. You may accept to spend this effort once, but maintaining and

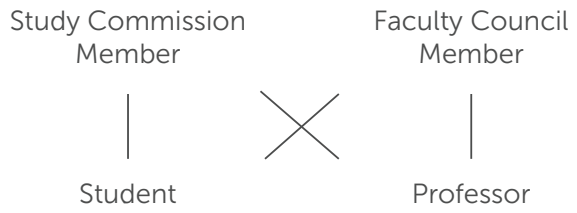


Figure 2.1: Specialization and Generalization Problem in Object-oriented Design; According to [79].

extending such systems becomes impracticable. Finally, designers may run into problems they cannot decide, whether it is a specialization or generalization. For instance, envision a study commission and faculty council as part of the university, where both, professors and students may be members of those. Figure 2.1 illustrates this situation. On the one hand, you can model both member types as subtypes resulting in multiple inheritance, one inheritance from student and one from professor. This would mean that each member instance is a student and professor at the same time, which is not what you want to express. On the other hand, you may model both member types as supertypes, whereas professor and student are subtypes of those. This is unintended, too, because each professor would be a member of the study commission and faculty council. To avoid such undecidable situation, modelers may use patterns as workaround [23]. Such a pattern is the introduction of separate subtypes of both, study commission member for student and professor as well as for faculty council. Applying this pattern, you will end up with two different subtypes expressing the same. Additionally, you have to implement the structure and behavior multiple times resulting in overhead for maintenance. On top of the mentioned problems, there exist several other ones, designers can run into. However, the described issues underline the need for more flexibility and dynamics to improve modeling languages.

During runtime, the static representation of entities in UML or ERM causes problems, too. Over the whole lifetime of an entity these sets might change and usually do, so that some attributes or methods are only valid for a certain period (i.e., being a student), other may come once and stay the whole lifetime (parenthood). Additionally, entities of the same type may have different sets of attributes and behavior at the same time. Most of the problems of static typing is caused by the entity's immutable set of attributes and methods, because it must be part of a certain type for its whole lifetime. One workaround is recreating an entity within a different type, but at runtime, an entity is usually identified by a certain system internal ID. Thus, reinstantiation may provide an entity having the same attribute values, but with a different internal ID. The new entity may look alike, but internally it is a different one, thus, it is handled as a different one. Additionally, reinstantiation is only a workaround on the instance level for missing flexibility and dynamic typing. Another problem occurs when an entity is conceptually separated into multiple individual (sub)entities, for instance the student itself and the student representative are two separate entities related by an is-a association. The adaption's semantics intended to be expressed in this scenario, that is to say the student representative and the student is the same entity, are implied by the programmer, but not an integral part of the program itself. This complicates code maintenance as well as vanishes the originally intended semantics. Additionally, the runtime will handle both entities separately which may lead to object schizophrenia [40, 74]. That phenomenon describes the problem that occurs when conceptual entities are separated into several parts and each part forms an individual physical entity. Consequently, caused by the missing semantics in the code interpretation, each physically separated entity sees itself as the whole entity, although it is only a part of a greater whole. Of course, some runtime problems may be worked around using patterns, but the original problem of static entities and their misinterpretation remains.

In summary, modern software systems become more complex and act in various continuously changing environments resulting in problems during design time and runtime. Traditional modeling methodologies and paradigms, like the UML, frequently fail when faced with the requirements of such highly dynamic and complex software systems, because they assume entities to be static with

respect to their type and on the meta level those approaches provide only a single metatype to describe entity types. With respect to the two goals of modeling, capturing the real world and designing a software, the level of abstraction of those modeling methodologies is too high, which results in inappropriate models for today's requirements. Moreover and in contrast to this abstraction of static objects, real-world entities are dynamic and evolve over time.

To cope with these outlined problems and address the dynamics, flexibility, and evolution, researchers have developed a lot of approaches, like aspect-orientation [15] or context-oriented programming [41], during the past decades. Among these approaches, there exist the concept of *roles*, which is the focus of the research training group RoSI [30] as well as this thesis. The main idea behind the concept of roles is the *separation of concerns* within an entity, because several parts of an entity have different lifetimes and may only exist during a certain period of the entity's lifetime. This idea of separation of concerns in general is well-known and applied in a lot of areas in computer science. Layering, for example, is a sort of separation of concerns and helps to manage complexity within a software system. For instance, the ANSI/SPARC Architecture, which layers a database management system into three parts: (i) the external level defining the user's views on the data, (ii) the conceptual level defining logical data structures, and (iii) the internal level describing the physical data structures [52, 55]. However, the separation of concerns within an entity means, that the dynamic and flexible parts of an object are modeled separately from the entity core. Consequently, the situations or context an entity may be involved in, are designed in a different place. Special semantics combine the core entity and its extensions to enable context-dependent behavior and structure adaption. Thus, an entity's possibilities to evolve over time becomes an integral part of the modeling methodology and is explicitly modeled.

Generally, a role as concept or modeling primitive in our notion is the context-dependent and dynamic part of an object capturing situation-specific behavior and structure in a separate (meta)type. In contrast to traditional static type description and their workarounds for enabling entity adaption and evolution, roles **explicitly** model behavior adaption possibilities as first-class primitive. For instance, let student be a role of a person in a university. At design time, the student is designed separately from the person and related by *can play* semantics, giving a person entity at runtime the ability to extend its core structure by the student role structure and behavior. For instance, imagine the person as core type and student as well as club member as roles playable by this person. At first, there exists the person core entity only. When joining a university, he or she starts playing the student role and gains new structure, a student ID for instance, and new behavior like enrolling for a class. Later, the same person becomes a member of a sports club, thus, starts playing the role club member in addition to the student role. Hence, the entity evolves a second time without changing the type at all, but by starting or stopping playing roles. In summary, this helps when modeling and implementing complex and context-dependent behavior of entities.

2.2 ZOO OF ROLE NOTIONS

The term role and the underlying concepts are omnipresent in our every day's life, in our communication and speech as well as in the way we think. Thus, it is not surprising that the term is widely used in various research areas, too. For example, in sociology, linguistics, and computer science, of course. However, the origins of this term may go back to the ancient world, especially to the ancient theater. In theater, a role refers to a character or figure an actor or an actress plays in a certain play [8]. This captures the core notion of what a role is very well. Firstly, a role is something abstract someone or something can act like. Secondly, the role itself is defined independently of its player, because the role does not define who or what is going to play that corresponding role. Finally, the role requires a player to be filled with life, because there must be someone who adopts the role's behavior and structure.

This idea has been adopted in computer science as well. One way to use roles is role-based user access, which is mainly used to restrict access to any kind of information [73]. Each operating system as well as database system has some kind of role-based access implemented to restrict access to files and tables, respectively [55]. In database systems, for example, administrator roles have different privileges than simple users; an administrator can create tables, indexes, and promote access to tables. However, this is a special role notion from a human-computer interaction perspective. The general role perception considered in this thesis differs from that one; we utilize roles to enable entities managed in a software system to dynamically gain and lose behavior as well as structure during runtime. This particular role perception has been utilized a lot, for instance in modeling [34, 8, 83, 61, 79, 37, 20] or programming languages [39, 10, 60, 29, 68, 53]. One of the first researchers who introduced roles as modeling primitive into computer science is Charles W. Bachman [7, 8, 80]. Back in the late 1960s and early 1970s, two novel and competing data models for database system have been proposed, the network (CODASYL) model [81] and the relational model [19]. Bachman had one fundamental observation on entities in the network model: entities interact with each other by using roles [7]. Unfortunately, the original network model definition cannot represent roles natively, rather they are represented by additional records. Consequently, Bachman and Daya developed a data model that is able to represent roles as an extension to the network data model [8]. In detail, they extend the record construct by role-segments, which can be used for additional record characterization and specialization. This data model never became popular in the database research community, rather the relational data model has become dominant[1]. However, the relational data model does not provide the desired flexibility and dynamics of entities to overcome the aforementioned issues. For a detailed discussion on various implemented data models, including the relational data model, of database management systems and their shortcomings in the environment of a role-based software system, we refer to Chapter 3. In contrast to the static tuples in the relational data model, roles applied as separation of concerns within an entity can provide these flexibilities and dynamics.

Steimann's Role Features

The original idea of roles as primitives to dynamically extend and shrink entities is still alive and has attracted attention in the research community from time to time. As mentioned before, there exist many approaches for modeling and programming languages that utilize roles, each with its own interpretation of roles. Thus, there is no common notion of what a role is. In fact, there is a large variety of different, sometimes even contradicting, role notions. For instance, ERM. [18] as well as the UML [70] consider roles only as named places in relations and associations, respectively. There are neither role-specific attributes nor behavior in such approaches. In contrast, Bachman's and Daya's role data model utilizes roles as extension to entities, having a separate structure. Furthermore, programming languages like ObjectTeams [39] or the Scala Roles Language (SCROLL) [60] use roles to dynamically adapt the behavior of objects at runtime. However, to provide a universal formal role modeling language, Steimann surveyed and categorized many proposed approaches until the year 2000 [79, 78]. He analyzes the advantages and shortcomings of these approaches. Based on this knowledge, he identifies 15 different features that are related to the term role and three ways to represent them. Moreover, Steimann himself points out that some of them are contradicting, hence, no approach will ever serve all mentioned features. Finally, he develops a formal role modeling language called Lodwick. An overview of Steimann's 15 features of roles and the corresponding description is listed in Table 2.1. In his survey, Steimann uses the term role for both, types and instances, which may be confusing. To clarify which level is addressed by a certain feature, the corresponding Meta Object Family (MOF) level is shown in the right-hand column, which is extracted from Kühn et al. [58].

[1] Of course, there exists other implemented data models for database management systems, especially the NoSQL approaches like key-value stores [21], graph databases [5], and wide-column stores [16]; but the relational data model is still the dominating one.

Feature	Description	Level
1	A role comes with its own properties and behavior	M0, M1
2	Roles depend on relationships	M0, M1
3	An object may play different roles simultaneously	M0, M1
4	An object may play the same role several times simultaneously	M0
5	An object may acquire and abandon roles dynamically	M0
6	The sequence in which roles may be acquired and relinquished can be subject to restrictions	M0, M1
7	Objects of unrelated types can play the same role	M1
8	Roles can play Roles	M0, M1
9	A role can be transferred from one object to another	M0
10	The state of an object can be role-specific	M0
11	Features of an object can be role-specific	M1
12	Roles restrict access	M0
13	Different roles may share structure and behavior	M1
14	An object and its roles share identity	M0
15	An object and its role have different identities	M0

Table 2.1: Overview on Steimann's Role Features and Affected Meta Object Facility's Layers; Extracted from [79, 58]

Classification of Role Approaches

In a second survey, Kühn et al. look into role-based modeling and programming approaches between the years 2000 and 2014 [58] to update Steimann's survey and research as well as to find new trends in role modeling. Moreover, and importantly, they classify the surveyed approaches by three different aspects[2] roles try to serve. Additionally, this classification emphasizes the broad variety of different role notions and a missing common one. An illustration of this classification is presented in Figure 2.2.

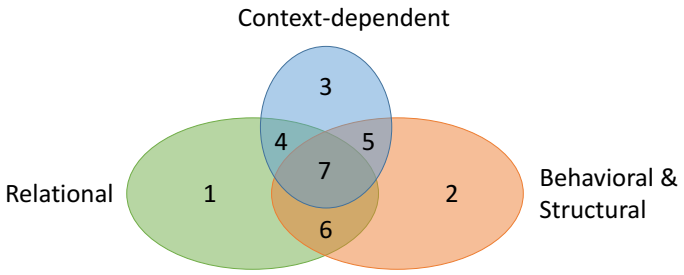


Figure 2.2: Classification of Existing Role Notions Based on the Three Aspects: Relational, Behavioral and Structural, and Context-dependent.

Each trend is depicted as separate ellipse having intersections to the other trends. At first, role approaches can focus on the relational perspective, which means different entities interact with each other or are connected by using roles. Secondly, roles address structural and behavioral aspects of entities, which refers to the flexible set of attributes as well as methods. In fact, Kühn et al. mentions the behavioral aspect only, but the behavioral and structural perspective aim for the same goal, changing the fixed set of methods and attributes, respectively. Finally, roles are utilized to describe context-dependent features of entities. Based on these three perspectives, seven classes of approaches are possible, which are indicated and referenced by the corresponding number in Figure 2.2.

[2]You may also call it perspective on roles, purpose roles are used for, or trends in the notion of roles.

1. These languages focus on the relational aspects of roles only. They are used to connecting entities by roles, may it be in relationships or as adjunct objects. Representatives of this area are for instance, the UML [70] or the programming language Rumer [10, 9]. Both approaches feature roles as named places within relations; UML in associations and Rumer in relationships.

2. There exist role languages that focus on adaption of behavior and structure of an entity only. For example, the generic role model presented in [20], the programming languages Chameleon [28] and Rava [36].

3. So far, there are no approaches that focus on the context-dependent nature of roles only.

4. Approaches like the HELENA approach [37] combine the relational and context-dependent nature of roles. Roles are utilized to express an entity's interaction with other entities. Additionally, this interaction is embedded into a context. Behavioral and structural adaption of an entity are not considered in this class of approaches.

5. Contextual role languages mainly focus on the adaption of behavior and structure with respect to a certain context or situation. Aspect-oriented and context-oriented programming languages are situated in this class. Representatives are ObjectTeams [39], the metamodel presented by Genovese [24] or Kamina and Tamai [53].

6. This class of approaches brings the relational as well as the behavioral and structural perspectives on roles together. Object-Role Modeling (ORM) [35], Lodwick [78], and the Information Networking Model (INM) [61] are representatives of this class.

7. The last category combines all three aspects in one approach. So far, the Compartment Role Object Model [57, 56] is the only proposed metamodel combining all these aspects. Moreover, the RSQL approach presented in this thesis belongs to this last category.

So far, the term context has been used frequently without providing a precise definition of this term. The term context is, like the term role, very general and everyone has a different perception of what a context is.

Context versus Compartment

To clearly understand the context-dependency of roles, there are two different notions applicable to this nature. Firstly, context as environmental information and secondly, as objectified collaboration containing other entities.

The first notion is the interpretation given by Abowed et al. [1]. It is more commonly used in computer science and defines context as follows:

> *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.* [1, pp. 306–307]

Consequently, a context is something that surrounds an entity and provides additional information. Information, in the authors' sense, may be time, place, temperature, the user's mood, or even the state of the application a user is running. Additionally, context is considered to be omnipresent, even no information regarding an entity is an information. Moreover, context is always related to a certain entity. For instance, classical context information is: it is a hot and sunny day in a certain area. This is only context information for people who are currently present in this area, but not for persons outside this area. In detail, this information is the air temperature, cloud formations, the date, and a location. In our digital world such information can by inferred easily from sensor data, like GPS or the heat sensor of a smartphone. To sum up, this context definition describes an entity's environmental information that has no specific identity, no intrinsic behavior and is omnipresent.

In contrast to the traditional context perceptions, there exist another notion, namely compartments; a term introduced by Kühn et al. [58]. As motivation for using and introducing this term, they provide the following explanation:

> *[W]ithin modeling languages, context represents a collaboration or container of a fixed, limited scope. To overcome this dichotomy, researchers avoided the term context by using other terms, i.e., Environments, Institutions, Teams and Ensembles. In turn, we use the term Compartment as a generalization of these terms to denote an objectified collaboration with a limited number of participating roles and a fixed scope.* [58, p. 146]

Since this definition is very general, we will detail and explain it. At first, a compartment is a constructed entity itself containing other entities. Secondly, it carries an identity, has its own structure and behavior. Finally, a compartment has a defined lifetime and life-cycle, respectively. For example, imagine a university as entity as well as the roles student and professor as part of the university. The university will be modeled as a compartment, because both roles, the student and professor, are only valid within this university. The university itself has its own identity, for instance the name, its own structure, and its own behavior. Regarding the definition of Kühn et al. the university acts as objectified collaboration for the roles student and professor. Consequently, these roles cannot be used outside of this defined collaboration. Table 2.2 compares both notions and as you can see, compartments are the exact opponents to traditional contexts. Moreover, this thesis is based on this compartment definition.

Characteristic	*Context*	*Compartment*
Nature	environmental information	construction containing entities
Identity	no	yes
Structure and Behavior	no	yes
Lifetime & Life-cycle	no	yes

Table 2.2: Characteristics of Context in Comparison to Compartment

Additional Role Features by Kühn et al.

In addition to the 15 features Steimann identifies, Kühn et al. define eleven additional features, most of them (8) describing the contextual nature of roles with respect to compartments [57, 56]. Especially this context-dependency has been neglected by most of the role approaches before 2000 and even by Steimann himself. An overview of these additional role features is given in Table 2.3 starting with feature 16, because they can be seen as addition to Steimann's features.

Feature	Description	Level
16	Relationships between roles can be constrained	M1
17	There may be constraints between relationships	M1
18	Roles can be grouped and constrained together	M1
19	Roles depend on Compartments	M0, M1
20	Compartments have properties and behavior	M0, M1
21	A role can be part of several compartments	M0, M1
22	Compartments may play roles like objects	M0, M1
23	Compartments may play roles which are part of themselves	M0, M1
24	Compartments can contain other compartments	M0, M1
25	Different compartments may share structure and behavior	M1
26	Compartments have their own identity	M0

Table 2.3: Overview on Kühn's Additional Role Features and the Affected Meta Object Facility's Layers; Extracted from [58]

In the first place, Kühn et al. extend the perception of roles by direct relationship constraints and constraints between relationships (16 and 17). Cardinality constraints, for example, are a representative for the first constraint specification (16) and an implication between two relationships for the second one (17). Additionally, they introduce a grouping of roles and a collective constraining of those (18), which simplifies modeling by constraining the group as its whole instead of defining the constrained for each group member separately. In sum, these three features provide additional options to restrain roles. The other features (19–26) refer to compartments and their properties. The first compartment feature relates roles and compartments to each other (19). In this sense, roles are directly embedded within compartments. For instance, a compartment university and a role student within this university. Additionally, compartments are specified like objects and roles, thus, they come with their own properties and behavior (20) and have their individual identity (26). Hence, a university can have a name and location as properties and a behavior like enrolling students. Furthermore, compartments are able to play roles (21), even in case the role is contained within the compartment itself (22). Imagine a sports team as compartment containing several team members. This certain team can then attend to a tournament as its whole by playing the attendee role for instance. Finally, Kühn et al. provide features for hierarchically structure compartments (24) and inheritance of those (25).

2.3 COMPARTMENT ROLE OBJECT MODEL

This thesis assumes the Compartment Role Object Model (CROM) proposed by Kühn et al. as base notion for roles [56]. CROM itself describes a metamodel for building role-based models and schemata, a Compartment Role Object Instance (CROI) as instance representation, and a Constraint Model to represent several constraints in the model and instance validity verification. To give you a deeper understanding of CROM and the underlying concepts and semantics, it is discussed in detail hereafter. Basically, CROM provides four different metatypes: *Natural Types*, *Compartment Types*, *Role Types*, and *Relationship Types*[3]. To provide developers and designers a clear specification to decide which type to use, an ontological foundation is given. Base of this type distinction are the established ontological properties *Rigidity*, *Foundedness*, and *Identity* [57]. Rigidity and Foundedness as ontological concept have been established by Guarino [31, 32] to distinguish Natural Type and Role Type in

[3]For the purpose of a strict optical and orthographic distinction between CROM metamodel elements (Natural, Role Type, or Role) and conceptual primitives (role, compartment), we refer to the former ones by writing them as proper nouns. In contrast, the latter ones are written in a standard English way.

particular [79]. These two properties suffice to differentiate between Natural Types and Role Types. As soon as Compartment Types and Relationship Types enter the game, an additional property is required. The ontological property of an Identity [64] provides further distinction options [57].

Semantic Rigidity relates to the concept that an instance has to be part of this type for its whole lifetime. The opposite of a rigid type is an anti-rigid type. This rigidity criterion can be applied to a person, for example, because you cannot stop being a person without losing your identity[4]. *Foundedness* describes the fact that a type is existentially dependent from the existence of other types. In fact, founded types cannot exist on their own, which also holds for instances of these types. This applies to Role Types for example, which cannot exist in isolation. Finally, the *Identity* property characterizes whether a type's instance has a unique, derived, or composed identity. For example, each person will be identified by a unique identity.

These three properties allow us to distinguish the four metatype elements of CROM. Firstly, *Natural Types* (NT) are considered to be rigid, non-founded, and have a unique identity. A person, for example, can exist on its own and each person has a unique identity. Additionally, a person cannot change its type. Secondly, *Compartment Types* (CT) are semantically rigid, but found and have a unique identity. For instance, a university has its individual identity and cannot change its type, but depends on the existence of Roles like professor or student. Thirdly, *Role Types* (RT) are the opposite of Natural Types; they are anti-rigid and founded while their identity is derived from their player types. The student Role Type is an example. It can only exist if there is a certain player type (person) and a Compartment Type (university) to exist in. In addition, the student identity is derived from its individual player. Finally, *Relationship Types* (RST) are rigid, founded and have a composed identity. Imagine a student taking a class, the corresponding Relationship Type will be *takes class*. The type of a certain Relationship cannot be changed and each Relationship Type requires at least two Role Types to participate in. Moreover, the Relationship identity is composed by the participating Roles. A summary of the ontological foundation of CROM's metatypes is presented in Table 2.4.

Concept / Property	*Rigidity*	*Foundedness*	*Identity*	*Example*
Natural Types	rigid	non-founded	unique	Person
Compartment Types	rigid	founded	unique	University
Role Types	anti-rigid	founded	derived	Student
Relationship Types	rigid	founded	composed	takes class

Table 2.4: Ontological Foundation of the Metatypes Introduced in the Compartment Role Object Model as defined in [57]

The CROM metamodel provides the aforementioned four different metatypes Natural Type, Compartment Type, Role Type, and Relationship Type, but they do not exist in isolation. In fact, they are connected by relations and functions having special semantics.

fills relation First, the *fills* relation connects player types with Role Types [57]. Basically, this expresses which type can play Role of which Role Type, while a player type can be a Natural Type or Compartment Type or both. Hence, the *fills* relation is not limited to only one player type per Role Type, multiple player types for a single Role Type are explicitly allowed.

parts function Secondly, a *parts* function relates each Role Type to a certain Compartment Type, thus, each Role Type will be located in exactly one Compartment Type [56]. Additionally, each Compartment Type must contain at least one Role Type, hence, this function must return a non-empty set of Role Types for a given Compartment Type. This is required, because an empty Compartment Type would not be founded, like a Natural Type, thus, it should be modeled as one and not as Compartment Type.

[4]Identity in this context refers to an entity's identity and explicitly not to the ontological property *Identity*.

rel function Finally, there exist the *rel* function mapping Relationship Type to distinct Role Types in the same Compartment Type [56]. As a result, Relationships over several Compartment Types are prohibited, which makes sense when you see a Compartment Type as a collaboration or a situation a certain Role is situated in. Why should Roles of different collaborations relate to each other? If they need to do so, they obviously belong to the same collaboration and the designer captured the domain insufficiently or made a mistake. This constraint also helps to clearly define and model where a certain Role Type needs to be located in.

In summary, a valid CROM model is defined as: $\mathcal{M} = (NT, RT, CT, RST, \mathit{fills}, \mathit{parts}, \mathit{rel})$ [56].

Additionally, Kühn et al. propose CROI, the instance representation of a well-formed CROM model [56]. On this level, Natural Types, Compartment Types, and Role Types are instantiated to Naturals (N), Compartments (C), and Roles (R), respectively. To connect and relate the instances, they provide relations and functions, too.

type function Firstly, each instance is related to its type by the polymorphic *type* function. This function basically returns for each instance the corresponding type. For instance, the object *John* would return **person** as type.

plays relation Secondly, the *plays* relation connects players, Roles and Compartments. In contrast to the type level, a single player is mandatory on the instance level. This is due to the different semantics on both levels. On the model level, *fills* represents **can play** semantics whereas *plays* on the instance level denotes an active playing of a certain Role instance. In addition, each Role Type can be played only once per Compartment. Hence, each Role is uniquely identifiable by its player, the corresponding Compartment, and the Role Type. Of course, one can imagine playing a Role of the same Role Type multiple times simultaneously, which is not prohibited by CROM, but it is not allowed within the same Compartment. For example, a **person** may be a **student** multiple times, but in different **universities**. Thus, the university is the differentiation criterion in such a situation.

links function Finally, the *links* function stores information about Roles participating in Relationships. It is the instance representation of the Relationship Types and returns all related Roles as tuple for a certain given input Relationship Type.

In sum, a CROI is defined as: $\mathfrak{i} = (N, R, C, \mathit{type}, \mathit{plays}, \mathit{links})$ [56].

Beside the model and instance definitions, Kühn et al. present a *Constraint Model* to apply several constraints to Roles and Relationships as well as role groups [56, 57]. First, they define various intra-Relationship constraints like being irreflexive. Secondly, Kühn et al. propose role groups featuring role group constraints that are applied to the whole group instead of each individual Role. Moreover, role groups can have cardinality constraints. For instance, imagine the Role Types principal investigator, post-doc, and research assistant, all being mutually exclusive. Instead of applying prohibition constraint on each Role Type, which can be very messy due to the squarish growth in prohibitions, you just group the Role Types together and apply a cardinality constraint that at most one of the corresponding Role Types can be played, on the whole group.

However, CROM as role-based and context-sensitive formal model can be related and categorized by the 26 features Steimann [79] and Kühn et al. [56] propose. A detailed classification of CROM is presented in Table 2.5. Generally, there are four different options for each feature: *yes* indicating the feature is fulfilled, *no* representing the feature is not fulfilled, *possible* referring to a feature that can be easily added, and *not applicable* if the feature cannot be applied to the approach at all. In general,

Feature	Description	CROM
1	A role comes with its own properties and behavior	yes
2	Roles depend on Relationships	yes
3	An object may play different roles simultaneously	yes
4	An object may play the same role several times simultaneously	yes
5	An object may acquire and abandon roles dynamically	not applicable
6	The sequence in which roles may be acquired and relinquished can be subject to restrictions	yes
7	Objects of unrelated types can play the same role	yes
8	Roles can play Roles	no
9	A role can be transferred from one object to another	not applicable
10	The state of an object can be role-specific	yes
11	Features of an object can be role-specific	yes
12	Roles restrict access	not applicable
13	Different roles may share structure and behavior	no
14	An object and its roles share identity	yes
15	An object and its role have different identities	yes
16	Relationships between roles can be constrained	yes
17	There may be constraints between relationships	no
18	Roles can be grouped and constrained together	yes
19	Roles depend on Compartments	yes
20	Compartments have properties and behavior	yes
21	A role can be part of several compartments	no
22	Compartments may play roles like objects	yes
23	Compartments may play roles which are part of themselves	yes
24	Compartments can contain other compartments	possible
25	Different compartments may share structure and behavior	no
26	Compartments have their own identity	yes

Table 2.5: Compartment Role Object Model's Feature List; Extracted from [56]

CROM fulfills 17 out of 26 features at which three features are not applicable to the Compartment Role Object Model. All these not applicable features refer to dynamics on the instance level that are correlated to time. CROM only represents a model (schema) and an instance that is a snapshot of a valid system state. There are no options to manipulate a CROI at all, because it is designed to represent a snapshot. By the way, this circumstance is common to all modeling languages representing snapshots and do not provide semantics for system changes. In summary, CROM and CROI build a powerful and, most importantly, formally defined base to build on.

2.4 UNIVERSITY MANAGEMENT SCENARIO

To demonstrate the advantages and superiority of role-based modeling, especially CROM, compared to traditional static typed modeling languages a running example is introduced. As general scenario, a university management system is employed in combination with the sports team management. This scenario is going to be used throughout the entire thesis and will be detailed and punctually extended when required. At first, we define the scenario's schema, which serves both general model goals; capturing the real world and being a design template for a real software system. In our case, the

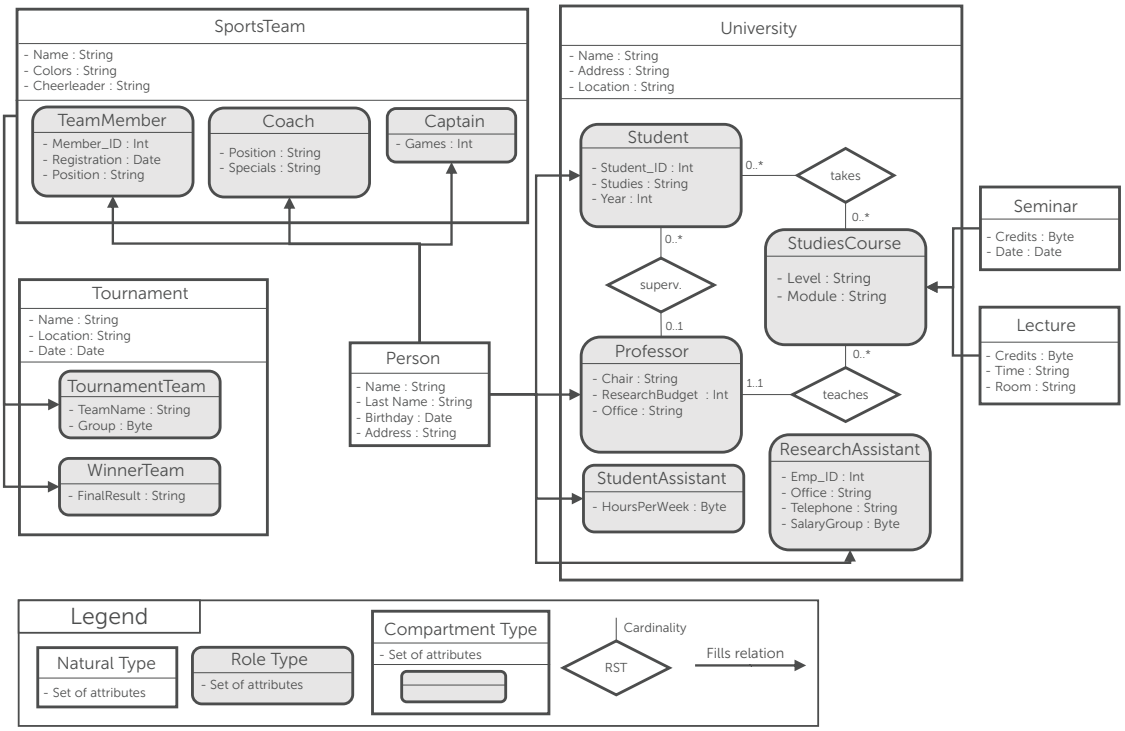


Figure 2.3: Role Modeling Example of the University Domain

database back-end will be the main application target. Secondly, a valid instance of the schema is detailed. Because the instance model can only provide a snapshot, we finally outline a small use case that captures the adaption of an instance over a certain period. Note, we are focusing on the entity's structure rather than on the behavior, thus, method heads are neglected throughout the scenario description.

Generally, a university manages students in general, students taking classes that are taught by professors, and professors supervising students. A class can either be a lecture or a seminar. Additionally, we consider two types of assistants as part of this university scenario. Beside the traditional university management, our scenario also encompasses a university sports teams that consist of members, a coach, and a team captain. These teams as conceptual unit can participate in several tournaments. Each tournament will also declare its winner team.

This university scenario is modeled based on the graphical CROM notation presented by Kühn et al. in [57]. As you can see in the legend of Figure 2.3, Natural Types are shaped as rectangles and do not hold any Role Types. In contrast, Compartment Types are shaped the same way, but they hold Role Types in their inner space. A Role Type is depicted as a rectangle with round corners and a light gray shade. Additionally, Relationship Types are represented in an ER-like manner as rhombus, but connecting two Role Types [18]. Finally, the *fills* relation is illustrated as a directed arrow pointing from a player type (either a Natural Type or Compartment Type) to the corresponding Role Type.

2.4.1 Schema Model

Figure 2.3 illustrates a possible role-based model of this scenario. The core of this scenario is a **Person**, who acts in two different environments[5]. Hence, the **Person** is modeled as Natural Type. The environments he or she is acting in are **University** and **SportsTeam**. Both are modeled as Compartment

[5]All types shown in Figure 2.3 are represented in **bold** type throughout the thesis.

Type, because they act as objectified context and there exist Role Types that can be played in this certain context. First, in the **University** Compartment Type the **Person** can play the Role Types **Student**, **StudentAssistant**, **ResearchAssistant**, and **Professor**. Additionally, the **University** features a **StudiesCourse** Role Type, playable by the Natural Types **Seminar** and **Lecture**. Moreover, the **University** includes the Relationship Types **takes** to manage which **Student** is taking which **StudiesCourse**, the Relationship Type **teaches** to model which **Professor** gives which **StudiesCourse**, and finally the **supervises** Relationship Type to hold the information which **Professors** supervises which **Student**. These Relationship Types all have cardinality constraints. A **Student** can take a **StudiesCourse** but he or she does not have to (0..*) and the other way around, a **StudiesCourse** may have **Students** attending (0..*). A **StudiesCourse** has to have a **Professor** who **teaches** (1..1), but not every **Professor** needs a **StudiesCourse** (0..*). Finally, a **Student** may be **supervised** by exactly one **Professor** (0..1), but not all **Professors** supervise a **Student** (0..*). The other Compartment Type a **Person** can act in is the **SportsTeam** consisting of a **TeamMember**, a **Coach**, and a **Captain**. This **SportsTeam** as its whole can participate in **Tournaments** filling the Role Type **TournamentTeam**. Additionally, in case the **SportsTeam** wins, it will also act as **WinnerTeam** in the **Tournament** Compartment Type.

2.4.2 Instance Model

On the instance level Natural Types are instantiated to Naturals, Compartment Types to compartments, Role Types to Roles, and Relationship Types to Relationships, respectively. As the graphical notation in Figure 2.4 indicates, the shapes of types and instances are almost the same, only differing in the illustration of Relationships. On the one hand, natural instances are illustrated as rectangles without any Roles inside, whereas on the other hand compartments are shaped the same, but featuring Roles. Roles are depicted as rectangles having rounded corners. Relationships are represented by lines between the corresponding Roles and a label indicating the type. Finally, a directed arrow represents the *plays* relation between a player instances and the played Role.

Figure 2.4 illustrates a possible valid instance of the university example shown in Figure 2.3. Generally, the instance model consists of five natural instances[6] of the Natural Type **Person**. These are *John*, *Max*, *Kai*, *Gert*, and *Tim*. All of them are acting in various Roles in two different compartments: *TUD* of the Compartment Type **University** and *bears* of the type **SportsTeam**. In detail, *John* is a **Student** *s1*, and **StudentAssistant** *sa1* in the *TUD* compartment. *Max* and *Tim* act as **Student** as well, *Max* as Role *s2* and *Tim* as *s3*. Furthermore, *Gert* is a **Professor** *p1*, and *Kai* a **ResearchAssistant** *ra1*. **StudiesCourses** Roles are also present in the example, the **Seminar** *sem1* is the **StudiesCourse** *sc1* whereas the **Lecture** *l1* acts as **StudiesCourse** *sc2*. Additionally, *TUD* Compartment features some Relationships. In particular, the **Student** Role *s1*, and *s2 take* the **StudiesCourse** *sc1* which is *taught* by the **Professor** *p1*. Moreover, *p1 teaches* **StudiesCourse** *sc2* and *supervises* **Student** *s1*.

Adjacent to the Compartment *TUD* there exist the **SportsTeam** *bears* featuring two **TeamMember** (*tm1* and *tm2*) Roles. *John* plays *tm1* whereas *Max* acts in the Role *tm2*. Additionally, *John* is the **Captain** *cap1* and *Max* the **Coach** *c1*. Finally, there is a *cc16* **Tournament** that features the Roles *tt1* and *w1*, both played by the **SportsTeam** *bears*[7].

[6] All instance elements shown in Figure 2.4 are represented in *italic* type throughout the thesis.

[7] We are aware of Tournaments with only one participant does not make sense in reality, but for the sake of brevity and clarity additional participants are omitted.

2.4.3 Instance Adaption Over a Period

An instance of a model as illustrated in Figure 2.4 always represents a certain snapshot of the system for a certain point in time. Especially when discussing about dynamic and evolving entities, it is important to consider time as integral part of the discussion. Generally, an entity is considered as a semantic unit having a set of attributes. In our case, such entities evolve over time by start or stop playing and featuring Roles, respectively. More precisely, an entity consists of a core, a set of played Roles, and a set of featured Roles. The dynamics and adaption comes into the game, when a core starts playing a new Role and, thus, gains additional attributes, i.e. the structure of the entity changes. The other way around, stop playing Roles results in a different structure, too.

Such a change of an entity's structure is shown in Figure 2.5 on an abstract level. For a clear arrangement, the concrete attributes are shown only in beginning, but are omitted in the rest of this illustration. All attributes of each instance's type used in this example, can be found in Figure 2.3. The example period consists of seven points in time ($t_0, .. , t_6$) and the focused entity is *John*. At t_0 *John* is just a **Person** who does not play any Roles. Hence, the entity consists of the **Person's** attributes only. This situation can be seen as basic set of attributes, i.e. the minimum structure an entity of the type **Person** will have. Less structure is not possible. At t_1 *John* registers at a university and becomes a **Student** *s1*. The concrete university this particular Role is featured in does not matter with respect to the entity *John*, because the university itself is a separate entity. However, the entity *John* grows and contains the attributes of **Students** in addition to the core set of attributes. As a **Student**, *John* is now able to take **StudiesCourses**, thus, the **Student** Role *s1* goes in Relationship to a **StudiesCourse** (the StudiesCourse itself is not shown in Figure 2.5) at t_2. Later on at t_3, *John* becomes a **StudentAssistant** *sa1*. Therefore, the entity grows once more and additionally contains all attributes of **StudentAssistant**. Beside his university life, John likes football and joins a football team as **TeamMember** *tm1* at point t_4. Again, the entity grows and changes its set of attributes. The same holds for t_5, when *John* starts to be the **Captain** *cap1* of his football team. Additionally, *John's* structure at this point in time conforms to the valid schema instance example in Figure 2.4. Finally, at the point in time t_6 two actions take place. At first, *John* graduates, hence, the Roles *s1* and *sa1* are dropped from this entity. Secondly, *John* joins another **SportsTeam** as **TeamMember** *tm2*, which is added to *John*. Consequently, the set of attributes differs from t_5 to t_6. Moreover, the university graduation does not affect the Roles played in several **SportsTeams**.

2.5 SUMMARY

Starting from traditional modeling approaches, we examined their shortcomings for highly dynamic and complex use cases. In particular, modeler can run into undecidable situations or produce semantically incorrect models. At runtime, there exists problems caused by static typing regarding the fix set of attributes and methods. To overcome these issues, the concept of roles, as methodology to separate the concerns an entity may have during it whole lifetime, was introduced. In detail, dynamic runtime adaption is provided by playing roles, which can be started or stopped dynamically during runtime allowing entities to gain new structure and behavior, without reinstantiating the entity itself. Over the past decades, many approaches for role-based modeling and programming have been proposed, resulting in a large zoo of different notions. To distinguish and classify them, the 15 role features by Steimann and the eleven additional features by Kühn et al. have been explained. Furthermore, the term compartment was introduced and separated from the often-used term context to have a clear distinction between both terms. As result of the classification and to build on a sophisticated metamodel, we decided to rely on the Compartment Role Object Model, which was introduced by providing an ontological foundation and a detailed schema and instance level description. Finally, a running university management scenario was outlined and explained on the schema and instance level, as well as for instance adaptation over a period. This scenario is illustrated using the CROM metamodel and its graphical notation.

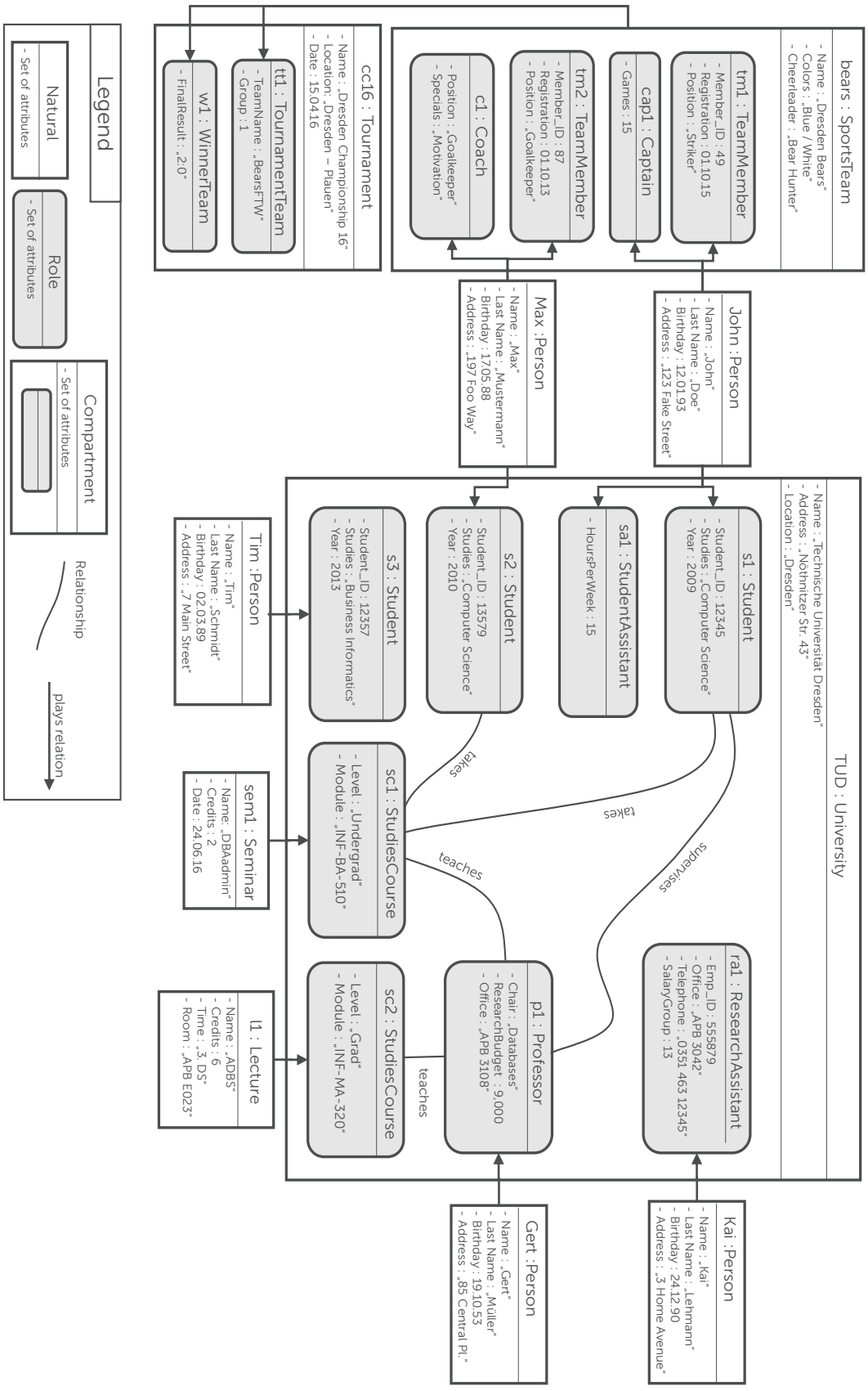


Figure 2.4: Valid Instance of the University Scenario Schema (Figure 2.3)

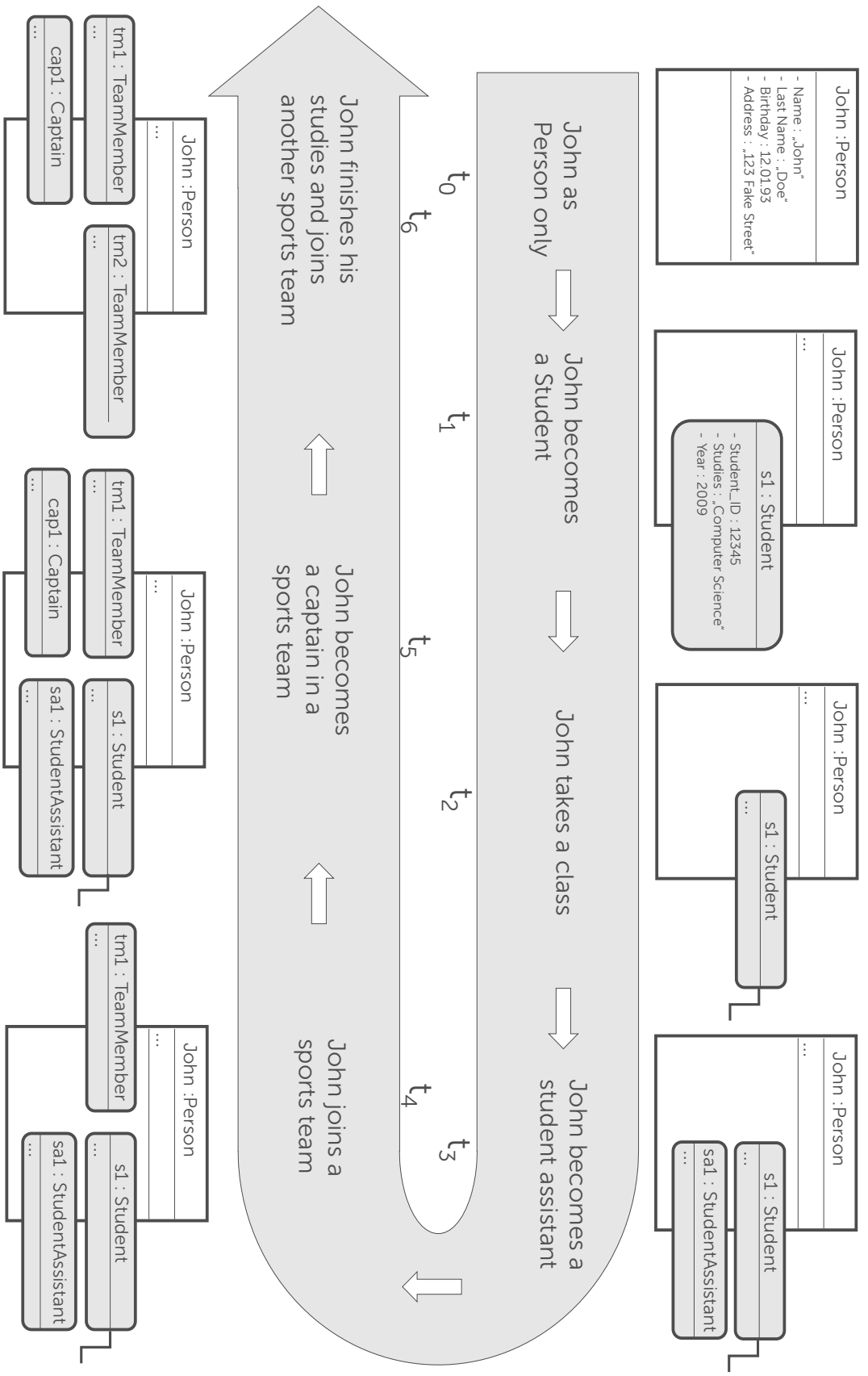


Figure 2.5: Timeline for the Scenario Based on the Instance "John"

NEED FOR ROLE-BASED DATABASE SYSTEMS

The concept of roles and its separation of concerns can be implemented in modeling and programming languages enabling the development of role-based software systems. These systems are able to adapt the entity's behavior and structure dynamically at runtime, depending on the context. In particular, we assume CROM as metamodel for the conceptual models as well as for the programming language. However, a continuous support throughout the entire software system is not guaranteed, because traditional database systems (DBSs) are not able to represent these semantics and the corresponding constraints natively. This results in a role-relational impedance mismatch. In particular, this mismatch describes the presence of a metatype distinction on the application level, but the absence of such a distinction on the database level. To overcome this mismatch and enables a continuous and crosscutting role support, we argue for a database integration of roles. Generally, this argumentation is focused on the usability aspects of a DBS within a software system and not on performance issues. We firstly describe the DBS's environment and its usual tasks within a software system, once in a traditional setting and once in a role-based setting assuming today's available technology. The latter scenario includes role-based modeling languages and programming languages in combination with a relational DBS. The setup in this latter scenario gives rise for the role-relational impedance mismatch that causes several problems. These problems will be classified into three classes, in detail, problems for the applications and application developers, issues for the database system, and finally problems for the software system in general. To solve these problems and overcome the role-relational impedance mismatch, several requirements for a role-based DBS are defined. Based on these requirements several related approaches are evaluated; starting from simple relational techniques like relational views, over mapping engines to more sophisticated, but not fully integrated solutions like the Information Networking Model (INM). Finally, we briefly outline the RSQL approach, which is designed meet the requirements and tackle the problems posed by the role-relational impedance mismatch, and conclude this chapter.

3.1 ECOSYSTEM OF DATABASE SYSTEMS

Today, software is almost everywhere and influences our modern day to day life; it runs our SmartTVs, it may be the backbone of businesses, or it runs the university administration. Sommerville defines software as "not just the programs, but also all associated documentation and configuration that is needed to make these programs operate correctly." [76]. However, software does not exist in isolation, rather it is combined in a software system to perform a superordinate task. For instance, a software system in a university administration may consists of distinct programs to manage the general student information, the exams, and lectures. Usually, software systems follow a layered architecture in which each layer relies on the abstraction and functionalities the layer below provides. An operating system, for example, provides basic functionalities to interact with a computer's hardware components. In contrast, a database system offers efficient data management capabilities based on the operating system's functionalities. Finally, client applications that require persistent data storage often use database systems for that purpose.

In our case, we consider a heterogeneous software system, from a DBS perspective, that consists of several applications, which manage their internal data structures transiently. That means, the data structure lives as long as the application runs. Consequently, the application data will be lost if the program terminates. Furthermore, we consider application servers or middleware systems in our software system that encapsulate business logic for some clients. Thus, there exist two kinds of possible applications. At first, applications that bring their own business logic and access the database directly, and secondly applications that rely on business logic definitions in an application server that also manages the data access for those applications. Additionally, each application has its individual local perspective on the data it handles. Moreover, a DBS is part of our considered software system, to store transient application data persistently. It is also assumed, that several applications and

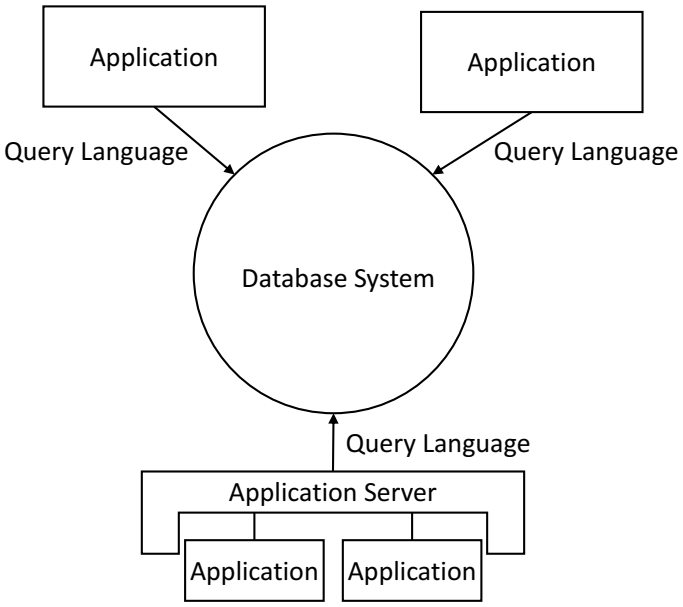


Figure 3.1: Traditional Ecosystem of Database Systems

application servers access the same database to share data, omit redundancy, and rely on a globally consistent data set. In regard of a clearly layered software system architecture, the DBS encapsulates all standard functionalities and required abstractions for the persistent data management on which the applications and application servers can rely. Hence, the DBS is an important part of a software system; it maintains a system global perspective on the data and acts as **single point of truth** within the software system. In consequence, the DBS, which is the central concern in this thesis, usually faces several heterogeneous interaction partners. All of these partners access the database system by a certain query language to store and restore their data as well as for analytical purposes or ad hoc queries. A conceptual overview of our considered DBS ecosystem is illustrated in Figure 3.1.

As example for such a software system, imagine a university management system, managing general student as well as employee information, exams taken by students, and lectures. All three applications rely on the same database that stores the personal master data, the student information as well as exam and lecture data. At first, there is a general management application, managing all personal master data as well as the student and employee information. For legal reasons, only authorized administration staff members can access and manipulate the exam data of students. This is encapsulated in an individual application. Finally, professors can set up lectures to which students can enroll. This is captured in a separate application, too.

Today's software is most often programmed in an object-oriented way using programming languages like Java[1] or C#[2]. Object-orientation (OO) abstracts entities having the same behavior and structure to classes [22]. At system runtime, classes are instantiated to runtime objects having a fixed set of methods and variables. Usually, at the application's runtime objects are transient and for persistence purposes a DBS is employed. Today, most applications that require persistent durable data storage, rely on a relational DBS, which manages the data in tables, the DBS abstraction of relations [55]. Unfortunately, object-orientation and the relational concept differ at important aspects of their paradigm, like the entity identification or inheritance. This difference results in the well-known object-relational impedance mismatch between object-oriented applications and relational DBSs [55]. However, this mismatch is manageable, because both paradigms share some features. For instance,

[1] `https://www.java.com`

[2] `https://msdn.microsoft.com/de-de/library/kx37x362.aspx`

both rely on static types only, namely classes in the case of object-orientation, and tables in the case of a relational DBS. Furthermore, this gap can be bridged by object-relational mapping engines, like Hibernate[3], which act as mediator between object-oriented applications and relational DBSs. However, the communication between object-oriented applications and the relational DBS is usually performed by the Structured Query Language (SQL) [45, 46]. In total, we consider traditional applications to be object-oriented and store their data in relational DBS, whereas object-relational mapping engines bridge both worlds. An abstraction of this situation is illustrated in Figure 3.2.

Additionally, this illustration includes the traditional design process, having its source in a conceptual domain model, for such software systems. Generally, developing a new or extending an existing software system is a quite complex job. It requires several stakeholders, like domain experts, software architects, software developers, database experts as well as users, to bring a new system to life. It is commonly practiced and well-known that a software system is best designed by conceptually modeling the software system's domain first. A conceptual model captures all aspects of a domain without any implementation specific information and is usually designed by domain experts. Once all software system stakeholders agreed on a conceptual model, this model is transformed into several implementation target models, like an implementation model for each application or the data model for persistent data storage. For instance, the aforementioned university management software system is designed in a general conceptual model and subsequently transformed into three object-oriented application specific models for the general master data management, the exam management and the lecture management application.

From a database system perspective, the conceptual domain model is transformed into the logical database model; in the traditional case it is a relation model [19]. For instance, relations storing the personal master data and the student information are created. Additionally, application specific perspectives on the data are created as external views to provide the applications customized data access to a subset of the whole data. These views are defined during the applications' design phase and are based on the logical database description as well as the data access requirements of the corresponding applications. For example, the lecture management application defines an external view that combines the master data, the student information and the lecture information, but not the exam grades. Finally, the implementation specific models are implemented into object-oriented program code and a relational database schema by application developers and database experts, respectively.

Roles in the Software System

As outlined in Chapter 2, role-based techniques have been invented and proposed as extension to the object-oriented paradigm, because this paradigm causes problems when faced with the requirements of modern ubiquitous applications and software systems, like representing context-sensitive and dynamically evolving entities. In detail, we consider role-based features that are built on the foundations of the Compartment Role Object Model (CROM) [57]. The most important feature of CROM is the distinction between several metatypes having their own semantics, to enable a separation of concerns within an entity (see Section 2.3). These metatypes are Natural Types, Role Types, Compartment Types, and Relationship Types. By introducing CROM to build a context-sensitive and role-based software system, the traditional software system design process as well as the database ecosystem remain conceptually the same, but the components change.

Today, state of the art role-based technology enables domain experts to use CROM for the conceptual design, application design as well as in programming languages. In detail, FRaMED[4] implements

[3] `http://hibernate.org/`

[4] `https://github.com/leondart/FRaMED`

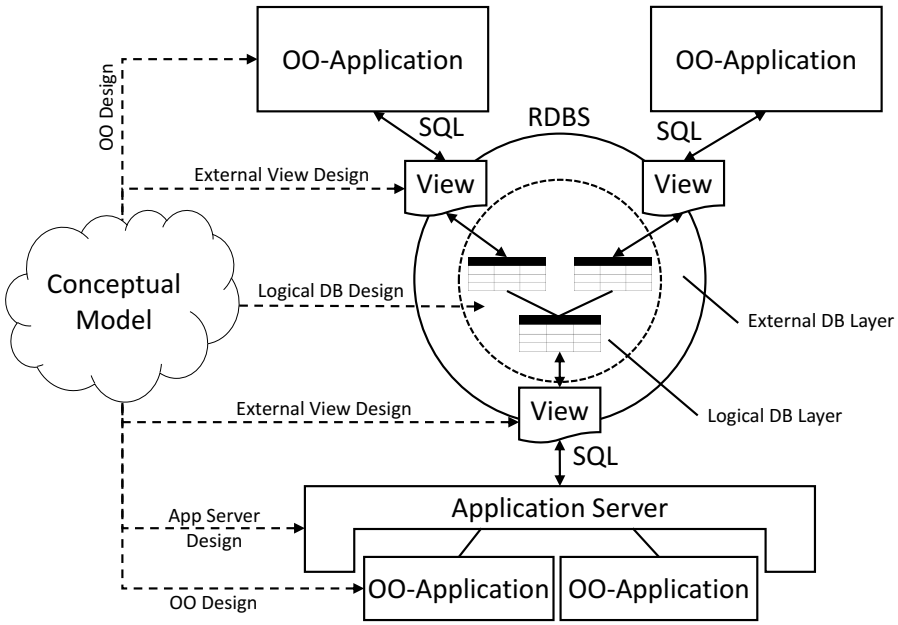


Figure 3.2: Traditional Software System and its Design Process

the CROM metamodel for the conceptual design as well as application design. On the programming language side, SCROLL[5] provides CROM support for the Scala programming language [59]. Unfortunately, for database management systems no such approach exists. As a consequence, we assume the following technology situation for context-sensitive and role-based software systems. The database system's ecosystem consists of a relational DBS as persistence provider, hence, the database layer is not changed at all and provides the same standard functionality to applications as in the traditional scenario. On top of the DBS there are several role-based applications handling their runtime data structures and entities by differentiating between the CROM metatypes to enable dynamic behavior and structure adaptations. Additionally, there exist some middleware or application servers that encapsulate the role-based business logic for some role-based applications. This specific situation is depicted in Figure 3.3[6].

The software system design process is adjusted to the new situation as well. At the beginning, there is role-based conceptual model that is created by the domain experts. Afterwards, this conceptual model is transformed into role-based application designs and the traditional relational database model, the logical database design. Finally, the logical design models are implemented. In case of applications, those will be implemented in role-based and context-sensitive application code, whereas the relational model is implemented in a physical relational database schema. Additionally, external views are created to provide an application-specific perspective on the data. Those are created on the basis of the role-based application requirements and the relational data model. This design process is illustrated in Figure 3.3.

In sum, today we are able to conceptually model a software system's domain in a role-based and context-sensitive way as well as writing the corresponding application code using CROM. In contrast, there is no way to explicitly store CROM's role-based semantics and metatype distinction a DBS. Hence, considering the DBS ecosystem, all software running on top of the DBS has a role-based perception of the software system's entities. In this regard, the main question is: *Do we need a notion of*

[5]https://github.com/max-leuthaeuser/SCROLL

[6]In the following architecture illustrations, all software system components having a notion of roles are depicted in a gray color.

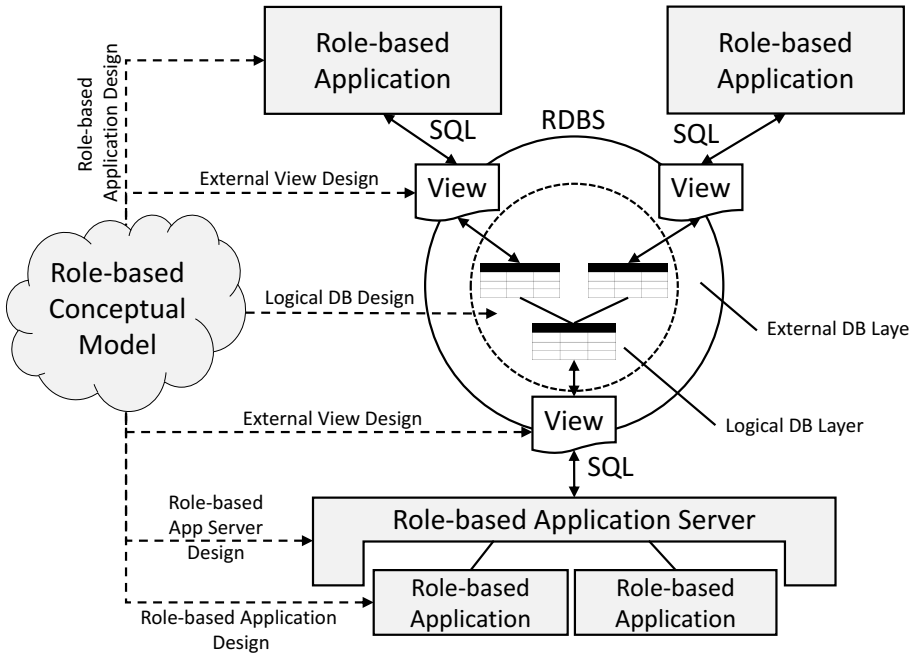


Figure 3.3: Today's Role-based DBS Ecosystem and its Design Process

roles, more precisely a notion of CROM and its semantics, in the DBS as well? To answer this question, whether a CROM-based notion within the database system is required or not, we are going to outline and discuss possible problems that arise when using role-based conceptual models and programming languages in combination with traditional, especially relational, database systems.

3.2 ROLE-RELATIONAL IMPEDANCE MISMATCH

The main problem is the **inability of these traditional database systems to explicitly represent the metatype distinction** of Natural Types, Role Types, Compartment Types, Relationship Types, and their interrelations. Hence, the DBS cannot provide a role-based standard abstraction out of the box, on which the applications can rely. Thus, dynamically evolving entities in applications cannot be directly represented in the DBS and have to be encoded using available relational techniques. As a result, the software system suffers from a **role-relational impedance mismatch**. In addition to the well-known object-relational mismatch, which occurs in object-oriented systems in combination with relational DBSs, the role-relational impedance mismatch is characterized by the role-based metatype distinction on the application layer and a missing differentiation on the database layer. Generally, the role-relational impedance mismatch is orthogonal to other impedance mismatches that might occur in the system. In case of role-object-oriented programming languages, like SCROLL, the role-relational impedance mismatch can be seen as extension to the traditional object-oriented impedance mismatch, because traditional problems regarding the object-oriented impedance mismatch are not affected. However, applications handle role-based objects that change their structure and behavior dynamically during runtime and in case of persistence these objects have to be reflected in the database system as well. Considering the current technology situation, role-based software systems face a more complicated impedance mismatch that is a combination of the object-relational and role-relational impedance mismatch. In the following argumentation, we focus on the role-relational impedance mismatch only, because the object-relational is out of scope in this thesis.

In consequence of the role-relational impedance mismatch, the CROM's metatype distinction is encoded using the available traditional techniques, like naming conventions for relational tables, resulting in a **loss on semantics** from the DBS perspective. Additionally, the CROM metamodel constraints, like each Role Type needs at least one player type and is located in exactly one Compartment Type, cannot be checked by the DBS. As aforementioned, the DBS is referred to as single point of truth in a software system ensuring global consistency across several applications. By losing the ability to check the constraints properly, it **loses the single point of truth criterion** as well. In consequence, invalid schemata may be implemented in the system, which are valid locally seen, but invalid from a global perspective. For instance, imagine an application that models the **Professor** Role Type within the **University** Compartment Type and another one having the **Professor** within an **Employment** Compartment Type, as shown in Figure 3.4.

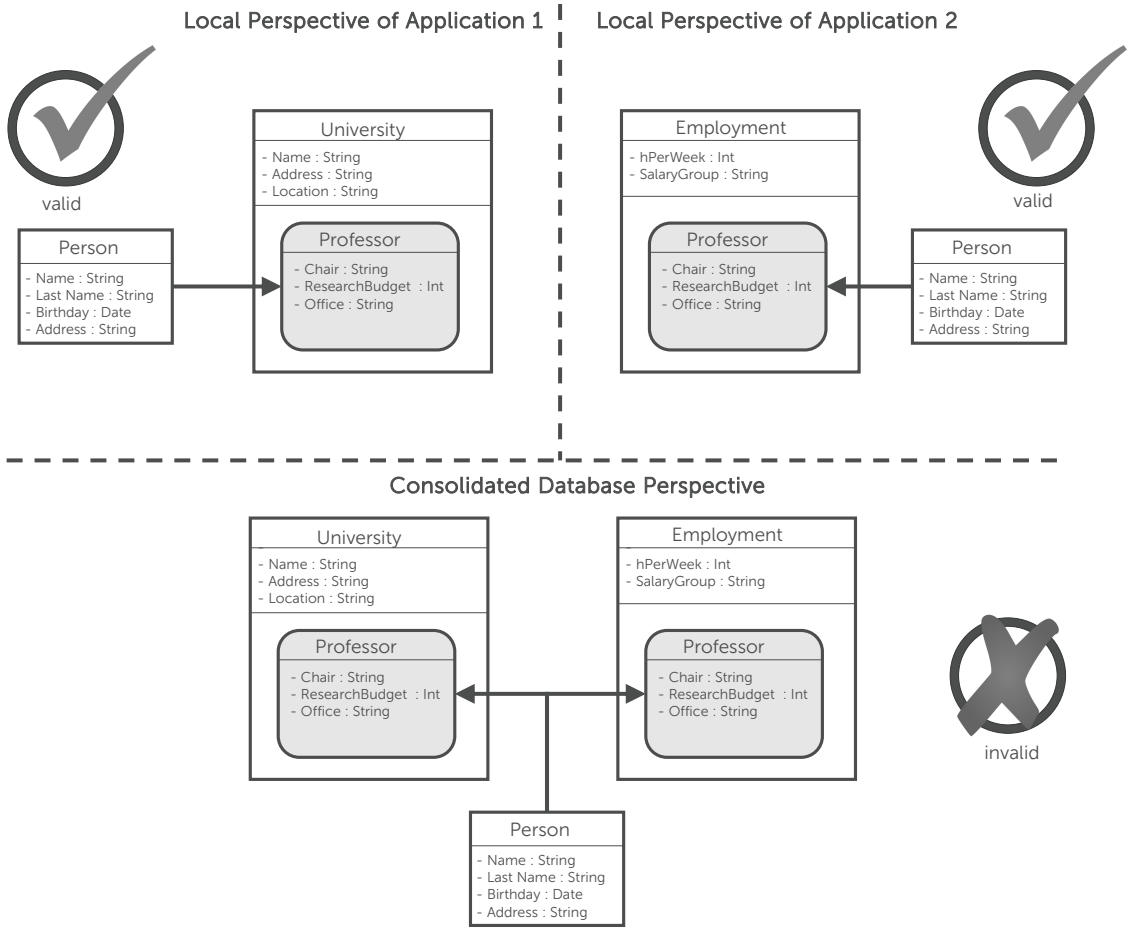


Figure 3.4: Locally Valid Application Schemata vs. Globally Invalid Database Schema

Locally seen, both are valid schemata, but in the global perspective this is an invalid schema, because a Role Type is defined within two different Compartment Types, which is prohibited with respect to the CROM metamodel. This cannot be checked by a single application, there has to be an agent having a global perspective, to ensure this constraint. However, a relational DBS cannot perform this check, because the necessary information is not present and schema checks are executed outside of the DBS. This fact is fatal for the DBS as main data management software, because it is downgraded to a simple storage place without any power to ensure system-wide consistency according to the schema. With respect to the fact that DBSs mainly do not care about the data, but schema, the situation of being unable to explicitly represent and check for a correct role-based schema, is unacceptable from a DBS's perspective.

However, there exists a conceptual separation line between the transient application world and the persistent DBS world. Due to the loss on semantics this line is moved into the transient world. Basically, the persistent DBS layer cannot take care of the whole data management, including the role-based consistency guarantees, hence, applications and users have to compensate this inability by manually implementing some persistent data management features in the applications. In sum, the role-relational impedance mismatch describes the inability of a DBS to represent role-based metatype distinctions and a shift of data management functionalities towards the applications. This situation causes issues and pains that can be classified in three categories, which we discuss in detail in the following: (i) Issues for applications and application developers, (ii), pains for the database system itself, and finally (iii) problems for the software system in general.

3.2.1 Problems for Applications and Application Developers

The first category comprises all problems related to role-based applications and their developers. The main problem in this category is that applications have to take over some data management functionalities, because the strict separation line between the applications and the DBS moves into the transient world. This situation results in several subproblems.

At first, application developers have to manually program the unavailable standard functionalities for a role-based data management by themselves, for each application. This brings them back in data management, causing additional tasks during the development process and a loss of their focus. In fact, application developers should be concerned with the application's functional and technical implementation and not with enforcing the correct metamodel mapping and constraints in the database. In general, the application developer will face more tasks to implement, which results in longer project development time, and thus, higher application development costs. Additionally, the same functionalities, in this case data management functionalities that cannot be performed by the DBS, are implemented in several applications individually. This results in code that is redundantly written for each application.

Secondly, the database system does not provide a standard abstraction for role-based semantics; additionally there is no standard mapping from this semantics onto relational tables. On these grounds, each developer chooses the mapping individually, resulting in possibly inconsistent mappings and different constraint checking techniques. The decision heavily depends on the preferences and experiences of the application developer with DBSs. More complicating from the application developer perspective, CROM defines constraints on the schema level as well as the instance level that need to be guaranteed. Especially the schema constraints, like the prohibition of empty Compartment Types, are hard to check in the database system itself. For instance, assume each type is mapped to an individual table and all tables representing a Role Type also reference the player types and Compartment Type they are played in. Such a situation is shown in Figure 3.5. In detail, this illustration shows three tables, one for the Compartment Type **SportsTeam**, one for Compartment Type **Tournament**, and one for the Role Type **TournamentTeam**. Additionally, the **TournamentTeam** table (RT_TournamentTeam) has columns that represent a referential integrity constraint to the player type **SportsTeam** (FK_ST) and one to the Compartment Type **Tournament** (FK_T). To check the schema validity with respect to non-empty Compartment Types, the database catalog has to be queried for existing foreign key constraints between a Role Type table and the corresponding Compartment Type table, because this information is not directly inferable from the tables itself. In particular, this requires each application to have access to the database catalog, which is obviously not a good idea, according to data security and privacy.

Unfortunately, the foreign key constraint does not provide any information whether it references a player type or Compartment Type it is played in. This can be confusing when Compartment Types

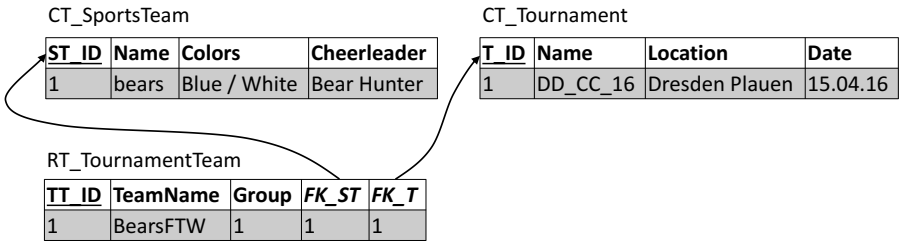


Figure 3.5: Possible Role to Relation Mapping Featuring Foreign Key Constraints

are able to play Role Types in other Compartment Types, like the **SportsTeam** Compartment Type in Figure 2.3 and Figure 3.5. In such a case it is not clear, which foreign key constraint references the player type and which the Compartment Type it is played in. In detail, a check for a referential integrity constraint on the table **SportsTeam** will provide a positive result, even if it does not feature any Role Types and only is able to play Role Types. Consequently, a check for the existence of referential constraints towards a certain table can result in false interpretations. Even naming conventions only help conditionally, because the interpretation of referential integrity constraints is manually implemented in the applications and can vary from application to application. In sum, this problem can lead to invalid schemata with respect to the role-based metamodel.

Thirdly, applications and especially their mappings are required to be aligned and synchronized for shared data objects. Traditionally, each application implements its mapping individually and independently of other applications. In case of shared entities or shared parts of an entity, for example Role Types, the different applications need to be aligned to a common mapping, at least for the intersection points. This problem is not exclusive for role-based software systems and applications, it also common in object-oriented software, but it becomes much harder to align, because standard mappings are unavailable. However, application developers are concerned with applications they are not responsible for and they have not programmed, too. In particular, an application developer is required to look into and understand other applications in the same software system, to chose the correct mapping or query the correct tables, and to align all dependent applications properly. This can result in inconsistent mapping schemes within an application, because different applications may have different mapping schemes, which have to be combined in another application.

For instance, assume the lecture management application is implemented as extension to a running university management system (see Figure 2.3) comprising the applications for general master data management and exams. Of course, the lecture management is required to rely on the master data, especially on the Natural Type Person and Role Type Student. In consequence, the lecture management also relies on the master data mapping of the other application. These master data are stored by a mapping shown on the left-hand side in Figure 3.6. The information about which **Person** plays which **Student** at which **University** is encoded in the relation of the **Student**, extending this relation by two additional columns for the referential constraints. However, the application developer for the lecture management decides to store the information on which Natural plays which Roles in which Compartment in separate relations, resulting in a mapping illustrated on the right-hand side of Figure 3.6. To align both applications, the lecture management application has to implement the mapping of the already existing applications. In the example case, the mapping in the lecture applications is changed according to the master data. As a consequence, the lecture management application implements two different mappings for the same type of information, in particular, the information which Natural plays which Role in which Compartment. In the worst case, there exists another application implementing a totally different mapping schema that has to be adopted in the lecture management, too. Especially the absence of a standard mapping and possible inconsistencies for mapping the same type of information, complicates application maintenance as well as application extension.

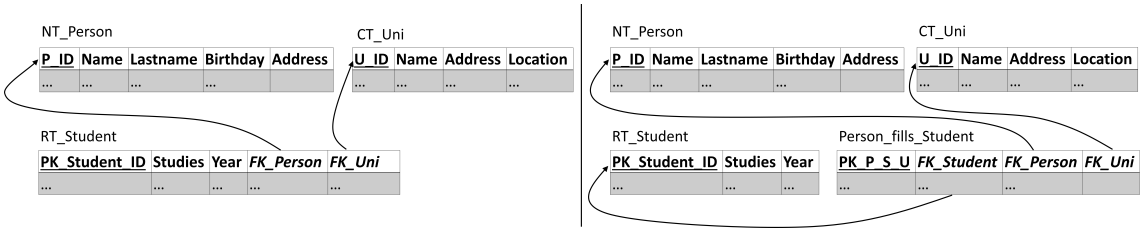


Figure 3.6: Two Mappings of the Same Relation Between a Person, a Student, and the Corresponding University

Finally, application developers have to think in two different worlds, because they are not only concerned with the application's data model, but the database data model as well. This complicates the application development process and may result in a *lost in translation* phenomena. This describes an application developer's confusion when continuously thinking in two or more different worlds. In particular, application developers of role-based applications code their applications in a role-based way, but have to store the persistent data in a relational DBS that does not feature any role-based semantics. Additionally, there exist various mapping options for the same information. Moreover, to restore role-based data from the database, a relational result has to be parsed to reconstruct the original semantics. Consequently, application developers will face two different metamodels during the development process at different interaction points with the DBS, which requires them to jump forth and back between the metamodels. This can result in more error-prone applications.

In sum, the huge semantic gap between the metamodels employed on the application layer and the DBS complicates the application development process in terms of thinking in two different worlds and additional tasks for the application developers. Additionally, these developers get back in the data management game and, thus, lose the focus on their actual tasks. Moreover, the software system contains redundant code in terms of concurrently existing manual mapping schemes in the applications. All these problems are likely to result in a longer development period, higher communication effort between the developers, worse maintainability, and finally higher development costs.

3.2.2 Problems for Database Systems

The DBS's inability to explicitly represent the CROM metamodel, including its constraints, and the move of data management functionalities, like checking for the correct schema according to the metamodel constraints, toward the applications, cause problems for the DBS itself. These problems are discussed in the following. At first, the DBS loses an important criterion within the whole software system; to be the single point of truth facility. Usually, the DBS combines all local application perspectives to a system global perspective on the data and, more importantly, on the schema [72]. In the current situation having CROM semantics on the application level, but not on the database layer, applications are in charge of ensuring consistency with respect to the given CROM schema. These applications can only check their very local perspective for validity, but not from a global point of view resulting in possibly invalid schemata on the database layer. This situation has been discussed in the beginning of this section and is illustrated in Figure 3.4. In sum, the DBS loses its single point of truth criterion and is degraded to a simple background storage that does not ensure global data consistency.

A huge advantage of commercial relational DBMSs is their optimizer, which produces the best logical and physical execution plan based on a given optimization technique using an underlying cost model [72]. To produce such a plan, all information about the data, the schema, the data distribution, interrelations between the data, and other statistics regarding the data, is of priceless value for the

Redundant data

Person_ID	Name	Lastname	Birthday	Address	Student_ID	Studies	Year
...	...	...	...	...	...	...	...
1	John	Doe	12.01.93	123 Fake Street	12345	Computer Science	2013
1	John	Doe	12.01.93	123 Fake Street	872432	Business Administration	2014
2	Max	Mustermann	17.05.88	197 Foo Way	13579	Computer Science	2013

Figure 3.7: Redundant Data in a Relational Result

optimizer. Unfortunately, knowledge, especially the role-based knowledge and corresponding constraints, about the data is detained from the DBS and its optimizer. Consequently, CROM specific correlations between various tables storing CROM-based types cannot be utilized by the optimizer. As result, optimization potentials that are based on the special role-based semantics represented in CROM, remain unused, which finally may result in suboptimal query performance.

Furthermore, relational DBS produces a relational result. This result can explode in their number of returned rows very quickly, which is caused by the relational encoding of complex (role-based) objects. The same phenomenon can be observed when object-oriented runtime objects containing lists or collections of other objects, are stored in a relational way. Basically, the relational data model does not allow sets as value for a certain attribute [19]. Consequently, the combinations of the object and its set members is returned resulting in a combinatorial explosion. For instance, assume a Person who is a Student three times at three different Universities. The relational result for this data will consist of three rows, each displaying the Person information in combination with the particular Student data. Like in object-orientation, role-based data face the same problem, because a player type (Natural Type or Compartment Type) can play several Roles of the same Role Type simultaneously resulting in a combinatorial explosion in the relational result set as well. For instance, assume the relational result set in Figure 3.7. There, the personal information of the **Person** *John* is contained twice, because he is a student two times. Once for the combination of John and his first **Student** Role, in which he studies Computer Science, and once for his second **Student** Role representing the Business Administration studies. It is easy to imagine, that redundant data representation and transmission can become a problem. In consequence of such huge or fast growing relational result sets, the same data is redundantly transferred from the DBS to the application, producing higher transmission costs and network load in the software system. Even intermediate results can produce pains, especially in terms of memory and disk consumption during the result computation process, when they suffer from the combinatorial explosion as well. Is the intermediate result too big to fit into the available memory, parts of this result are swapped to disk, which always causes slower query execution due to the additional I/O operations. Additionally, such result sets are hard to interpret, which is due to the absence of a mapping from columns to a certain type, in fact the relational result set consists of a single table representing all information at once and with no metatype distinction.

In sum, due to the shift from traditional DBMS functionalities toward the applications, the DBMS's standing is dramatically degraded within the software system. In consequence, the usability, from a role-based application perspective, of the database system is much lower, because role-based data management functionality has to be implemented by the application itself. Additionally, optimization potentials in terms of knowledge about the metatype distinction and the metatype interrelations remain unused, because they are not part of the cost model and DBS internal statistics. Finally, relational result sets of role-based data can become very huge very quickly and represent data redundantly, resulting a higher data transmission volume.

3.2.3 Problems for Software Systems

The last class of problems comprises all issues that cannot be assigned to one of the other classes, because they affect the software system in general. At first, the software system suffers from a role-relational impedance mismatch, which is caused by the different semantics applied on the different layers of the software system. The missing metatype distinction on the database layer results in totally different semantic expressiveness on the various layers implemented in a software system. For instance, conceptual domain models, application's implementation models, and programming languages on the basis of CROM exist, which features a separation of four different metatypes to distinguish several parts of entities. In contrast, a DBMS featuring this metatype distinction does not exist. Additionally, there are no continuous and crosscutting role semantics throughout the entire software system. Of course, at some point in the software system, these semantics cannot be preserved and need to be mapped to something simpler, e.g., there is no file system supporting CROM's metatypes. However, the data structures handled on each layer totally differ from each other, which complicates a direct linkage between them. For instance, a runtime object that features several Roles of various Role Types is handled as integrated structure in SCROLL. Hence, this structure is handled as a semantic unit. On the relational database layer such a structure might be split into its components (see Section 3.2.1) that are stored in separate tables. For the DBS, this structural and semantic unit information is lost, thus, it handles each tuple in a table as independent and stand-alone tuple (which can be constrained by integrity constraints like foreign key constraints), which also causes object schizophrenia [40]. However, the role-relational impedance mismatch causes totally different models for the applications and database, complicated linkage between the entity's representation in an application and the database, as well as semantics reconstruction overhead in the system that has to be performed by the applications.

Secondly, the role-based software system is not layered and structured very well from an architectural point of view. In general, such systems are structured by outsourcing any non-application-specific tasks and functionalities into separate layers, which can be used by every application in the system [72]. For instance, the operating system provides functionality to access main memory and disk, the DBMS combines functionality to efficiently store and access data, and the applications are in charge for any client side functionality. This reduces redundant functionality implementation and facilitates a clear structure within the software system. However, the aforementioned separation line between the persistent DBS and transient application is moved toward the applications, because the DBS does not feature CROM's metatype distinction. To compensate this DBS's inability, applications perform tasks that are supposed to performed by the DBS. In consequence, a role-based software system cannot be structured very well with respect to the clear separation of concerns within the system. In particular, the absence of role-based standard functionality in the DBS forces the applications to implement these data management routines by themselves, resulting in redundantly implemented mappings and a distribution of data management tasks over the DBS and application layer. Consequently, software systems having a bad architecture, in the sense of unstructured layers, are harder to maintain and to extend than clearly structured software systems. On the ground of the RoSI goal, to enable role-based and context-sensitive software systems that adapt and extend their behavior and structure dynamically during runtime, this more complicated maintenance and extension is a huge blocking factor for a successful role-based software system. As a final consequence, such a layer-mixing system can produce more costs during the development phase, in terms of man power to implement, as well as while running it, in terms of more complicated and longer lasting error searches or extension implementation.

3.3 REQUIREMENTS FOR ROLE-BASED DATABASE SYSTEMS

To represent dynamically evolving and context-dependent entities, the concept of roles as conceptual primitive has been introduced in modeling and programming languages. This results in several problems outlined in the previous sections. Mainly, these issues are based on the huge semantic gap between the semantics utilized on the conceptual and application level, but not on the database layer. However, these problems are concentrated under the term role-relational impedance mismatch. Closing this semantic gap and overcoming the role-relational impedance mismatch can be achieved by introducing the concept of roles, especially a CROM-based notion, on the database layer. To bring the DBMS back in its rightful position as single point of truth in a software system that ensures global consistency, we are going to define several requirements for a role-based and context-sensitive DBMS that is able to overcome the outlined issues properly and lift the DBMS on the same semantic level as role-based modeling and programming languages. Additionally, a role-based DBMS enables a crosscutting and continuous role support throughout the software system. In particular, a holistic solution requires a role-based data model and role-based interfaces in conjunction with role-based result representation.

Data Model At first, a holistic approach requires a proper data model in the database, to natively represent role-specific semantics. This is the most important piece to enable layer crosscutting and continuous role support in the software system. In general, the data model defines the concepts, in which the actual data are logically and physically managed [72]. A relational DBMS, for instance, relies on the mathematical definition of a relation and stores the data in tables [19]. In contrast, a graph database may rely on the property graph data model and stores data and relationships in edges and vertices [5]. However, a holistic, role-based DBMS solution requires a metatype distinction in its data model, at least between Natural Types, Role Types, Compartment Types, and Relationship Types. Moreover, the data model is required to represent **role-specific consistency constraints** to ensure role integrity, like multiple player types for a Role Type, or only a single player per Role. Once the data model is defined, a **set of operators** is required. This set ensures which type of actions and operations can be performed on the data model and which output they produce. Additionally, the operators build the bridge between the data model and the query language, because the query language issues certain operators defined on the data model. Finally, we require a **mathematical closure** of the data model, so that each query on the role-based database produces an instance of the implemented data model itself. In sum, a role-based data model for a DBMS directly addresses the role-relational impedance mismatch and helps to avoid transformation overhead in the applications, keeps the application developers and the applications itself out of the database management, and brings the strict separation line between the transient and persistent world back to its traditional position.

Interface Secondly, the DBMS interface needs to be adapted to the new data model. Of course, this requires a new **query language**, because it is the primary communication interface between applications and users on the one side and the DBMS on the other side. Thus, the new query language needs to represent the same notion of roles, compartments and relationships as the database model does. This requires a new **data definition language** to describe the database schema as well as a **data manipulation language** to populate the database. Additionally, the actual **query language** in terms of retrieving data from the database, has to be adapted as well. One use case for a role-based DBMS is the **persistence** of role-based data objects handled in the application. A query language supporting role-based semantics and consistency constraints can simplify this persistence significantly, because highly specified mappings become obsolete and the query language is settled on a similar semantic level as the applications are. This also helps writing queries to retrieve role-based data from the DBS. Furthermore, role-based systems

have different query characteristics than non-role-based systems have, like exchanging roles on the fly when the context is changed. Consequently, the persistence provider on the client side needs to be adjusted to that type of queries or at least is required to provide functionality to support these queries. Moreover, an adapted **database connectivity** is required to give users and applications the role-specific semantics on the client side. Please note, there are various other interfaces, like user defined functions (UDFs), stored procedures (SPs), or wrappers to integrate heterogeneous data sources into the database, but those are out of scope of this thesis. Finally, a proper result representation completes the circle for full role support in DBMS. The result itself is required to be an instance of the database model, thus, all role-specific consistency constraints that are ensured by the data model are also guaranteed in the result. Additionally, users and applications need to be able to **iterate or navigate** through this role-based result, especially, accessing Roles and navigating from a Role to its player or the corresponding Compartment it is located in. However, a role-based query language in combination with a proper result representation addresses query writing problems at design and runtime, role-based persistence helps to ensure role semantics of runtime data objects, and an adjusted connectivity enables seamless client side support for roles.

3.4 RELATED WORK

To address the issues caused by the role-relational impedance mismatch, there exist several approaches that can be classified into four classes. The classification is based on their target software architecture. This means, each class implies a certain software system layout. In detail, traditional techniques rely on a layout, in which role-based applications implement their individual mapping and store the data into a relation DBS. These traditional techniques try to lower the negative effects of the role-relational impedance mismatch on the software system by using only the existing relational database techniques. Furthermore, mapping engines try to build the bridge between a traditional relational DBS and role-based applications by abstracting the mapping into a separate layer. Depending on where the actual mapping is performed, these approaches can be subclassified into client side mapping engines or database system side mapping engines. Persistent programming language combine traditional programming language techniques with persistent data management functionalities to merge the transient application world with the persistent database world. Thus, these approaches tightly couple the data management with corresponding applications, but neglect the central data management, as sharing and storage facility, by focusing on the applications only. Finally, DBS implementation approaches implement a logical database model to overcome this mismatch. The resulting architecture of these approaches aims to crosscutting role semantics; starting from the applications, over the query language and result representation, down to the DBS's data model.

3.4.1 Traditional Techniques

The first class of approaches comprises traditional techniques that do not implement a new data model or query language in a DBMS, rather existing and available techniques are used to lower the role-relational impedance mismatch's negative effects on software systems. From an architectural perspective, these approaches assume a classical software system as described in Section 3.1 consisting of role-based applications and middlewares, but a traditional and relational DBS as database layer. As aforementioned, this setting requires a manual and individual mapping of role-based runtime objects onto relational tables, for each application, which finally results in manual query writing by the application developers. Thus, the applications access the database directly by using a query language for

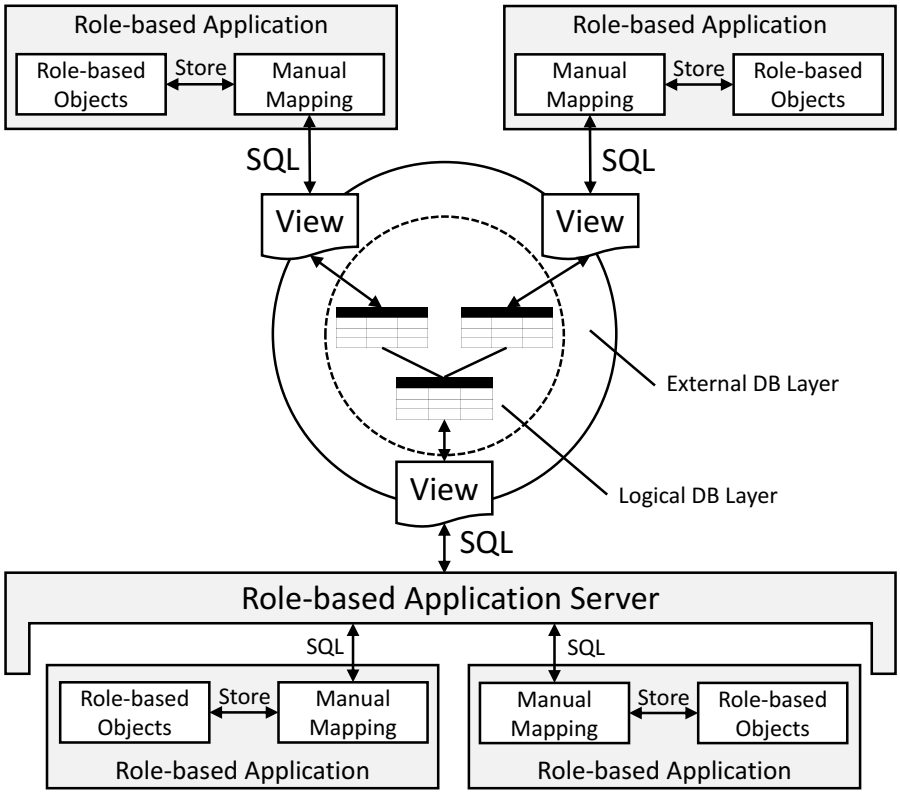


Figure 3.8: Role-based Software System Layout Utilizing Traditional Techniques

relational DBS, the Structured Query Language (SQL), for instance [48]. This situation is illustrated in Figure 3.8.

Additionally, this illustration shows the three abstraction layers of a DBS with respect to the ANSI/SPARC architecture [52, 72]. The first layer, in particular the physical layer, defines the physical storage layout of the data in the computer system, for instance, the page layout, indexes, or a partitioning of tables. Secondly, the conceptual layer defines what data is stored in the database, which means this layer represents the database schema without defining the physical storage. Finally, the external layer provides user-specific views on the data stored in the database. In contrast to the conceptual level, in which the data of the whole domain is defined, the external views describe only a subset of the whole database schema and expose this subset to the querying instance. Thus, the external views on the external layer can be aligned to user and application-specific requirements, like masking out certain columns of a table.

External View

Person_Student

Person_ID	Name	Lastname	Birthday	Address	Student_ID	Studies	Year
1	John	Doe	12.01.93	123 Fake Street	12345	Computer Science	2013
1	John	Doe	12.01.93	123 Fake Street	872432	Business Administration	2014
2	Max	Mustermann	17.05.88	197 Foo Way	13579	Computer Science	2013

Logical Schema

Person

Person_ID	Name	Lastname	Birthday	Address
1	John	Doe	12.01.93	123 Fake Street
2	Max	Mustermann	17.05.88	197 Foo Way
3	Gert	Müller	19.10.53	85 Central Pl.
4	...			

Student

Student_ID	Studies	Year	*P_P_ID*	*F_U_ID*
12345	Computer Science	2013	1	2
13579	Computer Science	2013	2	2
872432	Business Administration	2014	1	1
5463	...			

Figure 3.9: Relational Mapping of Role-based Data and a View-based Result

However, manual query writing is a pain for most of the application developers and as described in Section 3.2.1 it should not be their main task to care about the persistent data management and implement the corresponding functionality. External views are an option to optimize the data access for applications by providing an application-specific, but relational, perspective on the data. For instance, assume the role-based schema of **Person** and **Student** are encoded in an individual table for each type as illustrated in Figure 3.9. To retrieve all information about the students and their personal information, the Person and Student table have to be manually joined, like the following query demonstrates.

```
1 SELECT * FROM Person p, Student s WHERE s.p_p_id = p.person_id
```

In contrast, a view **Person_Student** that brings the personal information and student data together, can simplify data access, such that the user queries for the view instead joining manually. The corresponding query could look like the following one.

```
1 SELECT * FROM Person_Student
```

Moreover, such a view does not require any logical database design knowledge from application developers and users, but beside these obvious advantages of views, there are disadvantages. In the first place, views have to be updateable and insertable, because applications access the database by these views only, to keep the application developers out of manual query writing and covering complex joins. To be updateable, the SQL statement behind the view must not contain aggregations, `DISTINCT`, `GROUP BY`, `HAVING`, and `UNION (ALL)`. Additionally, the view has to address unique columns only and needs to include the key of a base relation [55].

Secondly, the base relations that store the data have to be created as well. External views require an existing logical schema, which hast to be created and maintained as well. For example, if there is no **Person** and **Student** table that store the data for the individual personal and student information, a **Person_Student** view cannot exist on that basis. In consequence, not only the base relations and the corresponding constraints have to be created, but the views in addition, resulting in additional effort when creating the database system.

This results in the next disadvantage, a possibly huge number of views has to be created and maintained. In general, each possible combination of a Natural Type and the playable Role Types defines a possible view, even if there is no instance associated to this view, which means the view returns an empty result set. For instance, assume the example of a Natural Type **Person**, a Role Type **Student** and a Role Type **StudentAssistant**, both playable by **Person**. In fact, the DBS has to provide four external views to the applications. In particular, a view representing the **Person** only, a view representing the **Person** in combination with the **Student**, a view combining the **Person** and the **StudentAssistant**, and finally a view that represents the **Person** in combination with both Role Types.

In sum, views cannot solve the role-relational impedance mismatch, in fact they only help by querying the relationally stored role-based data from an application perspective by covering complex joins. In contrast, database and software system problems in general are not addressed at all. This means, the DBS has neither a notion of CROM's metatype distinction, nor the schema constraints can be checked inside the database system and the results are pure relational. From the software system perspective, nothing has changed to the situation outlined in Section 3.2.3, the system remains unstructured and suffers from a role-relational impedance mismatch. Finally, external views neither provide a separate database model, nor a sophisticate role-based query language, nor a proper result representation.

3.4.2 Mapping Engines

The second class describes approaches that utilize mapping engines to store their role-based runtime objects in a traditional DBS. In contrast to the first class, a separate software, the mapping engine, is implemented in the software system to hide the mapping process from the application developers and the actual mapping from the applications. In general, there exist two types of mapping engines: (i) client side mapping engines and (ii) database system side mapping engines. Both types have in common to outsource the mapping process from the applications into a separate software. Hence, the actual database interface is hidden from the applications, which indirectly communicate with the database system by using a mediator.

Client Side Mapping Engines

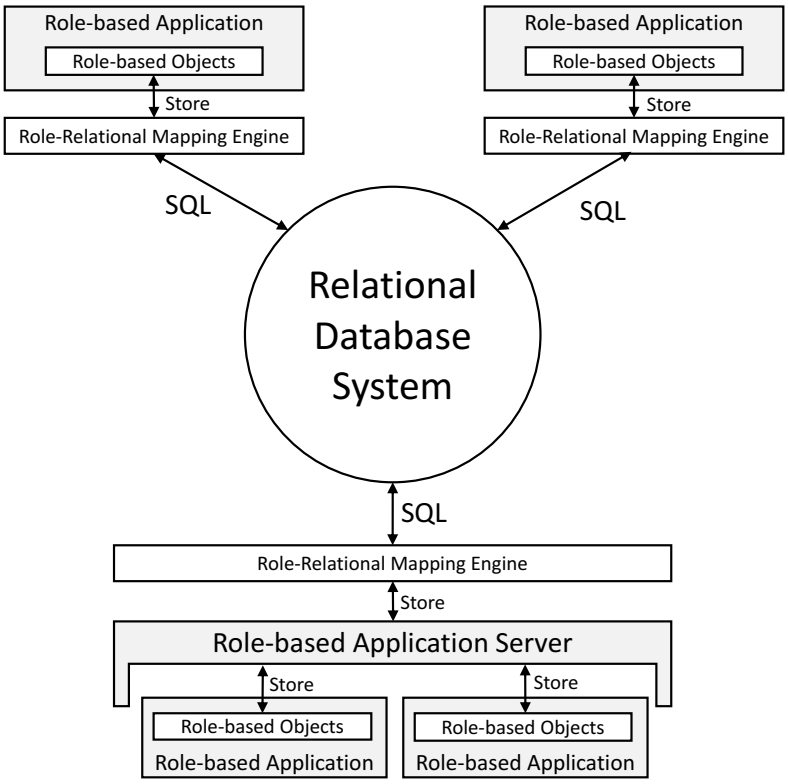


Figure 3.10: Role-based Software System Layout Utilizing Client Side Mapping Engines

Client side mapping engines are more common in a software system and usually they are employed to overcome the object-relational impedance mismatch and provide persistence for runtime objects. Hibernate[7], for instance, is a mapping engine for object-oriented applications into relational DBSs. Basically, client side mapping engines shift the manual mapping process into a separate software that can be employed for each application individually or on a middleware. The software system layout in case of client side mapping engines is depicted in Figure 3.10. On the application layer there are role-based applications and possibly some role-based middlewares. In contrast to the traditional techniques that implement an individual mapping for each application, this mapping is standardized and abstracted into a separate software. This software is implemented into the role-based applications

[7] `http://hibernate.org/`

and the middleware. However, the database system is not touched at all, hence, the software system relies on a relational DBS. The communication between the application layer and database layer is performed by SQL statements and queries, like in the traditional setting.

In comparison to the traditional techniques, client side mapping engines provide a role-based mapping abstraction to applications and keep the application developers almost out of the data management game. Thus, mapping engines can be seen as automated and standardized transformation technology of runtime objects onto the database data model.

There exist two representatives for this class of architectural solutions. In detail, the **Dresden Auto-Managed Persistence Framework** (DAMPF) [26] and **ObjectTeams JPA** [67], both rely on the programming language ObjectTeams[8]. This programming language is a member of aspect-oriented programming languages, but utilizes roles to extend runtime objects. Thus, it supports metatype discrimination in general. Even if the distinguished metatypes do not feature Compartment Types, these mapping engines are good to illustrate the general approach, including its disadvantages and weaknesses. In [27] DAMPF functionalities are explained using the example of schema evolution instead of roles, but basically it is the same procedure and workflow.

To illustrate the operation mode of client side mapping engines, DAMPF is explained in detail. Generally, Role to relational mapping is much more complicated than object to relational mapping, because the dynamic extension of the runtime objects has to be considered and recognized as well. Hence, these mapping engines have to take care of value changes, like traditional mapping engines, and additionally of structure changes of the corresponding objects. In detail, DAMPF consists of five following steps to enable role-based mapping and persistence [26]:

1. **Sublimate** – The domain model of the application is unfolded to reveal its internal data flow of role bindings. This information is stored in a fact base and is used to create the database schema. The event stream is utilized for role bindings or unbindings.
2. **Compare** – In this step the current application is compared to its last version stored in the fact base. For each element of the application, a fact base counterpart is searched and in case of any differences the schema and fact base will be adjusted accordingly.
3. **Adjust** – Fact base information and the database are adjusted according to the comparison step. Additionally, DAMPF implements a no-drop policy, which means deleted attributes in the application remain in the database, but do not show up in the queries.
4. **Trace** – This is an important step during runtime that traces the current state of an entity to log value and state changes.
5. **React** – The final step stores or restores domain objects to the database using one out of four different persistence strategies. The information collected during the trace phase are reflected in the database during this step.

The architecture of DAMPF is illustrated in Figure 3.11. In detail, DAMPF provides startup utilities and runtime utilities. The former one brings functionality to expose the original application's domain objects and data flow. Additionally, the database schema is created and altered, respectively. As output of the startup this framework provides a sublimated application with an exposed data flow, a schema as fact base and the relational database schema. The latter one enables tracing changes of entities during runtime and react accordingly. As input, the current application state stored in a fact base,

[8] `http://www.objectteams.org/`

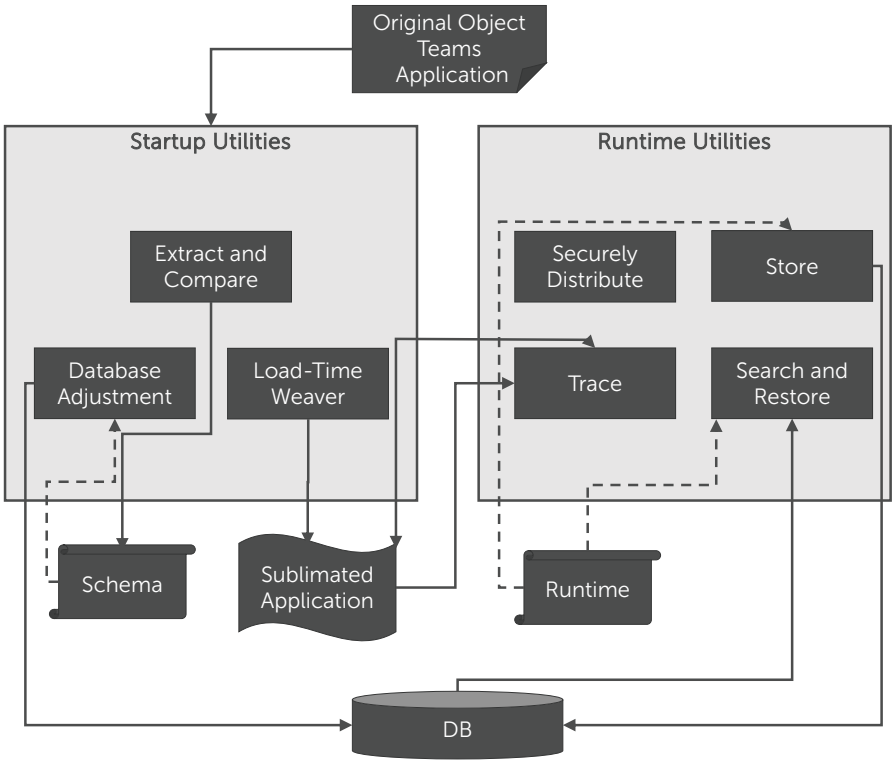


Figure 3.11: DAMPF's Detailed Architecture; According to Götz [26]

provides information when an entity needs to be stored, restored, or otherwise manipulated in the database. As you can see, DAMPF as persistence provider is tightly coupled with the applications.

Client side mapping engines address the problem of manual mapping by providing a standard mapping functionality. Hence, the application developers do not have to care about the persistent storage of their role-based runtime objects, which simplifies the development process. Moreover, traditional database system can be employed in the software system. Unfortunately, these mapping engines focus on the application side only and neglect the user component in the software system. However, they neither feature a proper database model, nor a query language, nor a role-based result representation. In fact, client side mapping engines have their own internal data abstraction and may provide a query language to query these internal data structures. The query language offered to the application developer by such a mapping engine, like Hibernate Query Language[9], does not represent the query language utilized for the communication between the mapping engine and the DBS. In consequence of this architectural approach, the database problems and the software system problems remain. In sum, these mapping engines do not solve the role-relational impedance mismatch, they only lower the negative effects for applications and their developers, in fact much more than relational views do.

Database System Side Mapping Engines

In contrast to client side mapping engines, which are implemented in the applications, database system mapping engines pull the mapping down to the database layer which requires a DBMS adaptation. This is achieved by providing a special query language as external database interface, in case of a role-based software system, a role query language. Characteristic for such mapping engines is

[9] `https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html`

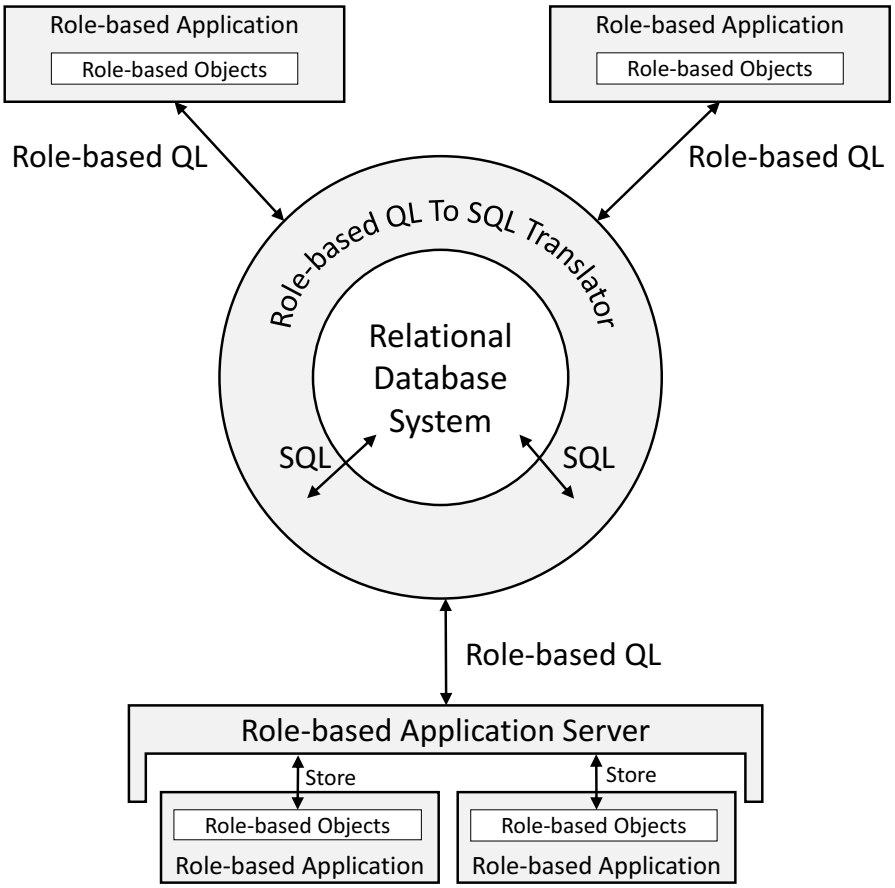


Figure 3.12: Role-based Software System Layout Utilizing Database System Side Mapping Engines

the background transformation of this query language into SQL inside the database system. A corresponding software system architecture is depicted in Figure 3.12. The system consists of role-based applications and possibly some role-based application servers. On the database layer an enhanced relational DBS is present. In detail, the enhancement consists of an additional layer within the database system, which transforms the role-specific query language into SQL statements. Consequently, the communication between the applications and the DBS is performed by a role query language.

In detail, the database system has to be extended by a special layer that sits on top of each other internal database system layer. This extension layer is responsible for parsing the role query language statements and transform these statements into SQL. The relational core of the DBS is not touched at all, thus, the optimizer is based on the relational algebra and statistics about data stored in relational tables. For users and applications the actual storage engine is hidden and they get the feeling of using a role-based DBS.

Representatives of database system side mapping engines are conceptual query languages, in general. In case of role-based query languages, there exists ConQuer [13, 33, 14], a conceptual query language based on the Object Role Modeling (ORM) methodology [34, 35]. Special mapping engines supporting Compartment Types and that are implemented in the database system in combination with a role query language, do not exist. ORM is an attribute free fact-based modeling language, which utilizes relationships instead of attributes. Consequently, both sides of the relationship can evolve independently enabling flexibility in terms of data modeling. In general, the primary modeling elements are entity types and value types. Roles are part of this language as well, but not as separation of concerns

within an entity. In fact, roles are named places in relationships between facts, i.e., between entities and values. Thus, the ORM data model does not provide a sophisticated notion of Role Types, Compartment Types, or Relationship Types.

ConQuer, as ORM-based query language, is designed to transform ConQuer queries into SQL and store the ORM data in a relational DBS [13]. Unfortunately, there is no syntax description for ConQuer, but it has been implemented into the query building framework InfoAssistant of Asymetrix [13]. Hence, ConQuer can be seen as graphical notation of a query language that does not feature a textual representation. Sadly, neither this framework is available nor the company exists anymore. Additionally, there is no discussion on the result representation of ConQuer queries, hence, it remains unclear if the query language is bidirectional in terms of receiving ORM-based queries and return an instance of the ORM metamodel, or unidirectional. In case of unidirectionality, relational result sets would be returned to the applications.

However, the idea of bringing the mapping into the database goes into the right direction to overcome the role-relational impedance mismatch in general. Even if ORM has a very weak notion of roles and ConQuer has no textual syntax notation they are good representatives for database system side mapping engines. In sum, a DBS internal transformation addresses the pains for application developers and applications by keeping them out of the data management and the problems of software system in general. In fact, by hiding the transformations inside the database, the software system will be structured more clearly. Due to the background transformation into SQL, the database system issues are not addressed at all and the database optimizer performs the optimizations on the relational data model, rather than on role-based data model.

3.4.3 Persistent Programming Languages

Persistent programming languages are characterized by their feature that the program's runtime objects can outlive the program runtime [4]. Storing a computer program's runtime objects on a persistent background storage is a fundamental part of such programming languages. Hence, the transient world and persistent world are merged into a single one, such that there is no clear separation between both worlds. To achieve persistence, programs may store snapshots of their current state on disk that can be restore on a program restart or after a failure. This is the simplest and most basic way, but does not provide any database system features like concurrent data access control or application-independent data representation. Moreover, each application is in charge of managing the persistent data individually and without any central data storage. Thus, persistent programming language focus on a single application rather than on a software system. A corresponding system architecture is depicted in Figure 3.13.

Basically, each application is extended by a persistent data management component that provides persistence for the application's runtime objects. In the same way, an application server is enriched by such a component and encapsulates the persistent data management from applications running on this server. As it can be easily seen, there is no central data management, like a DBS, employed in such an architecture layout, resulting in non-shareable persistent objects. Hence, there is no global domain perspective of the software system, rather only local ones. For instance, imagine the aforementioned lecture management application and the master data application. Both application persist their role-based objects individually and cannot share their persisted data. This results in redundant data storage and probably inconsistent schemata and data.

There exist two approaches that can be categorized in this class. At first, the Dynamic Object-Oriented database programming language with Roles (DOOR) approach has been proposed by Wong

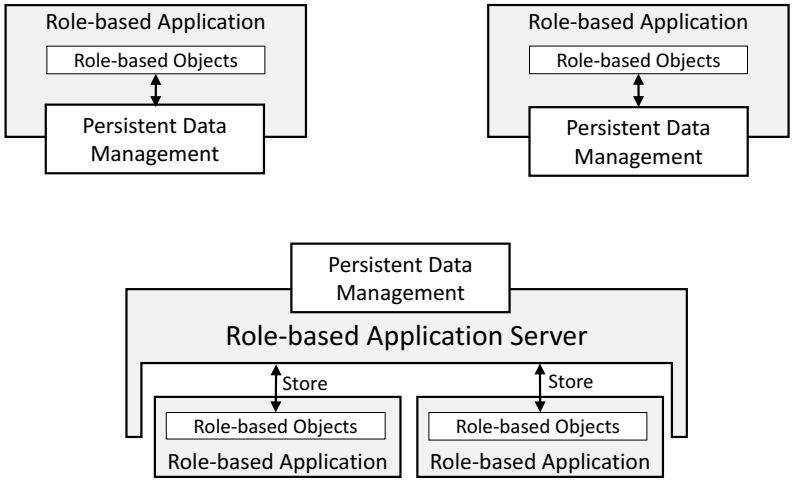


Figure 3.13: Role-based Software System Layout Utilizing Persistent Programming Languages

et al. [84, 86, 87]. Secondly, there exists Fibonacci, an object-oriented programming language providing persistence [4, 2]. DOOR as well as Fibonacci provide an extended object-oriented data model that has roles as first class citizen. However, both approaches claim to be a role-based database programming language, but in our perspective they are more a persistent programming language than a database programming language.

Database programming languages focus on the integration of database abstraction mechanisms into programming languages to avoid ad hoc queries that exchange information between the applications and the DBS [4]. They are integrated into a database system that hosts this language. Additionally, the database programming language operates on the same set of operators as the database system does to create and manipulate data. There is a fundamental difference in the design between a persistent and a database programming language. Persistent programming language are designed to provide persistence for a single application that is concerned with its local perspective on its data. In contrast, database programming languages are designed as extension to an existing database system, to provide database abstraction mechanisms within the programming language and to execute application code close to the data. Thus, the database system remains as single point of truth ensuring global consistency for usually several applications. Hence, database programming languages provide a convenient way to bring data intensive procedures into the database system by leveraging the database system's advantages of a global consistency with programming language comforts.

However, both approaches, DOOR and Fibonacci, rely on a simple serialization technique for persistence and focus on a single application scenario that does not consider data sharing between several applications [4, 84]. Moreover, they fail to present their application independent data storage, which is important for sharing data between several applications, within their storage engine. Additionally, there is no database system available that provides the set of operators these programming languages rely on, for example to bind or unbind roles to an object. Thus, DOOR and Fibonacci are classified as persistent programming languages.

DOOR

Wong et al. propose an object-oriented programming language for DBMS that is enhanced by role semantics, called DOOR [84, 86, 87]. In particular, they define object and role classes that are related by a *played by* or *own* relationship. Additionally, theses classes may form individual inheritance

hierarchies. Nevertheless, the data model does not feature a notion of context and relationships. The resulting data structure is an object graph in which objects point to each other. An example for creating a schema within DOOR is presented in Figure 3.14. In detail, a class **PERSON** is created having a *Name*, a *LastName*, and other attributes. Moreover, a role class **STUDENT** is created that can be played by a **PERSON**. This role class has the attributes *Student_ID*, *Studies*, and so on.

Additionally, DOOR features statements to manipulate objects and update values. However, a dedicated query language is not provided, rather path expressions are used to access roles in combination with a value-based selection to filter objects. In detail, they use a *!* between a player and the corresponding roles as delimiter, e.g., *Person!Student*, to determine a **Person** who plays a **Student** role.

DOOR manages each object and role using a unique ID and references the objects according to the *played by* and *own* relationship within the objects [87, 85]. Thus, objects and roles are individual instances and roles point to their player. This pointing may result in problems, in case the player object is deleted from the system [84]. However, the main reason for the DOOR approach is persistence of dynamic objects. As physical layer, the DOOR authors do not utilize a standard object-oriented or relational DBMS, in fact they implemented a simple storage system by themselves. This is based on serialization as binary data stream of objects using the metaobject protocol in a lisp-like language [87]. Thus, DOOR can also be seen as persistent programming language focusing on databases, but not as database programming language integrated into a database system. Unfortunately, the source code is not available and the project in general is not maintained anymore [26].

```
1 (create object-class PERSON:
2   STRING Name;
3   STRING LastName; ...)
4 (create role-class STUDENT played-by PERSON:
5   INT Student_ID;
6   STRING Studies; ...)
```

Figure 3.14: Example Statements to Create Object and Role Classes as well as Relate them in a DOOR Schema; According to [87]

In sum, DOOR is an approach, that extends the object-oriented paradigm by roles and implements this in a dedicated data model in combination with persistence. It suffers from a too simple metamodel that does not consider compartments and relationships and, thus, supports only role constraints without the integration into compartments. Moreover, the application and database system build an integrated unit such that the transient and persistent world are totally merged. Thus, logical as well as physical data independence is not ensured by the DOOR approach. Additionally, they usually focus on a single application, rather on a set of applications that access a database.

Fibonacci

Fibonacci is similar to DOOR. It is an extended object-oriented programming language featuring persistence functionalities [4]. In contrast to DOOR, the data model is different, for instance, roles of the same type can be played multiple times in Fibonacci. The language is strongly influenced by the Galileo language, but extends it by roles [3, 4]. In general, Fibonacci manages objects and an object internal graph of roles. Additionally, an object has a name, an immutable ID and a mutable state, which is inherited from the object-oriented paradigm. Moreover, objects are accessed by their roles only, thus, the object itself is only the container for the roles. All methods of an object are defined by the roles it is currently playing. Confusingly, each role instance can implement the corresponding methods differently. Hence, role definitions provide an interface only, the actual implementation depends on the instance. This can result in different behavior of various roles of the same type.

```
Let PersonObject = NewObject;

Let Person = IsA PersonObject With
  Name: String;
  lastname: Int;
  Birthday: Date;
  Address: String;
End;

Let Student = IsA Person With
  StudentNumber: Int;
  Studies: String;
  Year: Int;
End;
```

Figure 3.15: Example Statements to Create an Object and the Related Roles in Fibonacci; According to [4]

Fibonacci does not provide a dedicated query language, because the design goal is to overcome the traditional ad hoc query to result mechanism that does not require a query language in terms of select statements. Basically, objects build the container for roles, which are associated to their player object by an *IsA* relation. By the term object, the authors actually refer to classes in the object-oriented paradigm, which is confusing on the first sight. An example of a university domain defined in Fibonacci is presented in Figure 3.15.

This example firstly defines a **PersonObject** by **NewObject**, which states that a new container called **PersonObject** has to be created. Next, the **Person** information itself is created as role, because the container cannot have any attributes. This role is related to the **PersonObject** by an *IsA* relation. Hence, the object internal graph of playable roles is created and populated by the person. Finally, the **Student** is created as role playable by the **Person**. Thus, roles can play roles in Fibonacci. However, a role can be instantiated only once per object, thus, being a student at two different universities is not possible within the same object.

The architecture of Fibonacci consists of a compiler, a persistent hierarchical abstract machine (PHAM), and a persistent store [4]. The first component typechecks the Fibonacci expression and produces code for the PHAM component. That one manages the Fibonacci data structures on top of the data structures supported by the persistent store and executes the compiled Fibonacci code, like creating objects or storing them persistently. Finally, the persistent store manages a collection of data items that consist of an uninterpreted byte string. The architecture of Fibonacci is designed to serve single-user scenarios [4, p. 430]. Thus, they deploy the whole Fibonacci stack for each application individually and data between several applications cannot be shared. Hence, Fibonacci is considered to be persistent programming language rather than a database programming language.

In total, the role extension in Fibonacci is a bit different to that one in DOOR, but it suffers from the same problems. The role-relational impedance mismatch is solved from an application perspective, but not for the software system in general. Moreover, they do not present any solution on how the database problems, like global consistency guarantees for role-based data structures, are tackled by their approach. Thus, general problems remain, especially the fusion of the application and database world is problematic, because it neglects the users and multi-application scenarios, rather each application manages its data individually.

3.4.4 DBS Implementation

Finally, the last class of solutions comprises approaches that implement a metatype distinction as first class citizen into a DBS, but in contrast to role-based database programming languages as separate and application independent data model. These approaches aim for a holistic solution by providing traditional database features like multi-user operation mode and transactions, but enhanced with role semantics. They implement the database model directly into the DBMS by introducing a new logical and physical data model to the database. This is the optimal case for a software system, because the database management system has a notion of the metatype distinction and it manages the role-based data objects independently of applications. Hence, the classical software architecture is preserved. Moreover, the database issues are directly addressed by the introduction of the new database model, which could not be fulfilled by other approaches. The corresponding software architecture is illustrated in Figure 3.16. It consists of role-based applications and application servers, as well as a role-based DBS. The communication between applications and the DBS is ensured by a proper role query language.

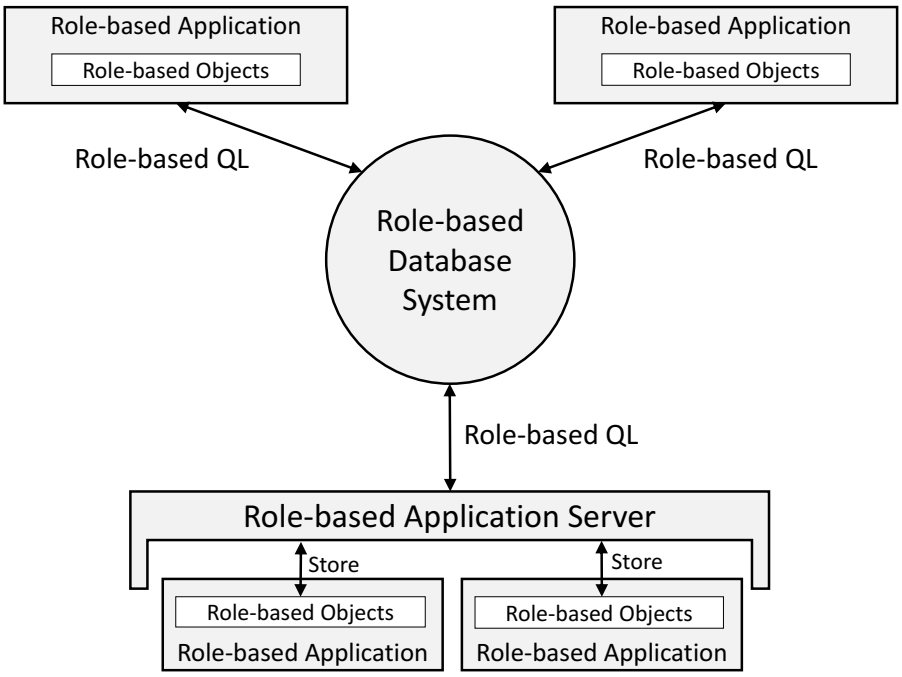


Figure 3.16: Role-based Software System Layout utilizing a Role-based DBS

There exist only one approach that introduces roles and the associated metatype distinction as first class citizen in a DBMS. This approach is the Information Networking Model (INM), which introduces a conceptual data model that has been implemented into a database [61, 62, 17]. It has been proposed by Liu et al. in 2009 and is designed to overcome traditional role model issues like their inability to describe context-dependent information [61]. Basically, they assume a very simple role model that can be categorized into the behavioral and structural class (see Section 2.2). As modeling elements they employ classes and role-relationships. The former one is the equivalent to traditional object-oriented classes, while the latter one represent special behavior and structure in relationship to other classes. Additionally, classes and role relationships may form individual hierarchies[44]. In contrast to ORM, there is no large community supporting and further developing this approach. An example INM model is illustrated in Figure 3.17.

This scenario describes a university domain with professors, students, and classes in general. Context-dependent information is modeled either as context-dependent relationship or context-dependent attribute. The first one is used between two role relationships or between a role relationship and a class,

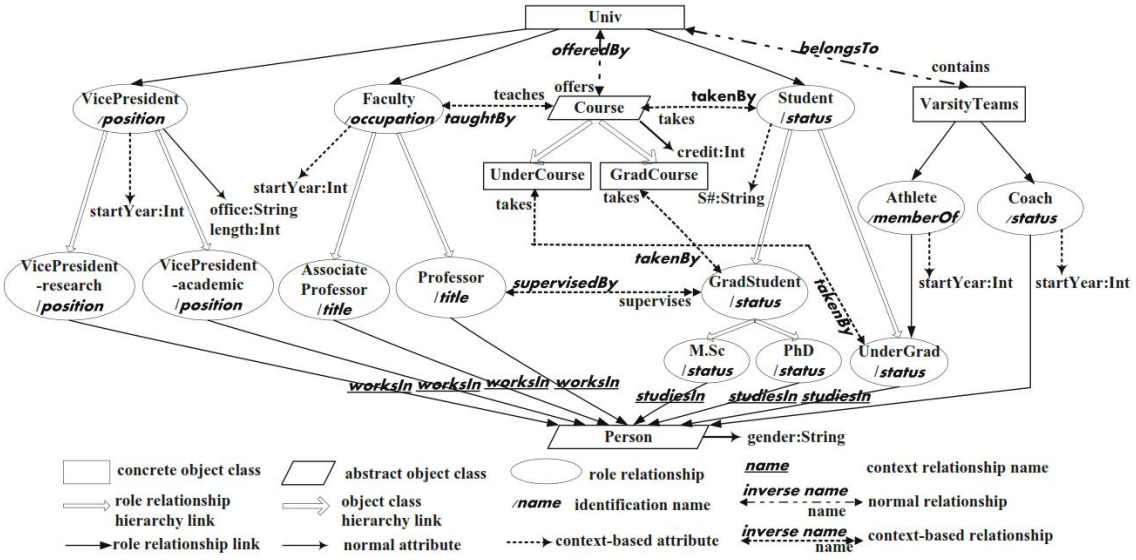


Figure 3.17: University Domain Modeled in INM; Extracted from [61]

indicating only if this particular role relationship is active. The second one describes attributes that are only valid in this role relationship. For example, the *startYear* of the *VicePresident*. However, based on this model description, they induce various other elements to the model, to represent them in an object-oriented way such that INM classes are represented as traditional classes and role-relationships as classes in combination with an *isa* association [44]. Additionally, they create an inheritance hierarchy for the classes based on the role relationships.

A context in their terms only exists virtually depending on the direction the role relationship is constructed. For instance, the **Univ** in Figure 3.17 is seen as context for the role relationships *VicePresident* and *Faculty*, but is modeled and implemented as class. Thus, a direct distinction between context (or rather compartment) and classes is not provided by this data model. Additionally, there can be only one player type per role relationship.

The INM is implemented on top of a key-value store featuring an XPath-like query language called INM Query Language (IQL) [43, 17]. The database implementation supports schema and instance creation based on the INM. On the physical level, the instances are stored as key-value pairs with respect to the schema. However, because role relationships are modeled as classes, they are stored separately in individual values resulting in a fragmented entity [17]. The query language features sublanguages for creating the schema and instances as well as querying INM-like stored data. In general, the IQL consists of two parts, one defining the path and hierarchical structure that has to be queried and a construction part, stating how to represent this information in the result.

```
1 query person $x//worksIn:$y//title:Prof
2 construct person:$x//worksIn:$y
```

Figure 3.18: Example IQL Query Illustrating the Query and Construction part; According to [43]

An example query is given in Figure 3.18. This example searches for all persons $\$x$ who work in $\$y$ as Prof. Additionally, the result displays each person $\$x$ and each university $\$y$, $\$x$ is working in. Unfortunately, there is no explanation about the result representation, thus, it remains unclear whether the database provides instantiated objects to an application based on the result construction rules or the tree-like structure.

To summarize, INM is the most mature approach to enable dynamic and evolving entities in a DBMS. From the data model perspective it suffers from various weaknesses. At first, the data model does not consider compartments explicitly, in fact the compartment notion is implied by role relationships. Secondly, role relationships are represented as separate classes and there is no integrated structure that brings classes and role relationships together. Additionally, they derive induced class structures to represent the role relationships resulting in a hierarchically structured data model. Moreover, the data model misses an operator description, so it remains unclear which operations can be performed within the data model. Beside their query language IQL, a result representation including navigation or iteration functionalities is completely missing. All in all, the INM approach is incomplete and the data model does provide a Compartment Type metatype discrimination.

3.4.5 Discussion

Conceptual modeling and programming languages nowadays implement roles to separate entity information into several metatypes for flexibility and complexity reduction reasons in a software system. In contrast, DBSs have not been adapted to the new requirements of such systems resulting in the role-relational impedance mismatch. To overcome this mismatch and the accompanied problems several architectural approach can be employed. Each approach aims at different goals and solves different problems, but in general, no approach totally complies to all requirements to overcome the role-relational impedance mismatch. An evaluation of related approaches with respect to the requirements posed to a fully integrated role-based and context-sensitive DBS is presented in Table 3.1. For the ranking three options exist: a □ indicating the requirement is not fulfilled, a ⊞ referencing for a partially fulfilled requirement, and ■ for a fulfilled requirement.

Requirement	Database Model	Query Language	Result Representation
SQL Views	□	□	□
DAMPF	□	□	□
ConQuer	□	■	□
DOOR / Fibonacci	⊞	□	□
Information Networking Model	⊞	■	⊞

■: yes, ⊞: partial, □: no

Table 3.1: Evaluation of Related Approaches to Overcome the Role-relational Impedance Mismatch

This evaluation is based on related approaches' ability to represent the metatype differentiation between Natural Types, Role Types, Compartment Types, and Relationship Types in combination with its constraints on the schema and instance level. The focus on CROM is due to its superiority over other role-based data models by supporting all three aspects of role-based modeling (see Section 2.3). The first architectural approach uses traditional and existing techniques in terms of external views in relational DBS. This class does not comply to any requirement, in fact they cover complex joins, but the mapping and the corresponding semantics reconstruction have to be implemented by the application developers individually. Client side mapping engines, like DAMPF, that are implemented in the applications, try to overcome the manual mapping and reconstruction overhead by providing a standard abstraction layer that pushes all mapping related functionalities into this layer. This simplifies the application development process, but neglects the database system issues and software system problems, thus, the role-relational impedance mismatch remains present. Consequently, no requirement is fulfilled. Database system side mapping engines provide an application-oriented query language that is transformed into the database query language, like SQL, in the background. In contrast to their client side opponents, they provide a role-based query language and in the best case

a proper result representation. The representative of this class, ConQuer, does not provide a result discussion, thus the requirement is not fulfilled. Persistent programming languages for role-based software systems, like DOOR and Fibonacci, integrate database abstraction mechanisms into a programming language and couple the database system tightly to the application. The representatives of this class feature a role-based data model, but without considering Compartment Types. A query language and result representation is obsolete due to the tight integration of the programming language and database, in fact results in terms of well-known relational results do not occur in such systems. Rather, the role-based objects are directly serialized on disk for persistence purposes. Finally, the INM is a representative for a DBMS integration of a role-based data model. INM has weaknesses in its data model, because it is not CROM compliant, and the result representation in terms of missing iteration or navigation options. Moreover, an integrated data structure enabling a separation of concerns is missing. However, it features a sophisticated query language including a construction part to restructure stored data structures for the client side representation.

As can be seen, there is no fully integrated approach available that overcomes the role-relational impedance mismatch that is posed by CROM-based applications and domain models. In the ideal situation, all requirements have to be fulfilled. This would keep the application developers out of the data management, enable role data model optimizations on the basis of special operators, give applications the opportunity for ad hoc queries, and finally represent the results as instance of the role-based data model.

3.5 OVERVIEW OF RSQL

To overcome the shortcomings of related approaches and provide a DBS that is able to represent a sophisticated metatype distinction and its corresponding constraints on the schema and instance level, a novel approach is required. In particular, this approach has to fulfill the requirements discussed in Section 3.3. Additionally, the aimed approach has to be independent of applications and their programming languages to ensure DBS independence and a loose coupling between the DBS and the applications. In detail, we aim for a traditional software architecture that has a strict separation between the transient application world and persistent database world. Consequently, the application handles its data structures by its means and the DBS by its individual interpretation, but both rely on a role-based notion of the entities. Thus, the goal is to keep the well-known and often applied software architecture, but overcome the role-relational impedance mismatch by introducing a CROM-based database model in combination with a sophisticated query language and result representation.

To achieve this goal, we are going to introduce the RSQL approach, consisting of a CROM-based database model, a query language, and a result representation on this database model basis. RSQL can be classified into the lastly discussed group of approaches, the group for a full DBS integration of role-based semantics. In detail, the RSQL data model defines the basic structures the novel DBS will handle as well as operators on this model. Thus, the RSQL database model is a logical database model. On the schema level this data model features Dynamic Data Types (DDTs) that build the structural integrated unit. A DDT consist of a player type and several Role Types that are separated in two dimensions, depending on whether they are played or featured. On the instance level Dynamic Tuples (DTs) represent the instances that can be extended or shrunk dynamically during runtime, but only within the DDT boundaries. Furthermore, the RSQL query language is based on the RSQL data model and extends well-known query mechanics by config-expressions. This expression acts as schema filter for Dynamic Tuples of the same Dynamic Data Type but having different schemata. Finally, the RSQL Result Representation (RuN) provides an RSQL data model instance to applications and users, and defines functionalities to navigate between Dynamic Tuples of different Types as well as iterate over ones of the same Type.

Thus, the RSQL approach integrates several components for a CROM-based DBS in one approach and addresses a database system's data system and set-oriented interface. In detail, the RSQL data model is discussed in Chapter 4. Additionally, operators on the basis of this data model are defined. The explanations on the RSQL query language, in particular its syntax and the relation to the database operators, are the topic of Chapter 5. Moreover, this chapter includes a discussion on processing RSQL queries and the result representation.

3.6 SUMMARY

In the beginning, we outline the software system we consider and the place a DBS has within such a software system. Based on the trend of implementing role-based features to cope with complexity and context-sensitivity issues of modern software system, we explained the shift toward role-based modeling and programming languages. Using role-based semantics and features on the level of applications, but not on the database level causes the role-relational impedance mismatch that is mainly based on the missing metatype distinction in the DBS. Problems resulting from the role-relational impedance mismatch can be categorized into three categories. These are problems for the applications and the application developers, pains for the DBS itself, and issues for the software system in general. Based on this problem discussion, we defined requirements to be fulfilled by a DBS to overcome the discussed mismatch. This is followed by a related work discussion that takes these requirements into consideration for evaluating these approaches. In general, the related approaches can be classified by their aimed software architecture layout. We came to the conclusion that no approach provides full role support, especially Compartment Types, as a notion of context, are always neglected. Based on this and to overcome the shortcomings of the related approaches, we outlined the RSQL approach consisting of a logical database model, a role-based query language and result representation supporting CROM metatypes and their inherent constraints on the schema and instance level.

4

RSQL DATABASE MODEL

To integrate the concept of roles as first class citizen in a database system, a proper database model, supporting the notion of roles and a metatype distinction within an entity, is inalienable. At first, several requirements for such a database model are defined and explained in Section 4.1. These requirements build the basis for evaluating related approaches, in the second place. As it will be seen in our related work discussion, none of the related approaches provides a data model that is able to fulfill the entire list of requirements and none has an explicit notion of a context, respectively Compartment. Consequently, a novel database model named RSQL database model is defined. The main body of these definitions are based on the data model descriptions in [50, 49, 51] as well as the CROM definitions in [57]. This features the notion of Dynamic Data Types on the type level, to express which entity types can fill or contain which particular Role Types. On the instance layer, an entity is represented by a Dynamic Tuple. This allows for the representation of entities having a common core but different structure within one type definition. Additionally, it enables to change an entity's structure during runtime without changing its overall type. Furthermore, Dynamic Data Types and Dynamic Tuples overlap in certain Role Types and Roles, respectively. This ensures the embedding of actually played Roles in a Compartment. However, these definitions build the fundamental and integrated data structure the database model operators are defined on. This builds the mathematical foundation to process and filter the information an application is asking for in queries. These operations range from selecting Dynamic Tuples based on their structure, over the selection of overlapping Dynamic Tuples and attribute focused filters, to intersecting and uniting sets of Dynamic Tuples. All definitions regarding the operators are explained and discussed after the database model definitions. A short conclusion and summary completes this chapter.

4.1 REQUIREMENTS

To overcome the problems outlined in Section 3.2 a proper data model that supports metatype distinction on the foundation of CROM is required. An overview of the requirements is presented in Table 4.1. CROM builds this basis, because it is the only metamodel that is able to represent all three trends in role modeling, as shown and discussed in Section 2.2 and Section 2.3. Consequently, the first three requirements, DM.1 – DM.3, address the metatype distinction between Natural Types, Role Types, Compartment Types and Relationship Types. An explicit notion of Natural Types is omitted as requirement, because they build the static core of an entity type, which is shared by all approaches as basis abstraction. Role Types required to express the core type's ability to extend its structure dynamically during runtime, without changing the type or reinstantiation. Compartment Types are required to represent the context-dependent nature of roles. Additionally, an explicit notion of Relationship Types is mandatory to avoid relationship mappings onto the logical data model, like it is done for relational DBS.

Moreover, it is required that Role Types are embedded into a Compartment Type (DM.4). This ensures context-dependency in the way that the Compartment Types act as objectified context for each Role Type. Thus, Role Types outside a Compartment Type have to be prohibited. In addition to the aforementioned metatype distinction, multiple player types for one Role Type are required (DM.5). For instance, the **StudiesCourse** Role Type, as shown in Figure 2.3 is playable by both Natural Types, **Lecture** and **Seminar**. This avoids redundant Role Type definitions having the same behavior and structure for multiple player types. Furthermore, it is required that an entity can play Roles of the same Role Type multiple times simultaneously (DM.6). This requirement is motivated from an entity's structural point of view. Basically, it is natural that the same structure is acquired multiple times, for instance, a **Person** can be a **Student** multiple times at different **Universities**, or a **Person** may be a **Member** of various **SportsTeams**. Hence, the **Student_ID** and **Member_ID** attributes have to be present multiple times as well. Allowing a player to play a role of a certain Role Type only once,

Requirement	Description
DM.1	Explicit notion of Roles
DM.2	Explicit notion of Compartments
DM.3	Explicit notion of Relationships
DM.4	Roles are only valid in combination with a certain Compartment
DM.5	Multiple player types per Role Type
DM.6	Multiple Roles of the same Role Type simultaneously
DM.7	Integrated data structure for dynamic and schema flexible types and entities
DM.8	Operator definitions
DM.9	Mathematical closure

Table 4.1: Overview of Requirements Posed to a Role-based Data Model

avoids entities to carry the same structural information, but with probably different values, several times. Hence, players must be able to play Role of the same Role Type multiple times simultaneously.

Furthermore, we require an integrated data structure that unites the semantics of the metatype distinction and most importantly their interrelations (DM.7). Basically, roles describe a structural extension of an entity and should be treated as such an extension. Consequently, they form an integrated logical unit. A pure metatype distinction between Natural Types and Role Types cannot preserve this perception of a dynamically growing and shrinking entity as integrated unit. Rather, the semantic interrelations between the metatypes have to be integrated as well. This can be done within a higher data structure that combines the entity's core and its extensions within a separate higher level type. For instance, imagine the Natural Types **Person** and **Lecture** and the Role Type **Student**. Based on this information, it does not become clear which Natural Type can play which Role Type. Basically, this integrated data structure defines on which logical data structure perception the database system will work on. Relational database systems work on the notion of tables and in contrast to this a role-based database system has to work based on a different but well-founded notion as well. In sum, the integrated data structure is utilized to preserve the semantic interrelations between the different metatypes and defines the way the database system operates on the data.

Finally, it is essential to provide formal operators that describe possible operations on the data model (DM.8) as well as a mathematical closure (DM.9). The formal operators enable the database management system to mathematically describe the parts of a query and to logically optimize a query plan. Moreover, the mathematical closure ensures that query (intermediate) results are an instance of the data model as well, and any operator does not produce an output, which is not covered by the data model.

4.2 RELATED WORK

During the past years the research community proposed several data models to represent data and their interrelations in a DBS. The following discussion is focused on database models only and neither discusses various conceptual or programming language role models nor their advantages and disadvantages. For a survey on role models under certain perspectives, like conceptual modeling, and non-persistent role-based programming languages, we refer to Steimann [79, 78] and Kühn et al. [58].

4.2.1 The Role Concept in Data Models

One of the first data models featuring roles as first class citizen is proposed by Bachman and Daya as extension to the network model [8]. Their initial observation was the "*most conventional file records and relational file n-tuples are role oriented*" [8, p. 465]. Generally, their data model consists of entity types, which represent the static aspects of an entity, and role types that describe dynamic aspects of entity types. In detail, an entity type is characterized by its playable role types, in contrast, role types may be playable by several entity types. For instance, the role type **StudiesCourse** might be playable by both, the **Lecture** and **Seminar**. In contrast, a **Person** might become a **Student**, or **StudentAssistant**, or both.

Entity types are represented in records and role types in role-segments, each having their own data items consisting of a name and a value. Within the record description, several role-segments may be referenced, but a role-segment may be referenced in several record descriptions. Moreover, a record occurrence (entity) is a vehicle for one or more role-segment occurrences (roles). However, a role-segment can appear only once within in record description and describes a single role-segment occurrence only. Thus, in their data model a role can be played only once [80].

Role types have several attributes in this data model. First, they can be shareable or non-shareable, indicating whether a role type may occur in several record descriptions or only in one. Secondly, role types may be attributed with essential or non-essential. The former implies that a record occurrence indicates a role occurrence as well, for instance, in relationships having a cardinality constraint larger than 1. The latter one describes a role type that might occur but does not have to. A conceptual summary of this data model is illustrated in Figure 4.1.

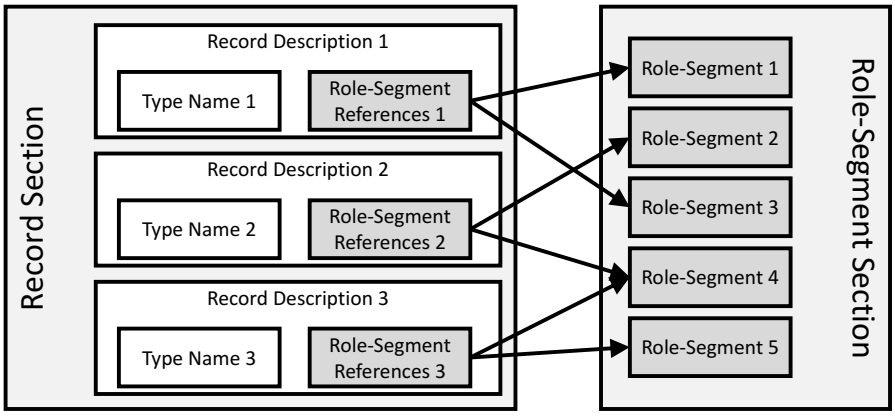


Figure 4.1: Bachman and Daya's Role Data Model; According to Descriptions in [8]

Additionally, Bachman and Daya's data model definition does not feature a notion of context, but binary owner / member relationships inherited from the network model [8, p. 469]. Hence, they provide a metatype distinction between two metatypes, the entity types and roles types. However, their description provides an integrated data structure that brings the static core component and played roles together. Additionally, roles are acquired when an entity joins a certain relationship, hence, roles are only present in case the entity participates to (conceptual) relationships; in fact, relationship types are not part of their data model. Moreover, they textually define only basic operations on the data model for manipulating record and role occurrences, but no formal operators. Consequently, operators are undefined, but for the basic operations they defined the mathematical closure is guaranteed. Over recent decades, this data model has been honored to be the first proposing roles, but further development was unfortunately not granted to it. In sum, Bachman and Daya's data model suffice the requirements DM.1, DM.5, and DM.7 as well as DM.8 and DM.9 partially.

4.2.2 Object Role Modeling

The Object Role Modeling (ORM) approach is a fact-based attribute free conceptual modeling language[1]. ORM was proposed by Halpin in 1998 [34] and revised and extended in 2005 [35]. The main difference between traditional modeling languages, like UML or ERM, is that entities are modeled attribute free. In fact, attributes are modeled in a separate type and related to entity types by predicates. Hence, ORM is strongly relationship focused providing the modeler the freedom to evolve attributes and entities independently from each other.

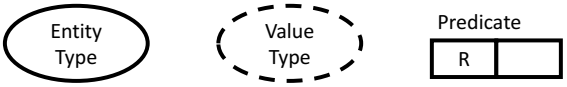


Figure 4.2: Basic Modeling Elements of ORM; According to Definitions in [34]

The graphical notation of ORM is depicted in Figure 4.2 according to [34]. Entity types are represented as ellipses with a continuous line, value types as ellipses as well, but with dashed lines. Additionally, n-ary predicates are depicted as compound boxes, one box per participating entity or value type. The term predicate in Halpin's sense is a special interpretation of a relationship. Finally, roles are the named places in predicates and specify the role a certain entity is playing in this particular predicate. Furthermore, ORM provides a wide variety on constraints, thus, modelers are able to design their domain very fine-grained and catch very special situations separately.

An example model of a simple university domain is illustrated in Figure 4.3. As you can see, **Person**, **Student**, and **University** are modeled as entity types and the person's **Name** as well as the **StudentID** as value types. All types are related by predicates; between **Person** and **Student** there exists a **is a** relation, between **Student** and **University** a **studies at**, and between the entity types and values types a **has** relation.

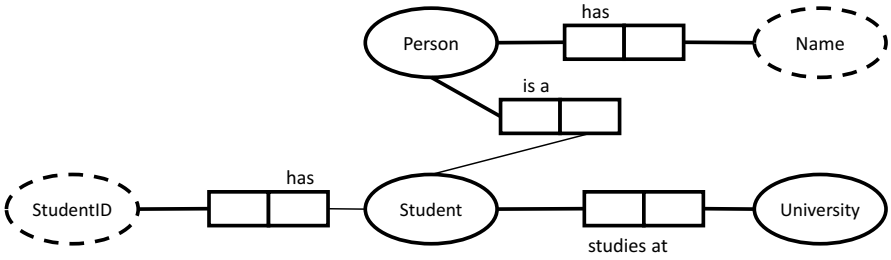


Figure 4.3: Example ORM Model of a Simple University Domain

As mentioned in the ORM's graphical notation description, roles are named places in relations only. Moreover, there are explicitly not used to temporarily classify entities [78], like the most approaches do. Thus, the ORM data model is very weak in terms of using roles for a separation of concerns within an entity. In contrast, modeling each attribute individually using a value type and relate them by predicates to entity types is somehow the most radical separation of concerns. Nevertheless, entity information that originally belong together, like a person is a student at a certain university, is distributed over several individual entity types.

As the example clearly shows, the usage of roles is very limited, because as soon as a role type needs to be attributed with some values, the role type is modeled as entity type, since there is no option to

[1]We outline this approach in the data model section, because this basic knowledge is required for the understanding of the conceptual query language ConQuer, which is discussed in Section 5.2

attribute predicates. Thus, the ORM tends to model entity information as ERM [18] does, but with a slightly different focus on relationships and attributes. Consequently, ORM provides a weak notion of roles and an explicit perception on relationships, but any other requirement is not satisfied. Hence, DM.3 and DM.6 are satisfied while DM.1 is partially fulfilled. All other requirements are not fulfilled at all.

4.2.3 DOOR

The Dynamic Object-Oriented database programming language with Roles, short DOOR, is, as the name directly implies, a programming language for databases [84, 86, 87]. Precisely, DOOR is a role extension to the object-oriented paradigm that also features persistent data objects. In detail, DOOR supports operations for dynamic role bindings at runtime and modeling context-dependent behavior [86]. Basically, this approach distinguishes between object classes and role classes. The former are equivalent to classes in traditional object-oriented languages. The latter one describes a set of similar roles that need to be connected to a player by the *played by* link.

As aforementioned, the DOOR data model features objects and roles that are connected by a played-by relation. This relation explicitly includes roles playing roles, such that played-by can also be established between two roles. Moreover, role classes are always connected to exactly one player type, which means shareable role types are not allowed in this data model. Hence, DOOR data structures are always trees having an object class at their root. Thus, the transitive closure of a role class has to end at an object class. However, the data model can exist of more than one tree, so that multiple objects can be specified in a schema. A sample schema of a DOOR model is illustrated in Figure 4.4.

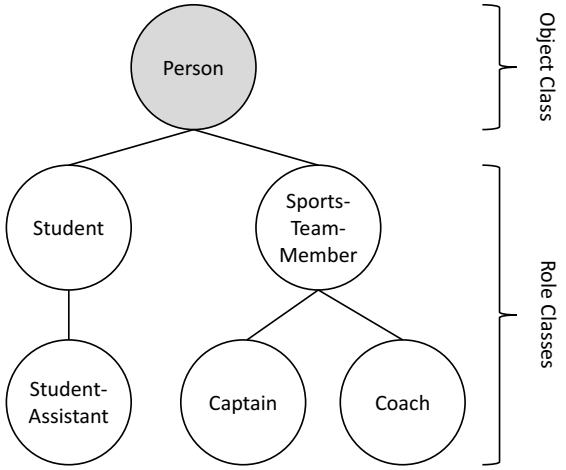


Figure 4.4: Example DOOR Data Structure of a Simple University Domain

This illustration contains a **Person** as object class as root of the tree. Furthermore, this **Person** can play roles of the role class **Student** and **SportsTeamMember**, respectively. Additionally, the **Student** role class can play role of the class **StudentAssistant** whereas **SportsTeamMember** roles can play roles of the classes **Captain** and **Coach**.

On the instance level, DOOR describes objects and roles as instances of their corresponding classes. Roles of the same class can be played several times simultaneously by the same object. Hence, multiple roles of the role class Student may exist concurrently and are played by the same object. The resulting data structure is a strictly hierarchical tree and each role is nested in its corresponding player.

The DOOR authors explicitly mention context-dependency of their data model. This statement subsides to a context-dependent access of attributes and methods. In fact, roles are not situated in any context, rather the way you can access roles is seen as context-dependency in their data model. In detail, accessing roles is done by the delimitation mark *!*, which means *Person!Student* accesses the **Student** roles of a **Person**. However, in their example illustrated in [86, p. 407] they define an attribute *id* on both, the **Person** and the **Student**, implying that *Person.id* is different from *Person!Student.id*. The context-dependency they are describing in this example is simply resolving the name of the attribute correctly. Actually, the *id* attribute defined on the Student is conceptually a different one than this one defined on the Person. The first one is the student *id* you will get when enrolling at a university, whereas the second refers to the person's *id*, like a social security number. Since both *id* attributes refer to a different (mental) concept, they should be clearly distinguishable, for instance by naming these attributes differently[2]. Instead of calling it context-dependency, we would call it class-specific attributes, which is not a problem at all, because they access the attribute directly and not indirectly.

Caused by their data model, they run into several referencing problems when objects or roles are deleted from the system. For instance, a **Student** role is discarded from the system, it is not clearly stated what happens to the **StudentAssistant** roles connected to this particular role. They are aware of such problems as [84] describes, but those were never addressed by the authors. However, DOOR has been developed for persistent role-based dynamic objects. Hence, the objects are persistently stored, but not in full fledged DBS. Rather, they implemented a storage engine by themselves. This is based on serialization as binary data stream of objects using metaobject protocol in a lisp-like language [87]. Hence, real DBMS features like concurrent data access control are not guaranteed. Another problem occurs when multiple roles of the same type are played and the data access does not access a certain role, rather a set of roles. In such a case, the responding role is randomly chosen, resulting in a nondeterministic result, as Baumgart describes [11, p. 25].

In sum, the DOOR approach satisfies the requirements DM.1, DM.6, DM.7, DM.8, and DM.9. However, the notions of compartments and relationships are completely missing. Additionally, role classes can be related to one player type only, which results in circumstantial subclassing for roles and multiple implementations of the same behavior and structure. Moreover, the DOOR approach is designed as persistent programming language not guaranteeing typical DBMS specific features like concurrent data access or fault-tolerance. Unfortunately, DOOR has not been maintained and further developed since the late 1990s and there is no source code available on the Internet.

4.2.4 Fibonacci

Like DOOR, Fibonacci is an object-oriented database programming language featuring roles to dynamically change an object's type during runtime [4, 2]. The language is strongly influenced by the Galileo language, but extends it by roles [3, 4]. In general, Fibonacci manages objects and an object internal graph of roles. Additionally, the object has a name, an immutable ID and a mutable state. Moreover, objects are accessed by their roles only, thus, the object itself is only the container for the roles. All methods of an object are defined by the roles it is currently playing. An example of Fibonacci's data structure is illustrated in Figure 4.5.

As you can see, the **Person** is specified twice, once as object type and once as role type. This is caused by the object type definition, that does not feature any attributes or methods. These are defined in the role types only. Thus, an object's behavior and structure is defined by role types. Furthermore, role types can play roles types, resulting in a hierarchical structure of object types and their playable role

[2] In their terms, even relational tables would be context-dependent when accessing columns by their qualifying table in a result set (s.id vs. p.id), in case attribute names are non-unique.

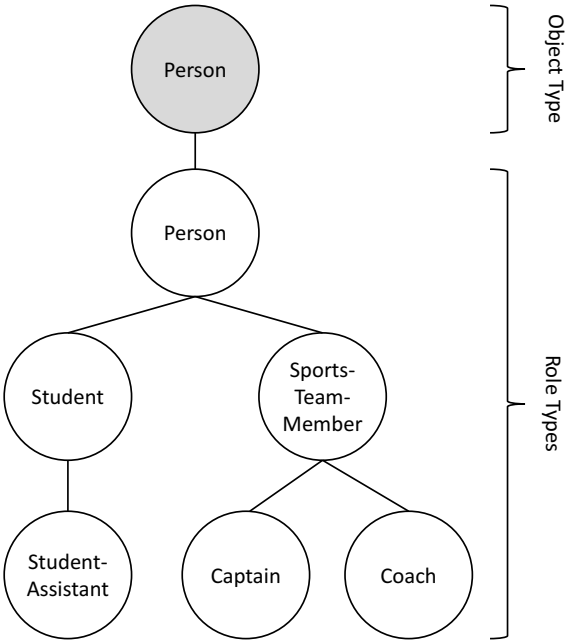


Figure 4.5: Example Fibonacci Data Structure of a Simple University Domain

types. Internally, role types are represented as acyclic graph, enabling multi-inheritance within this hierarchy. This multi-inheritance is different to DOOR, which does not allow this feature. However, a role type is bound to a certain hierarchy, thus, multiple player in terms of objects are not supported. Nevertheless, the data model of Fibonacci and DOOR are very similar. A formal foundation of this data model is presented in [25].

At runtime, an object may acquire or abandon a role of a certain role type dynamically, resulting in additional and less behavior, respectively. A role type can be instantiated only once within a certain object. Furthermore, role types provide only a method signature, thus, at instantiation time the actual implementation has to be provided. This can be done manually by the application developer, or by a constructor that implements the same behavior for each role. This approach of providing only method signatures, also known as interfaces, is unusual, because a role is the instance of a certain role type in a certain object. Thus, you would naturally assume that roles of the same type realize the behavior the same way, but in Fibonacci the implementation can vary from role to role, giving you inconsistent behavior within the same type. However, messages or methods are passed through the object's internal graph to find the corresponding attributes and methods. There exist two options to search for these: (i) upward lookup and (ii) double lookup. The first searches for methods in the role itself, if the method is not defined in this role, it continues the search in the super-roles. Due to the static typing, it is ensured that at least one super-role implements this method. The second option searches in the sup-roles at first. In case of a negative result, the upward lookup is performed.

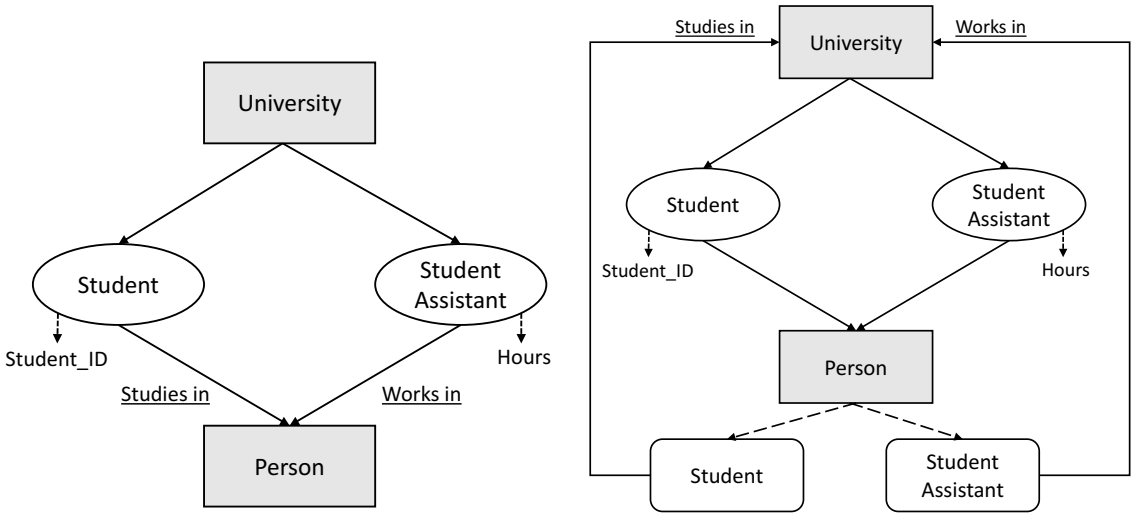
The data store to persistently store the data objects of Fibonacci is very limited in its functionality. Basically, they store all object and role information as uninterpreted byte string, and reference to other data objects. For fault tolerance and recovery they use *frozen versions*, a kind of snapshots. As the authors clearly state, the required set of operators is very minimal, to test and compare different storage engines [4]. Thus, traditional DBMS features are not supported, rather only file-based storage is employed. Moreover, the query language provides rudimental functionalities, like put and get, only.

In sum, Fibonacci is similar to DOOR and fulfills the requirements, DM.1, DM.6, DM.7, and DM.9. The operators are very basic, thus, they are rated as partially fulfilled. In contrast to DOOR, roles of

a particular role type can be played only once per object. Additionally, the object type is designed as container unable to represent any attributes or methods, which results in specifying the same (logical) object twice when the core has to have attributes or behavior. Finally, it is a persistent database programming language not featuring any usual DBMS functionalities, thus, the application and data storage become an integrated and inseparable layer avoiding multi application scenarios and user specific ad hoc queries.

4.2.5 Information Networking Model

Information Networking Model (INM) has been designed to overcome problems of traditional and role modeling languages [61, p. 132]. Interestingly, they assume very basic role models that do not have any context notion and can be categorized into the structural and behavioral category (see 2.2). However, this approach has been proposed in 2009 by Liu et al. [61] and is based on their previous work on context-dependent relationships [62, 44]. As basic modeling elements INM uses object classes and role relationships. An object class is equivalent to a traditional class in object-oriented systems. It simply describes the static information about an entity. Role relationships specify an entity in a certain relationship by associating a role to it and are directed from a source class to a target class. These relationships are also utilized to represent context-dependent properties, like attributes or behavior. Additionally, all role relationships are directed and may form inverse role relationships. Moreover, role relationships and object classes may form individual inheritance hierarchies. A small example INM model is presented in Figure 4.6(a).



(a) INM Model (b) Induced INM Model

Figure 4.6: Example INM Model of a Simple University Domain; according to [61]

This example consists of two classes, a **University** and a **Person** class. Both are connected by the two role relationships **Student** and **StudentAssistant**. Interestingly, the role relationships are directed from the university to the person and not the other way around, which is confusing. Naturally, the role in a role relationship details an entity that plays this particular role, but in INM it is graphically expressed the other way around by directing the role relationship link from the university to the student and student assistant. In fact, the person class can play roles of the type student or student assistant and not the university class. To get the information which role can be played by a class, the inverse context relationship has to be used. In the small example model this is indicated by the underlined annotations on the role relationship links from student and student assistant to the person.

```
Student, StudentAssistant John [
  @Birthday: 12.01.93
  StudiesIn:{
    TUD[@StudentID: 12345],
    TUC[@StudentID: 55874]
  },
  WorksIn:{
    TUD[@Hours:15]
  }]
Student Tim [ ... ]
```

Figure 4.7: Example Instance Representation in INM; according to [62, p. 724]

Furthermore, this example shows context-dependent attributes connected to role relationships, especially the Student_ID for the student and the hours attribute for the student assistant. These can only be populated, if a class participates in this role relationship.

However, the core of their models is based on object-oriented notions of entities. Classes and role relationships form an induced object-oriented class hierarchy, in which each role relationship, connected to another class, builds a subclass of the target class [61, p. 138]. Hence, each role relationship is represented as class itself, although it is connected to another class by an *is a* association. An example of such an induced model is depicted in Figure 4.6(b). As it can be seen, the role relationships for separate role relationship classes under the class person. Additionally, the have a context-relationship to the university. In contrast to the original model, the classes that can play a role in a role relationship are denoted more naturally. However, a role relationship can be related only once, which avoids multiple player types for a role relationship.

The resulting data structure forms a tree having a class as root. It nests all information from a class and its subclasses into a hierarchical structure. Consequently, the instances of such a model are structured hierarchically as well. At runtime they use a multi typing mechanism to weave all information into an entity. Moreover, this multi typing defines which role relationship dependent attributes an entity can have. However, all attributes of subclasses, like the student or student assistant, are combines in the entity using object references to specify the context these attributes are valid in.

Figure 4.7 illustrates the instance *John* studying at two universities in detail. As it can be seen, each instance forms a separate hierarchical structure using the class and subclass hierarchies defined on the type level. However, this structure is directed from the class to subclasses by referencing other objects, like *TUD* and *TUC*. These referenced instances are defined separately, but do not hold any information on the students studying at these universities, because the subclasses are defined for the person and not for the university. This results in distributed information about the students from a university perspective, because each student has to be scanned to collect the information whether he or she is a student at a particular university.

In [43] the authors present four ways of evaluating INM queries on a INM database. All options rely on producing a graph of connected information and evaluate this graph for the query. In detail, they describe a forward chaining search, a backward chaining search, a hybrid search, and a multi expression search. Each has its advantages for a special type of queries. Moreover, their discussions on the INM query processing suggest an implementation specific operator definition, which is tailored to their individual physical data representation [43, p. 532]. An implementation independent description on the logical level would be beneficial to define various physical operator implementations, as known from the relational database world. However, there are no formal operators defined, in fact, they describe their evaluation strategy textually.

In total, the Information Networking Model implements roles in combination with relationships and puts both notions into one meta type, call role relationship. Those are related from one class to another to represent relationship and context-dependent information. An explicit notion of context is still missing, because it is implicitly induced by following the direction of a role relationship. These roles are represented as subclasses of traditional classes, but instances can join and leave these subclasses dynamically. Hence, the requirement DM.1 is fulfilled. DM.3 is partially fulfilled, because the notion of a relation ship is mixed with roles. DM.4 and DM.5 are not fulfilled, because there is no explicit compartment or context notion and role relationships can be related only once. However, a role relationship can be instantiated multiple times for one class, hence, roles are playable multiple times and the requirement DM.6 is satisfied. Moreover, they provide a tree-based structure that combines all entity relevant information within an integrated data structure, but only from one direction. Especially for the class acting as virtual context the information which role relationships are active is unknown. Nevertheless, they provide a data structure that combines an entity with its roles, so DM.7 is fulfilled. DM.8 and DM.9 are not fulfilled, because they describe the query processing textually and very implementation specific and do not define formal operators.

4.2.6 Discussion

The evaluation of related approach shows a consistent lack in the explicit notion of compartments or objectified contexts. Additionally, relationships as first class citizen are mentioned in two approaches only. Moreover, the embedding of roles in a context, no matter what kind of context perception the particular approach relies on, is missing in each approach. Rather, roles are used to characterize one entity only, but not the entity it is embedded in. This avoids an actual context-dependent representation of roles. Furthermore, it can be concluded that each approach defines its notion of roles a bit differently, which supports the statement of Section 2.2, especially the statement that there is a zoo of role notions and the overall valid notion of a role does not exist. In one approach role types can have multiple player types and in another one a role type can be instantiated once per entity only. An integrated data structure that combines the entity itself and the role it plays, is provided by almost each of the related approaches. Only the Object Role Model does not satisfy this requirement. Finally, only one of the evaluated works provides a formal operator description, especially an implementation independent description. Some provide partial descriptions, but these are mostly very basic operators. This situation of missing operator descriptions prohibits query optimizations on a mathematical level. An overview of the related work evaluation in regard of the previously specified requirements is represented in Table 4.2.

Requirement	Bachman	ORM	DOOR	Fibonacci	INM
Notion of Roles	■	⊞	■	■	■
Notion of Compartments	□	□	□	□	□
Notion of Relationships	□	■	□	□	⊞
Roles in Compartments	□	□	□	□	□
Multiple Player Types	■	□	■	□	□
Multiple Roles simultaneously	□	■	□	■	■
Integrated Data Structure	■	□	■	■	■
Formal Operators	⊞	□	■	⊞	□
Mathematical Closure	⊞	□	■	■	□

■: yes, ⊞: partial, □: no

Table 4.2: Evaluation of Related Data Model Approaches

4.3 RSQL DATABASE MODEL

As the discussion on the related work clearly shows, none of the related approaches satisfies the complete list of requirements, especially the notions of a Compartment as objectified context is missing. To overcome the shortcomings of these data models, the RSQL database model is introduced and formally defined in this section. RSQL's database model comprises definitions on the schema level as well as on the instance level. An overview of the database model definitions and their interrelations are presented in Figure 4.8. Precisely, the schema level consists of definitions for a *Dynamic Data Type* (DDT), a *Configuration* and a *Relationship Type*. This instance level is represented by *Dynamic Tuples* (DT) and *Relationships*. A Dynamic Data Type combines the role-based semantics and metatype distinction within an integrated data structure and consists of a core and two sets of Role Types. The instance level opponent of a DDT is a Dynamic Tuple. A DT represents an entity that can dynamically change its structure during runtime. Different instances of the same DDT can have different schemata, depending on which Roles are currently played. Which structure can be added to an instance, is covered in Configurations. A Configuration describes a certain valid schema of its corresponding DDT. Hence, a DDT defines a space of valid Configurations and Dynamic Tuples can adapt their structure by changing their Configuration. This ensures that only this structure is added to an entity that is intended to be added. For instance, a **Person** can be extended by the structure defined in the **Student** Role Type, but not by structure specified in the **TournamentTeam**, because there is no valid Configuration covering this schema and the **TournamentTeam** is not part of the DDT **Person**.

Additionally, Relationship Types and Relationships are part of the database model definition to omit relationship mappings onto Dynamic Data Types. A Relationship Type connects two DDTs by two distinct Role Types. On the instance level, a Relationship connects two DT by two distinct Roles.

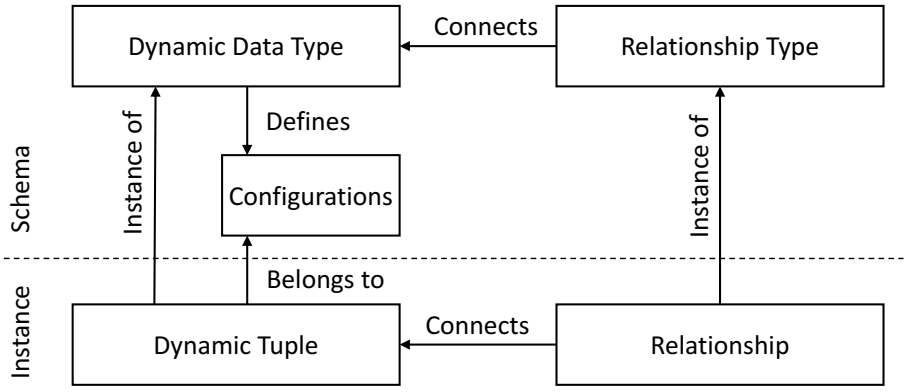


Figure 4.8: Overview of RSQL's Data Model Concepts

4.3.1 Schema Level

In database systems the type level defines the schema of a database and the interrelations of various schema elements. Before defining the database schema, we have to define the schema elements, the database system is able to represent. In general, the RSQL database model discriminates the four metatypes that are discriminated in the CROM metamodel, too [57]. These metatypes represent the schema elements of a role-based database system. Additionally, each metatype features a set of attributes. The corresponding type definitions are presented in Definition 1.

Definition 1 (Attributes). *Let nt, rt, and ct be the metatypes Natural Type, Role Type, and Compartment Type, respectively. Then, $t = \{A_1, \ldots, A_n\}$ with $n \in \mathbb{N}$ and $t \in (nt, rt, ct)$. Additionally, an attribute A is defined by a name and a domain, such that $A = (name : domain)$.*

This definition allows representing different metatypes and the corresponding attributes as schema elements of a database. Although, Relationship Types do not have any attributes, because attributed Relationship Types can be modeled as separate Compartment Type. To have a clear target for each metamodel element, attributes are prohibited for those types. The definition itself is comparable to the relational schema definition of a relation database system, but with the difference that multiple metatypes are distinguished in our definition. RSQL schema elements are specified by the following pattern illustrated by the Natural Type Person (see Figure 2.3).

$$Person : \{[Name : String, LastName : String, Birthday : Date, Address : String]\}$$

As in the relational world and also in the standard literature, the [] specify the attributes that all instances of corresponding type have to share [55]. For Natural Types this indicates the Natural constructor, for Role Types the Role constructor and so on. Additionally, the $\{\}$ indicate that a set of instances is related to this type and represent the set constructor.

So far, the database model differs from the traditional relational database model by discriminating four, instead of only one, metatype. This changes when the database schema is defined. The database schema definition adds role-based and metatype specific characteristics to the standard schema element's definitions. Please note, for a clear arrangement of the following definitions we omit attributes in them, but detail some examples by using attributes. However, it can be expected that each type has a certain number of attributes.

The schema definition specifies the logical schema elements in the DBMS and their interrelations. A relational database schema consists of relational tables, a graph data store of vertexes and edges, and a key-value store of simple key-value pairs. Consequently, a role-based database system consists of various types of roles. Roles cannot be considered in isolation, because they extend an entity and so the corresponding schema elements do. The following schema definition is based on the CROM type level description presented in [56]. Moreover, it is a combination of the schema specifications given by [51, 49].

Definition 2 (Schema). *Let NT, RT, CT, and RST be mutual disjoint sets of Natural Types, Role Types, Compartment Types, and Relationship Types, respectively. Moreover, let $Card \subset \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$ be the set of cardinalities represented as $i..j$ with $i \leq j$.*
Then, a Schema *is a tuple $\mathcal{S} = (NT, RT, CT, RST, fills, parts, rel, card)$ where:*

- *$fills \subseteq (NT \cup CT) \times RT$ is a relation,*
- *$parts : CT \to 2^{RT}$ is a function,*
- *$rel : RST \to (RT \times RT)$ is a function connecting two Role Types, and*
- *$card : RST \to (Card \times Card)$ is a total function assigning cardinality constraints.*

Additionally, we require the following axioms to hold for a well-formed schema.

$$\forall rt \in RT \; \exists t \in (NT \cup CT) : (t, rt) \in fills \tag{4.1}$$

$$\forall ct \in CT : parts(ct) \neq \emptyset \tag{4.2}$$

$$\forall rt \in RT \; \exists! ct \in CT : rt \in parts(ct) \tag{4.3}$$

$$\forall rst \in RST : rel(rst) = (rt_1, rt_2) \wedge rt_1 \neq rt_2 \tag{4.4}$$

$$\forall rst \in RST \; \exists ct \in CT : rel(rst) = (rt_1, rt_2) \; \wedge \; rt_1, rt_2 \in parts(ct) \tag{4.5}$$

In detail, the schema definition distinguishes between the four metatypes and collects the corresponding types into their respective sets. Additionally, a relation and three functions are defined to represent the semantic interrelations. At first, the *fills* relation declares a player type (either Natural Type or Compartment Type) able to fulfill a certain Role Type. Secondly, the *parts* function collects all Role Types contained in each Compartment Type. Thirdly, the *rel* function maps a Relationship Type to two Role Types, such that a Relationship Type is always binary and only exists between Role Types. Finally, *card* defines the cardinality constraints for each Relationship Type.

In addition, we constrain the model to the five axioms aforementioned. In detail, it is required, that each Role Type has at least one player type (4.1). This avoids isolated and non-playable Role Types in the database schema. For example, the **Student** Role Type in our university domain example has the player type **Person**. Without this player type the **Student** is not part of the *fills* relation and violates the first axiom.

Furthermore, we require Compartment Types to contain at least one Role Type, which means empty Compartment Types are prohibited in a valid role-based database schema (4.2). This constraint is necessary to distinguish Natural Types from Compartment Types. Empty Compartment Types would look like Natural Types, additionally Compartment Types are characterized as founded types that means they depend on other types. This foundedness is provided by the Role Types included in each Compartment Type. For instance, imagine the **University** Compartment Type as shown in Figure 2.3, but without any Role Type contained in it. It looks like a standard Natural Type and the *parts* function would return an empty set of contained Role Types for this Compartment Type.

The third constraint requires a Role Type to be part of exactly one Compartment Type (4.3). Consequently, this avoids Role Type assignments to multiple Compartment Types and ensures the membership of each Role Type in a Compartment Type. Thus, Role Types are semantical bi-founded. This means, they are founded in two directions; on the one hand they depend on a player type and on the other hand on a Compartment Type. For example, imagine the **Student** Role Type; it can neither exist without the **Person** nor without the **University**. This also ensures the availability of the **Student**-specific structure and behavior only within the corresponding **University** Compartment Type and not outside of that.

The fourth and fifth axioms constrain the Relationship Types. In detail, the fourth limitation avoids Relationship Types between the same Role Type (4.4). This constraint simplifies to determine the instances participating in such a Relationship Type by clearly stating which Role Type is the left part of this Relationship Type and which one is the right part. Additionally, the cardinality constrain mapping is simplified by this constraint. Finally, both related Role Types of a certain Relationship Type have to be part of the same Compartment Type (4.5). This ensures context-dependent Relationship Types only. Additionally, this reflects the intended semantics in which relationships can be established within a certain situation only and not over several Compartment Types. For instance, when two Role Types have to be related to each other, they obviously belong to the same Compartment Type, because the Role Types act in the same situation. To illustrate the schema definition in detail as well as the sets it contains and the relations it stores, a small example schema based on the university domain depicted in Figure 2.3 is created.

Example 1 (Schema). *Let $\mathcal{U} = (NT, RT, CT, RST, fills, parts, rel, card)$ be the model of the university domain (Figure 2.3), where the individual components are defined, as follows:*

$$
\begin{aligned}
NT &:= \{\mathit{Person}, \mathit{Seminar}, \mathit{Lecture}\} \\
RT &:= \{\mathit{Student}, \mathit{Professor}, \mathit{StudentAssistant}, \mathit{ResearchAssistant}, \mathit{StudiesCourse}, \mathit{TeamMember}, \\
&\quad \mathit{Coach}, \mathit{Captain}, \mathit{TournamentTeam}, \mathit{WinnerTeam}\} \\
CT &:= \{\mathit{University}, \mathit{SportsTeam}, \mathit{Tournament}\} \\
RST &:= \{\mathit{takes}, \mathit{teaches}, \mathit{supervises}\} \\
\mathit{fills} &:= \{(\mathit{Person}, \mathit{Student}), (\mathit{Person}, \mathit{Professor}), (\mathit{Person}, \mathit{StudentAssistant}), (\mathit{Person}, \mathit{ResearchAssistant}), \\
&\quad (\mathit{Person}, \mathit{TeamMember}), (\mathit{Person}, \mathit{Coach}), (\mathit{Person}, \mathit{Captain}), (\mathit{Seminar}, \mathit{StudiesCourse}), \\
&\quad (\mathit{Lecture}, \mathit{StudiesCourse}), (\mathit{SportsTeam}, \mathit{TournamentTeam}), (\mathit{SportsTeam}, \mathit{WinnerTeam})\} \\
\mathit{parts} &:= \{\mathit{University} \to \{\mathit{Student}, \mathit{Professor}, \mathit{StudentAssistant}, \mathit{ResearchAssistant}, \mathit{StudiesCourse}\}, \\
&\quad \mathit{SportsTeam} \to \{\mathit{TeamMember}, \mathit{CoachCaptain}\}, \\
&\quad \mathit{Tournament} \to \{\mathit{TournamentTeam}, \mathit{WinnerTeam}\}\} \\
\mathit{rel} &:= \{\mathit{takes} \to (\mathit{Student}, \mathit{StudiesCourse}), \mathit{teaches} \to (\mathit{Professor}, \mathit{StudiesCourse}), \\
&\quad \mathit{supervises} \to (\mathit{Professor}, \mathit{Student})\} \\
\mathit{card} &:= \{\mathit{takes} \to (0..\infty, 0..\infty), \mathit{teaches} \to (0..\infty, 1..1), \mathit{supervises} \to (0..\infty, 0..1)\}
\end{aligned}
$$

The example schema specifies a university domain $\mathcal{U}$. At first, each entity type of the domain model is classified for its metatype and placed it in the respective set. For instance, the domain model consists of three Natural Types; **Person**, **Seminar**, and **Lecture**. These are placed in the respective set for the Natural Types, NT. In contrast, types that contain Role Types are classified as Compartment Types, such as **University**, **SportsTeam**, and **Tournament**.

Next, the *fills* relation is populated and describes which player types can fulfill which Role Types. For example, **Person** and **Student** are related in this relation, stating a **Person** is able to play Role of the Role Type **Student**. Thus, a player type can only be extended by Role Types that are part of this relation and database system is in charge to ensure this. Afterwards, the *parts* function is populated in accordance to the specified domain model. In detail, the **University** Compartment Type contains the Role Types **Student**, **StudentAssistant**, **ResearchAssistant**, **Professor**, and **StudiesCourse**. Finally, the Relationship Types and their corresponding cardinality constraints are defined. Hence, the *rel* function defines a **take** Relationship Type between the Role Types **Student** and **StudiesCourse** whereas the *card* function applies the corresponding cardinality constraints that each **Student** Role does not have to take a **StudiesCourse** and a **StudiesCourse** Role does not need a **Student** to exist.

As it can be easily seen, $\mathcal{U}$ fulfills the axioms (4.1) to (4.5), as each Role Type is filled and part of the *fills* relation (4.1), each Compartment Type contains distinct Role Types (4.2 and 4.3), and finally each Relationship Type relates two distinct Role Types (4.4) in a common Compartment Type (4.5).

So far, the database schema consists of loosely coupled types, which are related by relations and functions, but no integrated data structure to bring them together. Hence, we augment the existing database schema by the notion of a Dynamic Data Type (DDT).

Dynamic Data Types

A Dynamic Data Type integrates the various metatypes into an integrated logical data structure. This facilitates the perception of objects that can be extended and shrunk during runtime. In contrast to the schema definition, at which the types are collected in separate sets and related by special relations and functions, the DDT brings all these definitions and constraints together in an integrated structure. Dynamic Data Types define how the database system structures and organizes its schema elements. The corresponding DDT specification is defined in Definition 3. This definition is an extension of the specification presented in [50] by introducing a second dimension Role Types can be located in.

Definition 3 (Dynamic Data Type)**.** *Let* $\mathcal{S} = (NT, RT, CT, RST, fills, parts, rel, card)$ *be a schema and* $t \in NT \cup CT$ *be either a Natural or a Compartment Type. A* Dynamic Data Type *is defined as* $ddt = (t, FT, PT)$ *with* $FT := \{rt \in RT \mid (t, rt) \in fills\}$ *and* $PT := parts(t)$.

As such, a DDT is considered as individual type having a Natural Type or Compartment Type in its core. Additionally, Role Types are structured in two dimensions within the DDT, in particular the filling and participating dimension. The former defines which Role Types are playable by the core and the latter specifies which Role Type can participate in the core. Hence, the participating dimension can only be populated for Compartment Type cores, because Natural Types cannot have any Role Types in their inner. However, both dimension are represented as set of Role Types within the DDT. Moreover, the DDT defines by which Role Types a core can be extended. Hence, it denotes the natural limits of an entity extension. Furthermore, it integrates all axioms of the basic schema elements in an integrated data structure and avoids object schizophrenia. This means, a Role Type will always be considered as part of a DDT and not as separate stand-alone data structure. In sum, a Dynamic Data Type represents the intended perception a separation of concerns within an entity type on the schema level, especially that a core can be extended by roles dynamically without changing the type at all, but by varying the played roles. Additionally, Role Types within a Compartment Type are considered in a separate dimension. A DDT, including its components and dimensions, is graphically illustrated in Figure 4.9. The notation of the various metatypes follows the CROM notation as shown and explained in Figure 2.3.

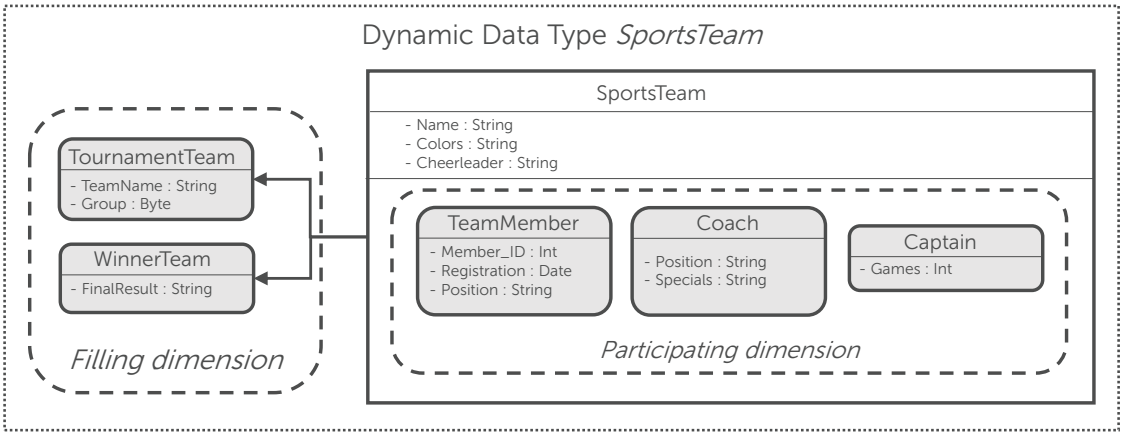


Figure 4.9: Dynamic Data Type *SportsTeam*

This DDT describes the SportsTeam and features the **SportsTeam** Compartment Type as core type. The filling dimension consists of the Role Types **TournamentTeam** and **WinnerTeam**. In which Compartment Type these Role Types are integrated in, does not matter for this DDT. In contrast, the participating dimension holds the set members **TeamMember**, **Coach**, and **Captain**. The player types

of these Role Types does not matter for this dimension. Hence, **SportsTeam** entities can grow their structure into two general directions, each having possibly multiple options in terms of Role Types.

Representing the university domain model, as outlined in Section 2.4.1, using the augmented database schema, results in the DDT specifications listed in Example 2.

Example 2 (Dynamic Data Type). *Let $\mathcal{U} = (NT, RT, CT, RST, fills, parts, rel, card)$ be the data model for the university domain (Example 1); then the above definition gives rise to the following DDTs within $\mathcal{U}$:*

$$\begin{aligned}
ddt_{Person} &= (Person, \{Student, StudentAssistant, ResearchAssistant, Professor,\\
&\qquad TeamMember, Coach, Captain\}, \emptyset)\\
ddt_{Seminar} &= (Seminar, \{StudiesCourse\}, \emptyset)\\
ddt_{Lecture} &= (Lecture, \{StudiesCourse\}, \emptyset)\\
ddt_{University} &= (University, \emptyset, \{Student, StudentAssistant, ResearchAssistant,\\
&\qquad Professor, StudiesCourse\})\\
ddt_{SportsTeam} &= (SportsTeam, \{TournamentTeam, WinnerTeam\},\\
&\qquad \{TeamMember, Captain, Coach\})\\
ddt_{Tournament} &= (Tournament, \emptyset, \{TournamentTeam, WinnerTeam\})
\end{aligned}$$

Basically, the formal representation of a DDT has three components. At first, the core that is either a Natural Type or Compartment Type. Secondly, the set of filled Role Types denoting the filling dimension. Finally, a set of contained Role Types specifying the participating dimension. Consequently, each Natural Type and Compartment Type forms an individual DDT. However, Role Types can be shared among several DDTs, for instance, the Role Type **StudiesCourse**, which is playable by the Natural Types **Lecture** and **Seminar**. In case the core is a Natural Type, the participating dimension will always be an empty set. The number of Dynamic Data Types is directly related to the amount of Natural Types and Compartment Types defined in the system. Thus, the number of DDTs can be determined by $|DDT| = |NT| + |CT|$.

Each type consists of a set of attributes that can be unfolded to represent the DDT's structure. For instance, $ddt_{SportsTeam}$ can be displayed as follows:

$$\begin{aligned}
ddt_{SportsTeam} = &\{[Name : String, Colors : String, Cheerleader : String],\\
&\Big(\{[Member_ID : Int, Registration : Date, Position : String]\},\\
&\{[Position : String, Specials : String]\},\\
&\{[Games : Int]\}\Big),\\
&\Big(\{[TeamName : String, Group : Byte]\},\\
&\{[FinalResult : String]\}\Big)\}
\end{aligned}$$

At first, the core defines the basic structure each entity of this Dynamic Data Type will have. Precisely, the specification comprises a *Name*, a *Color*, and the *Cheerleaders*. Next, the structure defined in the filled and participated Role Types is added. For instance, the **TeamMember** Role Type in the filling dimension defines the attribute *Member_ID*, *Registration*, and *Position*. All Role Type attributes are related by *can have* semantics, indicating that these attributes may be acquired during runtime or may be not. Additionally, the multiple Roles of the same Role Type can be acquired, indicated by the surrounding $\{\}$. Finally, the participating dimension of $ddt_{SportsTeam}$ is unfolded, for instance, the Role Type **TournamentTeam**. This Role Type adds a *TeamName* and a *Group* to this DDT. In sum, the Dynamic Data Type definition enables the specification of complex and dynamically evolving entities.

Overlapping Dynamic Data Types

By definition, Dynamic Data Types overlap by their Role Types. Role Types are connected to at least one core type that fills this particular Role Type and to exactly one Compartment Type this Role Type is featured in. Thus, a Role Type in the filling dimension of a certain Dynamic Data Type has to be part in the featuring dimension of another one. There might exist Role Types that do not result in a DDT interconnection. This occurs in case a Compartment Type fills a Role Type that is contained in itself. Consequently, the Role Type would be part of the filling and participating dimension of the same DDT. However, the usual resulting data structure is a net of interconnected DDTs as shown in Figure 4.10.

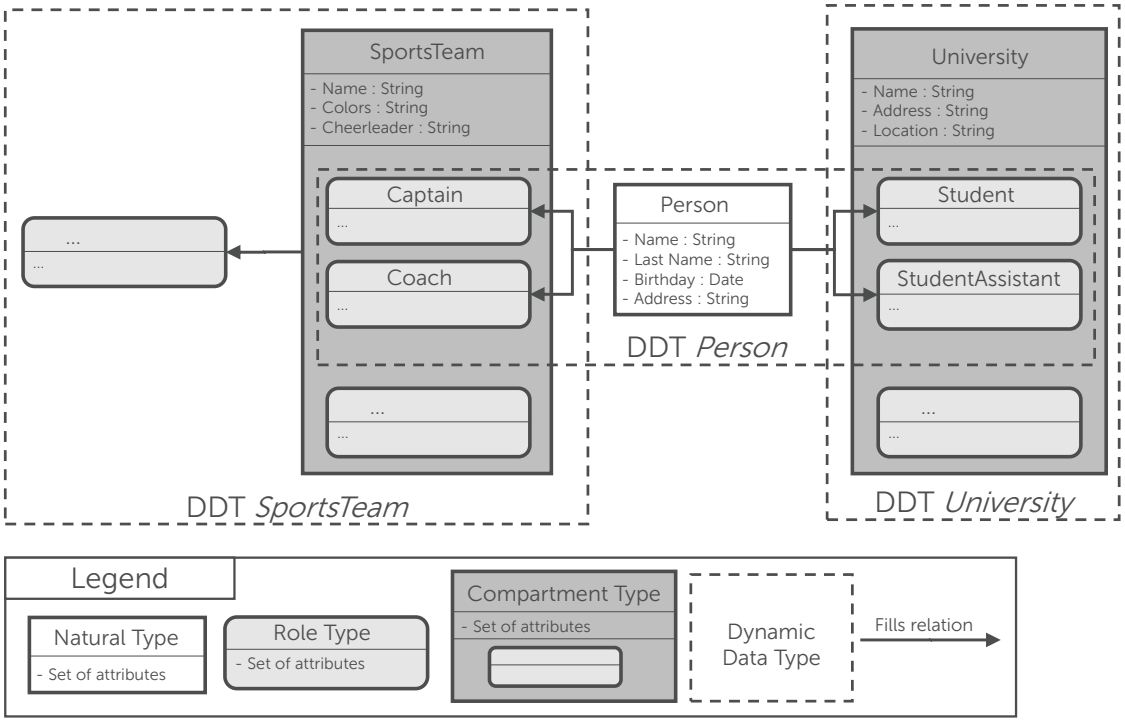


Figure 4.10: Net of Interconnected Dynamic Data Types

Natural Types share Role Types in the filling dimension with Compartment Types having the same Role Type in their participating dimension. For instance, **Person** fills the Role Type **Student** in the Compartment Type **University** and the Role Type **Captain** in the a **SportsTeam**. Thus, the DDT formed around **Person**, namely ddt_{Person} overlaps with the DDT $University$ in the Role Types $Student$ and $Professor$.

This can also be seen in the formal example represented in Example 2. The ddt_{Person} fills the Role Type $Student$, $Professor$, $StudentAssistant$, and $ResearchAssistant$. These Role Types are also part of $ddt_{University}$, but in the participating dimension. Consequently, both Dynamic Data Types overlap at these Role Types. This overlapping information will be used during query writing and processing.

There is another option for two distinct DDTs to overlap; in case they share the same Role Type in the filling dimension. As the *fills* relation is surjective, multiple player types for one Role Type may exist. Hence, this particular Role Type is part of a DDT's filling dimension several times. For example, $ddt_{Seminar}$ and $ddt_{Lecture}$ share the Role Type $StudiesCourse$. In sum, the database model's schema level consists of interconnected Dynamic Data Types that overlap by certain Role Types.

4.3.2 Instance Level

The instance level of a database represents the actual data, their structure, and interrelations. Hence, Natural Types, Role Types, Compartment Types and Relationship Types are instantiated to Naturals, Roles, Compartments, and links, respectively [51, 57]. With these instances context-dependent information and dynamically extending entities can be handled and the metatype distinction is preserved. The formal definition of an instance is given in Definition 4. Notably, this definition is a combination of the instance specifications presented in [51, 49]. In detail, the definitions of [51] are enriched by the relationship specific descriptions and constraints given in [49].

Definition 4 (Instance). *Let $\mathcal{S} = (NT, RT, CT, RST, fills, parts, rel, card)$ be a schema and N, R, and C be mutual disjoint sets of Naturals, Roles and Compartments, respectively. Then, an* Instance *of $\mathcal{S}$ is a tuple $\mathfrak{i} = (N, R, C, type, plays, links)$, where:*

- *$type : (N \to NT) \cup (R \to RT) \cup (C \to CT)$ is a labeling function from the instances to their corresponding type,*
- *$plays \subseteq (N \cup C) \times C \times R$ a relation, and*
- *$links : RST \times C \to 2^{R^\varepsilon \times R^\varepsilon}$ is a total function such that $R^\varepsilon \coloneqq R \cup \{\varepsilon\}$ with $\varepsilon \notin R \cup N \cup C$.*

Moreover, $E \coloneqq N \cup C$ denotes the set of all entities in $\mathfrak{i}$ and $E^c \coloneqq \{e \in E \mid \exists r \in R : (e, c, r) \in plays\}$ the set of entities that play a role in a compartment $c \in C$. Furthermore, $pred(rst, c, r) \coloneqq \{\overline{r} \mid (\overline{r}, r) \in links(rst, c) \wedge \overline{r} \neq \varepsilon\}$ and $succ(rst, c, r) \coloneqq \{\overline{r} \mid (r, \overline{r}) \in links(rst, c) \wedge \overline{r} \neq \varepsilon\}$ the predecessors respectively successors of a given role with respect to a relationship type.

To be a valid *instance of schema $\mathcal{S}$, $\mathfrak{i}$ must satisfy the following axioms:*

$$\forall (e, c, r) \in plays : (type(e), type(r)) \in fills \wedge type(r) \in parts(type(c)) \tag{4.6}$$

$$\forall (e, c, r), (e, c, r') \in plays : r \neq r' \Rightarrow type(r) \neq type(r') \tag{4.7}$$

$$\forall r \in R \, \exists! e \in E \, \exists! c \in C : (e, c, r) \in plays \tag{4.8}$$

$$\forall rst \in RST \, \forall c \in C : (\varepsilon, \varepsilon) \notin links(rst, c) \tag{4.9}$$

$$\begin{aligned} \forall rst \in RST \, \forall c \in C \, \forall r \in R \, \forall e \in E \, \exists \hat{r} \in R^\varepsilon : rel(rst) = (rt_1, rt_2) \wedge \\ \big(((e, c, r) \in plays \wedge type(r) = rt_1) \Leftrightarrow ((r, \hat{r}) \in links(rst, c))\big) \wedge \\ \big(((e, c, r) \in plays \wedge type(r) = rt_2) \Leftrightarrow ((\hat{r}, r) \in links(rst, c))\big) \end{aligned} \tag{4.10}$$

$$\forall rst \in RST \, \forall c \in C \, \forall (r_1, r_2) \in links(rst, c) \cap R \times R : (r_1, \varepsilon), (\varepsilon, r_2) \notin links(rst, c) \tag{4.11}$$

$$\begin{aligned} \forall rst \in RST \, \forall c \in C \, \forall (r_1, r_2) \in links(rst, c) : card(rst) = (i..j, k..l) \wedge \\ \big(r_2 \neq \varepsilon \Rightarrow i \leq |pred(rst, c, r_2)| \leq j\big) \wedge \\ \big(r_1 \neq \varepsilon \Rightarrow k \leq |succ(rst, c, r_1)| \leq l\big) \end{aligned} \tag{4.12}$$

In general, an instance of a schema is a collection of entities and Roles together with their individual interrelations. In particular, the *type* function is polymorphic and maps each instance to its respective type. Moreover, the *plays*-relation is the instance level equivalent of the *fills* relation in combination with the *parts* function, as it identifies those entity cores (either Natural or Compartment) playing a role in a certain Compartment. Similarly, the *links* function represents the context-dependent Relationship Type, by collecting all Relationships (i.e., all linked Roles) of this type for each Compartment.

On top of that, *valid* instances are required to be consistent to a schema, i.e., they satisfy the above seven axioms. The first axiom ensures that the *plays* relation complies to *fills* and *parts* on the type level (4.6). Precisely, this guarantees that an entity plays only Roles of Role Types, which are connected with the entity's core type on the schema level by the *fills* relation. Additionally, the Role must be played in a Compartment it is intended to be played in, which is represented by the *parts* function.

The second axiom restricts an entity to play a Role of a certain Role Type only once per Compartment (4.7). In general, playing multiple Roles of a certain Role Type simultaneously is possible with respect to the instance definition, but only in distinct Compartments. For instance, the **Person** *John* can play the **Student** Role within the **TUD** Compartment only once, giving him the ability to enroll in **StudiesCourses** and gain the structure of a **Student_ID**, for instance. Within this *TUD* Compartment he cannot obtain another **Student** Role, but within any other **University** Compartment he is currently not playing a **Student** Role.

The third axiom enforces Roles to have exactly one player and is contained within one Compartment only (4.8). On the schema level, Role Type can be connected to several player types, which cannot be transferred to the instance level. Rather, a specific Role is exclusive to a certain player and within a certain Compartment.

The next three axioms, in turn, enforce that for each Role participating in a Compartment whose Role Type is linked by a Relationship Type, there is a tuple in the corresponding *links* function (4.10). To represent zero-to-one and zero-to-many Relationships, a Role can also be linked to the empty counter role ε. 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, ε is introduced 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. However, it is required that a Role can only be linked once to ε (4.11). Additionally, an empty counter role cannot be related to another empty counter role by the *links* function (4.9).

Last but not least, axiom (4.12) validates the cardinality constraints imposed by the *card* function. In detail, it checks for each compartment and relationship type whether the number of *predecessors* and *successors* are within the lower and upper bound of the corresponding cardinality constraint. The empty counter Role ε is not counted. In conclusion, the intuitive semantics of the cardinality constraints can be established locally, i.e., within each Compartment, because (4.12) enforces their lower and upper bounds and (4.8) ensures no entity plays two distinct Roles of the same Role Type in one Compartment. In sum, these instance level definitions and consistency constraints can represent both dynamic complex objects and context-dependent relationships. To demonstrate these specifications and constraints, the example illustrated in Figure 2.4 is represented as formal instance in Example 3.

Example 3 (Instance)**.** *Let* $\mathcal{U} = (NT, RT, CT, RST, \mathit{fills}, \mathit{parts}, \mathit{rel}, \mathit{card})$ *be the schema defined in Example 1; then* $\mathfrak{u} = (N, R, C, \mathit{type}, \mathit{plays}, \mathit{links})$ *is an instance (Figure 2.4) of that schema (Figure 2.3),*

where the components are defined as follows:

$$
\begin{aligned}
N &\coloneqq \{John, Max, Tim, Gert, Kai, sem_1, l_1\} \\
R &\coloneqq \{s_1, s_2, s_3, sa_1, ra_1, p_1, sc_1, sc_2, tm_1, tm_2, cap_1, c_1, tt_1, w_1\} \\
C &\coloneqq \{TUD, bears, cc16\} \\
type &\coloneqq \{(John \to Person), (Max \to Person), (Tim \to Person), (Gert \to Person), \\
&\quad (Kai \to Person), (s_1 \to Student), (s_2 \to Student), (s_3 \to Student), \\
&\quad (sa_1 \to StudentAssistant), (ra_1 \to ResearchAssistant), (p_1 \to Professor), \\
&\quad (sc_1 \to StudiesCourse), (sc_2 \to StudiesCourse), (tm_1 \to TeamMember), \\
&\quad (tm_2 \to TeamMember), (cap_1 \to Captain), (tt_1 \to TournamentTeam), \\
&\quad (w_1 \to WinnerTeam), (TUD \to University), (bears \to SportsTeam), \\
&\quad (cc16 \to Tournament)\} \\
plays &\coloneqq \{(John, TUD, s_1), (John, TUD, sa_1), (John, bears, tm_1), \\
&\quad (John, bears, cap_1), (Max, TUD, s_2), (Max, bears, tm_2), (Max, bears, c_1), \\
&\quad (Tim, TUD, s_3), (Kai, TUD, ra_1), (Gert, TUD, p_1), (sem_1, TUD, sc_1), \\
&\quad (l_1, TUD, sc_2), (bears, cc16, st_1), (bears, cc16, w_1)\}
\end{aligned}
$$

$$
\begin{aligned}
links(takes, TUD) &\coloneqq \{(s_1, sc_1), (s_2, sc_1), (s_3, \varepsilon), (\varepsilon, sc_2)\} \\
links(teaches, TUD) &\coloneqq \{(p_1, sc_1), (p_1, sc_2)\} \\
links(supervises, TUD) &\coloneqq \{(p_1, s_1), (\varepsilon, s_2), (\varepsilon, s_3)\}
\end{aligned}
$$

The instance $\mathfrak{u}$ of the university schema $\mathcal{U}$ is constructed by assigning the Naturals, Roles, and Compartments to their respective sets. Moreover, the *type* function is specified to map each instance to its corresponding type. For instance, *John* is assigned to the Natural Type **Person** (John $\to$ Person). Next, the *plays* relation is populated for each role in Figure 2.4. Each triple denotes that the entity e is contained within the Compartment c by playing the Role r. Additionally, these triples are unique with respect to the entity, Compartment, and Role, because each Role has exactly one player in exactly one Compartment. Moreover, for a given entity, a Compartment and the Role type, each Role is identifiable. Finally, the *links* function is defined to capture the relationships between **Professors** and **Students**, **Students** and **StudiesCourses**, and **Professors** and **StudiesCourses**. All of them are located within the *TUD* Compartment. As it can be seen, Roles of Role Types connected by a Relationship Type having lower bound of zero, Student to StudiesCourse for instance, are included in this function as well, but with an empty counter role (s_3, ε). In sum, $\mathfrak{u}$ is the formal representation of the graphical illustration shown in Figure 2.4.

Furthermore, the instance $\mathfrak{u}$ is consistent with respect to the schema $\mathcal{U}$ by fulfilling all instance's axioms. At first, each triple in *plays* complies the to schema level definition of *fills* and *parts* (4.6). Secondly, no Natural or Compartment plays Roles of the same Role Type in the same Compartment multiple times simultaneously (4.7, 4.8). In fact, *John* plays two Roles in the Compartment *TUD*, but of different Role Types, which is totally allowed. Thirdly, each Role is present in the corresponding links function, at least with an empty counter role (4.10) but at most once to this empty counter role (4.11). Additionally, each links function does not contain the mapping of an empty counter role to another empty counter role (4.9). Finally, the number of predecessors and successors is within the range of the cardinality constraints for each Relationship Type (4.12).

So far, the schema instance representation contains loosely interrelated instances of Naturals, Roles, and Compartments, but no integrated data structure that brings the role-semantics together. Hence, the instance definitions are augmented by the notion of a Dynamic Tuple.

Dynamic Tuple

A **Dynamic Tuple** (DT) is the instance level representation of a Dynamic Data Type. It brings an entity core and the played as well as featured Roles together in an integrated data structure. A DT notion builds the basis of handling dynamically changing instances in a DBS. The DT's formal definition is specified in Definition 5 and is based on the specification in [51].

Definition 5 (Dynamic Tuple)**.** *Let $\mathcal{S} = (NT, RT, CT, RST, fills, parts, rel, card)$ be a schema, $\mathfrak{i} = (N, R, C, type, plays, links)$ a valid instance of $\mathcal{S}$, and $e \in E$ is an entity, either a Natural or Compartment, of type t, i.e., $type(e) = t$.*
A Dynamic Tuple *$dt = (e, F, P)$ is then defined with respect to the played roles and featured roles given as:*

$$F := \{\{r \mid (r, rt) \in \overline{F_e}\} \mid rt \in RT\} \qquad \text{with } \overline{F_e} := \{(r, type(r)) \mid (e, \_, r) \in plays\}$$
$$P := \{\{r \mid (r, rt) \in \overline{P_e}\} \mid rt \in RT\} \qquad \text{with } \overline{P_e} := \{(r, type(r)) \mid (\_, e, r) \in plays\}$$

While DDTs are defined to capture all possible configurations of a dynamic entity type, a Dynamic Tuple is defined to exactly reflect a dynamic entity including its Roles. Hence, a DT is defined to capture the current rigid instance, all the Roles it currently plays, and all the Roles it contains. Like the DDT on the schema level, a Dynamic Tuple consists of Roles in two dimensions, the *playing* and the *featuring* dimension. However, as an entity can play and contain multiple Roles of the same Role Type multiple times, they are grouped by their corresponding Role Type into the sets F of actually filled Role Types and P of participating Role Types, respectively. Consequently, only those Role sets will show up in these sets F and P, if there exists at least one Role of the corresponding Role Type, which is being played or featured by this Dynamic Tuple. Hence, empty sets are omitted in F and P. Note, if the set of currently filled or participating Role Types is empty, i.e., no Role is actually played or featured in a given entity. In sum, this definition captures both dimensions of dynamic complex entities. Henceforth, components of individual instances $\mathfrak{i}$ and Dynamic Tuples dt are referred to with subscripts, whenever their origins would be ambiguous. An illustration of a Dynamic Tuple is depicted in Figure 4.11.

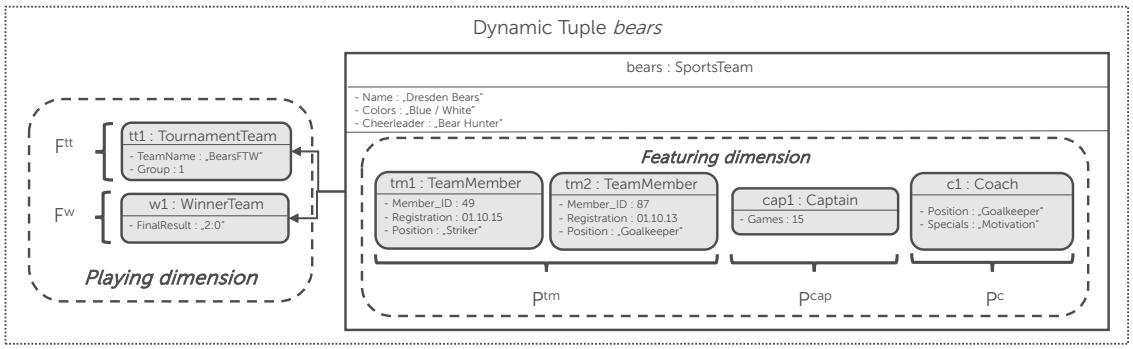


Figure 4.11: Dynamic Tuple *bears*

In particular, the *bears* Dynamic Tuple consists of the *bears* Compartment in its core. Otherwise, in case of a Natural as core, the featuring dimension will be empty. However, this Dynamic Tuple consists of two dimensions, the playing and featuring dimension, to group the Roles that are played and featured, respectively. The playing dimension holds all Roles that are currently played by the core, whereas the featuring dimension stores Roles that are currently featured within this Compartment. In detail, the playing dimension is populated with two Roles, tt_1 and w_1, both holding additional structure. Additionally, the Roles are grouped into their respective sets F^{tt} and F^{w}. The featuring

dimension holds four Roles, two **TeamMember** Roles, one **Captain** Role and one **Coach** Role. Again, the Roles are grouped by their respective Role Type, for instance, the **TeamMember** Roles tm_1 and tm_2 are grouped into the set P^{tm}.

Applying Definition 5 on the instance $\mathfrak{u}$ (see Figure 2.4) results in the Dynamic Tuples presented in Example 4.

Example 4 (Dynamic Tuple). *Let $\mathfrak{u} = (N, R, C, type, plays, links)$ be the instance of the university schema $\mathcal{U}$ (Example 3). This model contains the following Dynamic Tuples:*

$$
\begin{aligned}
dt_{John} &:= (John, \{\{s_1\}, \{sa_1\}, \{tm_1\}, \{cap_1\}\}, \emptyset) \\
dt_{Max} &:= (Max, \{\{s_2\}, \{tm_2\}, \{c_1\}\}, \emptyset) \\
dt_{Tim} &:= (Tim, \{\{s_3\}\}, \emptyset) \\
dt_{Gert} &:= (Gert, \{\{p_1\}\}, \emptyset) \\
dt_{Kai} &:= (Kai, \{\{ra_1\}\}, \emptyset) \\
dt_{sem1} &:= (sem1, \{\{sc_1\}\}, \emptyset) \\
dt_{l1} &:= (l1, \{\{sc_2\}\}, \emptyset) \\
dt_{TUD} &:= (TUD, \emptyset, \{\{s_1, s_2, s_3\}, \{sc_1, sc_2\}, \{sa_1\}, \{ra_1\}, \{p_1\}\}) \\
dt_{bears} &:= (bears, \{\{tt_1\}, \{w_1\}\}, \{\{tm_1, tm_2\}, \{cap_1\}, \{c_1\}\}) \\
dt_{cc16} &:= (cc16, \emptyset, \{\{tt_1\}, \{w_1\}\})
\end{aligned}
$$

This instance comprises ten Dynamic Tuples, one per Natural and one per Compartment. Hence, the number of Dynamic Tuples a system manages is directly determined by the number of Naturals and Compartments in the system. Thus, the amount of Dynamic Tuples in a system is defined as: $|DT| = |N| + |C|$. As it can be seen, all Dynamic Tuples having a Natural as core do not feature any Role (dt_{John}, dt_{Max}, dt_{Tim}, dt_{Gert}, dt_{Kai}, dt_{sem1}, dt_{l1}), thus, the featuring dimension is empty. In contrast, each Compartment features Roles (dt_{TUD}, dt_{bears}, dt_{cc16}), but only dt_{bears} plays Roles as well.

An entity consists of a core and Role in two dimensions enabling dynamic adaptations on the entities structure with the limits of its corresponding DDT. The core as well as the Roles can be unfolded to represent the whole entity including its attributes. For instance, dt_{bears} can be illustrated as:

$$
\begin{aligned}
dt_{bears} = \Big(& (Name : \text{"bears"}, Colors : \text{"Blue/White"}, Cheerleader : \text{"Bear Hunter"}), \\
& \{\{(TeamName : \text{"BearsFTW"}, Group : 1)\}, \\
& \{(FinalResult : \text{"2:0"})\}\}, \\
& \{\{(Member\_ID : 49, Registration : 01.10.15, Position : \text{"Striker"}), \\
& (Member\_ID : 87, Registration : 01.10.13, Position : \text{"Goalkeeper"})\}, \\
& \{(Games : 15)\}, \\
& \{(Position : \text{"Goalkeeper"}, Specials : \text{Motivation})\}\}\Big)
\end{aligned}
$$

At first, the core is unfolded giving the entity dt_{bears} the attributes $Name$, $Colors$, and $Cheerleader$. Next, all Roles in the two dimensions are unfolded by their respective sets. The bears participated to the *cc16* **Tournament** as **TournamentTeam** and won it. Hence, these Roles are unfolded to their

certain structure $TeamName$ and $Group$ as well as $FinalResult$. The interesting case is, containing multiple Roles of the same Role Type simultaneously, like the two **TeamMember** Roles tm_1 and tm_2. They are unfolded within the set of all **TeamMembers**, thus, the $Member\_ID$, for instance, is populated multiple times. In sum, the Dynamic Tuple definition enables the representation of complex and dynamically evolving entities.

Overlapping Dynamic Tuples

As the Dynamic Data Types on the schema level overlap, this characteristic is inherited to Dynamic Tuples. Each Role is required to have exactly one player and is featured within exactly one Compartment. Hence, different Dynamic Tuples overlap by certain Roles. In detail, each Role has to be contained in a Dynamic Tuple's playing dimension and in another Dynamic Tuple's featuring dimension. In contrast to DDTs, where a Role Type could be filled and participated within the same Compartment Type, Roles on the instance level have to be part of two distinct Dynamic Tuples. Consequently, they cannot be played and featured by the same DT. A graphical example of overlapping Dynamic Tuples is presented in Figure 4.12. Please note, for a clear arrangement we left out unnecessary Roles and all Role attributes in this figure. Additionally, we shaded the Compartments in gray to provide a better optical distinction between the several metatypes and the corresponding Dynamic Tuples.

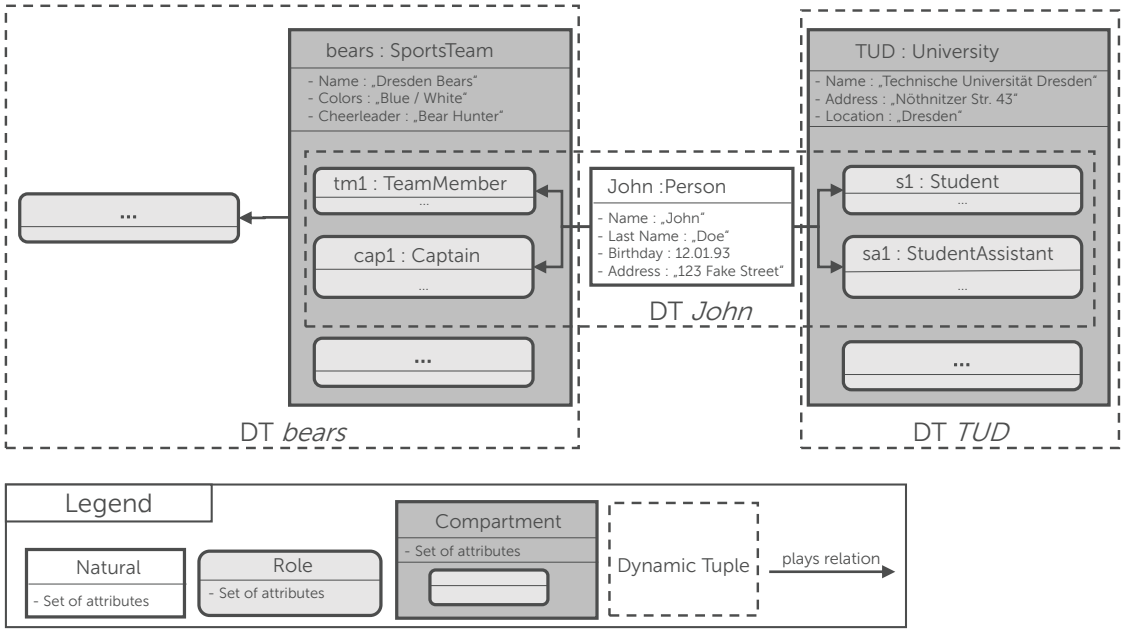


Figure 4.12: Net of Interconnected Dynamic Tuples

In detail, Naturals or Compartments share Roles in their playing dimension with Compartments in which the Role is present in the featuring dimension. For instance, dt_{John} plays the Roles s_1 within the Compartment TUD and tm_1 within $bears$. Because both, TUD and $bears$, form a Dynamic Tuple themselves, dt_{John} shares the Role s_1 with dt_{TUD} and the Role tm_1 with dt_{bears}.

The overlap can also be seen in the formal representation of this example (see Example 4). The Role s_1 is once referenced by $John$ in the playing dimension and once by TUD in the featuring dimension. Hence, both Dynamic Tuples share structure and values and overlap at this Role. The overlap is not limited to a single Role, in fact, two Dynamic Tuples may share several Roles.

Relations Within and Between Dynamic Tuples

To conclude the definition of Dynamic Tuples, both, endogenous and exogenous relations are specified with respect to the definitions in [51]. The former allows to navigate into the played and featured Roles of a particular Dynamic Tuple, whereas the latter allows to navigate from one Dynamic Tuple to another by means of a particular Role.

Definition 6 (Endogenous Relations). *Let* $\mathfrak{i} = (N, R, C, type, plays, links)$ *be a valid instance of an arbitrary schema* $\mathcal{S}$*,* $e \in E$ *an entity in* $\mathfrak{i}$*, and* $dt = (e, F, P)$ *the corresponding Dynamic Tuple. Then* dt plays *a role* $r \in R$*, iff* $r \in F^i$ *where* $F = (F^1, \ldots, F^i, \ldots, F^m)$ *and* $i \in \{1, \ldots, m\}$ *or simply* $(e, \_, r) \in plays$*. Similarly,* dt features *a role* $r \in R$*, iff* $r \in P^j$ *where* $P = (P^1, \ldots, P^j, \ldots, P^n)$ *and* $j \in \{1, \ldots, n\}$*, in essence* $(\_, e, r) \in plays$*.*

Basically, this lifts the notion of playing and featuring roles to the level of Dynamic Tuples. Consider, for instance, the Dynamic Tuple dt_{bears} that currently *plays* tt_1 as well as w_1 and *features* the Roles tm_1, tm_2, cap_1 and c_1. These two relations enable to access a Dynamic Tuple's internal information that are organized in dimensions and sets of Roles.

While these relations allow to navigate within a Dynamic Tuple, the *Exogenous Relations* permit navigation between Dynamic Tuples.

Definition 7 (Exogenous Relations). *Let* $\mathfrak{i} = (N, R, C, type, plays, links)$ *be a valid instance of an arbitrary schema* $\mathcal{S}$*. Furthermore, let* $e_1, e_2 \in E$ *be two entities in* $\mathfrak{i}$*, and* $a = (e_1, F_a, P_a)$*,* $b = (e_2, F_b, P_b)$ *their respective Dynamic Tuples.*

Then a *is* featured in b *with* $r \in R$*, iff* a *plays* r *and* b *features* r*, such that* $r \in F_a^i \cap P_b^j$*, where* $F_a = (F_a^1, \ldots, F_a^i, \ldots, F_a^m)$ *and* $P_b = (P_b^1, \ldots, P_b^j, \ldots, P_b^m)$*. Similarly, its inverse is denoted as* b *contains* r played by a*.*

In addition, a *is* related to b *using the Relationship Type* $rst \in RST$*, iff there is a compartment* $c \in C$ *and a link* $(r_1, r_2) \in links(rst, c)$ *such that* a plays r_1 *and* b plays r_2*.*

In general, *featured in*, *played by*, and *related to* represents the various interrelations between entities on the instance level lifted to Dynamic Tuples. The former two relations directly utilize the overlapping information of Dynamic Tuples to navigate between them. In particular, *featured in* and *played by* search for Roles, which are included in a Dynamic Tuple's playing dimension and in another's featuring dimension, and vice versa. For instance, the Dynamic Tuple dt_{John} is *featured in* the University dt_{TUD} (with the role s_1). Moreover, dt_{John} is *related to* dt_{Gert} using the $supervises$ Relationship Type; within the *TUD* Compartment and Dynamic Tuple dt_{TUD}. In sum, exogenous relations can be used to navigate from one Dynamic Tuple to another one while endogenous relations can be used to navigate to the Roles within a Dynamic Tuple.

4.3.3 Configuration

The central idea behind the abstraction of entities to types, is to determine an entity's structure just by its belonging to a certain type and the type description. This means, all entities of the same type share the same structure. In terms of role-based type descriptions, especially the given definition for a DDT, this holds partially. The DDT is designed to cover all possible schemata an entity of this type can acquire. This opens a space of possible schemata that are assembled under this specification. On

the one hand, a Dynamic Tuple is the instance of a Dynamic Data Type and this belonging cannot be changed at all without losing its identity. On the other hand, the DDT specification opens a space of valid schemata such that entities of the same type can have different schemata. Consequently, entities of the same type may share some parts of their structure and some not. This results in uncertainty in regard of an entity's concrete structure.

To reintroduce the conventional abstraction mechanism in RSQL's database model and provide certainty about the entity's structure, Configurations are introduced. They act as mediator between the space of possible valid schemata described by a single DDT specification and the Dynamic Tuples on the instance level. The formal definition of a Configuration is given in Definition 8 and is based on specification of a Configuration in [51].

Definition 8 (Configuration)**.** *Let $\mathcal{S}$ be a schema and $ddt = (t, FT, PT)$ an arbitrary Dynamic Data Type in $\mathcal{S}$, then the set of all possible Configurations of this ddt is defined as*

$$\mathcal{C}_{ddt} \coloneqq \{(t, \hat{FT}, \hat{PT}) \mid \hat{FT} \subseteq FT \wedge \hat{PT} \subseteq PT\}.$$

In particular, a given Dynamic Tuple $dt = (e, F, P)$ with $type(e) = t$ in a valid instance $\mathfrak{i}$ of $\mathcal{S}$ is in exactly one Configuration

$$c_{dt} = (t, \{rt \mid (\_, rt) \in \overline{F_e}\}, \{rt \mid (\_, rt) \in \overline{P_e}\}).$$

Notably, it always holds that $c_{dt} \in \mathcal{C}_{ddt}$ for all instances dt of a given ddt.

In detail, a Configuration c^i_{ddt} describes one valid schema in the space of possible schemata defined by a DDT specification. Additionally, a Configuration is the missing part between a DDT and its DTs, thus, a Configuration can be seen from the DDT perspective on the one hand and from a Dynamic Tuple's perspective on the other hand. At first, the DDT perspective specifies a set of Configurations $\mathcal{C}_{ddt}$. Consequently, the Configuration definition is located on the type level, hence, it specifies a player type and its Role Types in the corresponding dimensions.

From the Dynamic Tuple perspective, a DT belongs to exactly one Configuration, which can be changed during runtime. This possibility to change the Configuration enables the dynamical adaption of an entity's structure without changing its overall type. However, a Dynamic Tuples Configuration is specified by the Role Types of currently played and featured Roles. In particular, $\overline{F_e}$ includes all Role to Role Type pairs of the playing dimension and is defined by $\overline{F_e} \coloneqq \{(r, type(r)) \mid (e, \_, r) \in plays\}$. $\overline{P_e}$ is defined by $\overline{P_e} \coloneqq \{(r, type(r)) \mid (\_, e, r) \in plays\}$ to collect all pairs of featured Roles and Role Types. Notably, the definitions of $\overline{F_e}$ and $\overline{P_e}$ are also presented in the Dynamic Tuple's specification in Definition 5. However, for a Dynamic Tuple's Configuration only the Role Types of currently played and featured Roles is of importance. Empty Role Types will not show up in these sets.

Moreover, playing or featuring Roles of the same Role Type multiple times does not affect the Configuration of a Dynamic Tuple at all, which is totally intended. The Configuration definition on the type level basis ensures a finite set of possible Configurations. This finite set is important, to check the validity of structure adaptations during runtime within the given space of possible schemata. In contrast, a definition on instance level basis would create an infinite set of Configurations, because the structure would be defined on the basis of Roles instead of Role Types. Generally, the number of playable Roles of the same Role Type is undefined, which means there is no upper limit. Hence, an infinite set of playable Roles would create an infinite set of Configurations.

To illustrate the Configurations and their semantics, they are explained by the $ddt_{SportsTeam}$ in the following Example 5.

Example 5 (Configuration)**.** *Let $ddt_{SportsTeam}$ be a valid Dynamic Data Type of the university schema $\mathcal{U}$. This DDT opens the space $\mathcal{C}_{ddt_{SportsTeam}}$ including the following Configurations:*

$$
\begin{aligned}
c^{min}_{SportsTeam} &:= (SportsTeam, \emptyset, \emptyset) \\
c^{1}_{SportsTeam} &:= (SportsTeam, \{TournamentTeam\}, \emptyset) \\
c^{2}_{SportsTeam} &:= (SportsTeam, \{TournamentTeam, WinnerTeam\}, \emptyset) \\
c^{3}_{SportsTeam} &:= (SportsTeam, \emptyset, \{TeamMember\}) \\
c^{4}_{SportsTeam} &:= (SportsTeam, \{TournamentTeam\}, \{TeamMember\}) \\
c^{5}_{SportsTeam} &:= (SportsTeam, \{TournamentTeam, WinnerTeam\}, \{TeamMember\}) \\
&\dots \\
c^{max}_{SportsTeam} &:= (SportsTeam, \{TournamentTeam, WinnerTeam\}, \\
&\qquad \{TeamMember, Captain, Coach\})
\end{aligned}
$$

In general, the Configuration space is limited by a minimal and maximal Configuration. On the one hand side, the minimal Configuration does not hold any Role Types in both dimensions, hence, the entity does not play any Roles. Formally, $\hat{FT}$ and $\hat{PT}$ are empty sets in this case. On the other hand, the maximal Configuration holds all possible Role Types in both dimension. Thus, at least one Role of each possible Role Type is played and featured by this entity. From a formal perspective, $\hat{FT} = FT$ and $\hat{PT} = PT$ in the maximal case. In the case, the DDT has a Natural Type as core, the maximal Configuration is limited by all playable Role Types only, because the participating dimension is always empty. The minimal and maximal Configuration of the example shown in Example 5 are denoted by $c^{min}_{SportsTeam}$ and $c^{max}_{SportsTeam}$, respectively.

Between those extrema, there are a lot of other Configurations, in particular each possible combination of Role Types in the filling and the participating dimension. For instance, $c^{1}_{SportsTeam}$ describes instances that only play one or more Roles of the Role Type $TournamentTeam$, whereas $c^{4}_{SportsTeam}$ collects all Dynamic Tuples that play a $TournamentTeam$ Role and feature a $TeamMember$ Role. Dynamic Tuples are free to change their Configuration within the DDT given Configuration space. However, each Dynamic Tuple having a Configuration within this space, is of the same Dynamic Data Type. For instance, the Configuration of the Dynamic Tuple *bears* is $c_{bears} = (SportsTeam, \{TournamentTeam, WinnerTeam\}, \{TeamMember, Captain, Coach\})$ because each Role Type has at least one Role within this DT. As it can be seen, this Configuration equals the maximal Configuration, hence, $c_{bears} \equiv c^{max}_{SportsTeam}$.

Moreover, a Dynamic Tuple's structure can be determined by its belonging to a certain Configuration within the Configuration space of a certain DDT. This reintroduces the traditional abstraction mechanism between entities and their corresponding types while preserving the role-semantics in RSQL's database model. By changing their Configuration, DTs are able to adapt their structure without changing their overall type at all. In sum, by allowing DT to change their Configuration, we allow entities to adapt their structure without changing their type and to combine different DT having divers Configurations being of the same overall type.

4.4 RSQL OPERATORS

A database system usually stores data produced by several applications as well as it provides a global perspective on the data and their interrelations in a software system. Each application defines its

Operator	Functionality
Σ_{cex}	Schema-based filter
Π_{α}	Filters for Roles of queried Role Types
κ_{α}	Filters for overlapping Roles
Ω_{rst}	Filters for Roles participating in a Relationship
τ	Core union for Role Types that can be played by multiple core types
$\setminus_{R-}$	Difference of Dynamic Tuples
$\setminus_{R}$	Difference of Dynamic Tuples on the Role dimensions level
$\cap_{\circ}^{RT_a,RT_b}$	Intersection of Dynamic Tuples and various options to unite the Role dimensions
$\cup_{\circ}^{RT_a,RT_b}$	Union of Dynamic Tuples and various options to unite the Role dimensions
$\sigma_{preicate}^{t,RT_{overlap}}$	Attribute-based filter

Table 4.3: Overview of the Operators and Their Functionality

individual perspective on the data and do not need or is prohibited to access all data. In contrast, the database system provides the global perspective on each entity it stores. To give an application the information it asks for, the local perspective has to be extracted from the global one. The local perspective is expressed by a query statement and the extraction process by database operators. In this sense, the database operators build the bridge between a query language and the database model. To provide this well-established abstraction mechanism for Dynamic Data Types and Dynamic Tuples, novel operators on RSQL's database model are required.

In general, database operators are based on the database model and describe which operations can be performed on the data structures and these structures can be manipulated. RSQL's main data structures are Dynamic Data Types and Dynamic Tuples, thus, the operators are defined to manipulate those. All operators are formally defined using set theory, because the database system handles sets of Dynamic Tuples, which consist of a core and sets of Role sets. Notably, a mathematical definition of database operators paves the way for logical and physical query optimization, which is a large research area in database systems' research. However, the RSQL database model consists of ten formal operators. This list is not complete and additional operators are possible, but they suffice to represent a RSQL query statement. An overview of the formal operators and their functionalities, respectively targeted manipulations, is presented in Table 4.3.

The operators are categorized into three classes: (i) operators on an entity's structure level, (ii) operators to enable usual set operations on the level of Dynamic Tuples, and (iii) operators on the attribute level. The first class filters Dynamic Tuples based on their structure, for instance, a Role of a certain Role Type has to be played or featured by a Dynamic Tuple or Dynamic Tuples have to overlap by a particular Role. The second category consists of operators that enable set operations like a difference of two Dynamic Tuple sets or an intersection of such. Finally, the third class filters Dynamic Tuples as whole data structure or Roles as parts of this structure, based on their attribute values.

However, to achieve a mathematical closure on the data model, constraints concerned with the embedding of Roles in Compartments are relaxed. This enables the processing of Dynamic Tuples without considering their overlap with other Dynamic Tuples. As a result, RSQL has a relaxed operational database model, which is used during query processing. Notably, the operational database model is a superset of the base database model.

4.4.1 Operational Data Model

RSQL's base data model, as defined in Section 4.3, features some constraints, for instance, a Role needs exactly one player and is embedded in exactly one Compartment. These constraints are valuable when the actual role-based data is stored, especially in terms of consistency. The resulting data structure consists of overlapping DDT on the schema level and overlapping DT on the instance layer. However, this overlapping of DTs (RSQL's notion of an evolving entity) complicates manipulating and processing the base data, because DTs cannot be considered in isolation when Roles are involved. For example, an application queries for a Person entity and its Student Roles, but not the University these Roles are embedded in, because the University is of no interest in the application's scenario. Such a query cannot be performed by using the base data model, because it would violate the overlapping constraints of DDT and DT by providing a DT consisting of Roles, but no Compartments these Roles are embedded in. This would also require to specify all Role dependencies, in terms of players and Compartments, within a query, even if this information is unnecessary for the application. Moreover, operators having only one input in terms of a set of Dynamic Tuples, for instance, Persons with all played Roles, are not possible in this data model, because these Roles would not be embedded into a Compartment.

To overcome this limitation, we define an operational data model that relaxes the overlapping and relationship related constraints. Especially, the axioms in 4.6 and 4.8 to 4.12 are suspended with respect to a Dynamic Tuple and linkages between Roles. The operational data model is focused on the notions of Dynamic Tuples only. All operators consume Dynamic Tuples and produces sets of those, without paying attention to some base data model constraints. Thus, during query processing Roles may appear that are not embedded in a Compartment, because the query does not ask for this information.

However, the operational data model is a superset of the base data model, because it suspends constraints only and do not add any. The relation between both data model is presented in Figure 4.13.

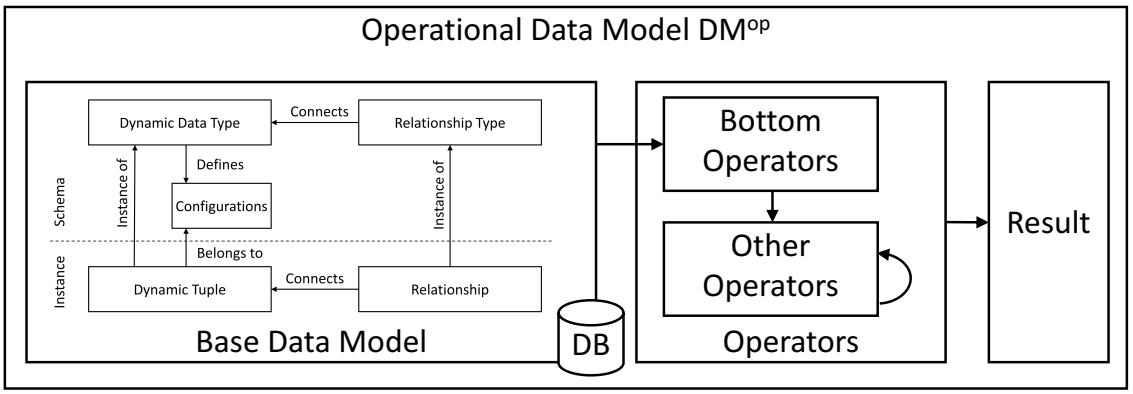


Figure 4.13: Operational Data Model as Superset of RSQL's Base Data Model

The database uses the base data model as described in Section 4.3 to store the actual data. Hence, for all base data stored the constraints, like each Role needs a player and a Compartment, are ensured. This builds the input for the most bottom operators. In detail, Dynamic Tuples, for which the constraints are ensured, in the base data are consumed by these operators, but without the overlapping information. For example, Dynamic Tuples of Persons in combination with all played Roles are the input for a certain operator. The overlapping information is uninteresting for a certain operator, because it is concerned with Dynamic Tuples rather than the whole instance model. The output Dynamic Tuples of an operator may be the input for another operator or directly lead to the result. All other operators do not consume base data, but an output of bottom or other upstream operators. In

the end, the topmost operators' output define the result. However, a query's output and the result is based on the operational data model. Thus, there may be Roles with no Compartment, because the Compartment has not been queried. Or the other way around, a Role is embedded in a Compartment but no player is provided.

The same holds for Relationships. In the base data all cardinality constraints are guaranteed, but for queries this information may be uninteresting. However, the base data information regarding overlapping and linkage between Dynamic tuples is leveraged by the operators.

In sum, the operational data model enables Dynamic Tuple focused query processing with the option to ignore overlapping and linkage information. Finally, all operators are based on the operational data model and produce a result that is an instance of this constraint relaxed data model, too. Thus, the operators build a mathematical closure on the operational data model.

4.4.2 Configuration Selection Σ_{cex}

Generally, a *configuration selection* selects all Dynamic Tuples of a certain Dynamic Data Type that are in a proper Configuration. In detail, a configuration selection is a unary operator that has one input, one output, and a condition. As input, it consumes a set of Dynamic Tuples that all are an instance of the same Dynamic Data Type. The output is a set of Dynamic Tuples that suffice the condition in terms of being in a proper Configuration. This operator can be seen as selection on the schema level, where only these instances will pass the operator that carry a certain structure. The formal operator definition of this operator is given in following Definition 9.

Definition 9 (Configuration Selection)**.** *Let $cex = (t_{cex}, \alpha)$ be a tuple holding a core type t_{cex} and a propositional formula α consisting of Role Types, and the logical operations $\vee$, $\wedge$, and $\neg$. Moreover, let DT be the input set of Dynamic Tuples for which $\forall dt = (e, F, P) \in DT : type(e) = t_{cex}$ holds. Additionally, $\mathcal{A} = \{\overline{rt} \mid rt \in RT\} \cup \{\underline{rt} \mid rt \in RT\}$ is the alphabet of Role Types on which the propositional formula is based on. Then $\Sigma_\alpha(DT) : DT \rightarrow DT$ is a function consuming a set of Dynamic Tuples and produces such set. Formally, it is defined as:*

$$\Sigma_\alpha(DT) = \begin{cases} \{dt \in DT \mid c_{dt} = (t, FT, PT) \wedge rt \in FT\} & \text{for } \alpha \equiv \overline{rt} \wedge rt \in RT \\ \{dt \in DT \mid c_{dt} = (t, FT, PT) \wedge rt \in PT\} & \text{for } \alpha \equiv \underline{rt} \wedge rt \in RT \\ DT \setminus \Sigma_\alpha(DT) & \text{for } \alpha \equiv \neg\beta \\ \Sigma_\beta(DT) \cap \Sigma_\gamma(DT) & \text{for } \alpha \equiv (\beta \wedge \gamma) \\ \Sigma_\beta(DT) \cup \Sigma_\gamma(DT) & \text{for } \alpha \equiv (\beta \vee \gamma) \end{cases} \tag{4.13}$$

To illustrate the configuration selection operator, we employ the following example. Imagine the following two Dynamic Tuples as input set $DT_{example}$.

$$dt_{John} \coloneqq (John, \{\{s_1\}, \{sa_1\}, \{tm_1\}, \{cap_1\}\}, \emptyset) \text{ and}$$
$$dt_{Tim} \coloneqq (Tim, \{\{s_3\}\}, \emptyset)$$

Additionally, cex is given with $cex = (Person, \overline{Student} \wedge \overline{StudentAssistant})$. Please note, the concrete queried Role Types are annotated with a $\overline{rt}$ or $\underline{rt}$ to express their belonging to the set of available Role Types in the filling or participating dimension. Hence, $Student$ and $\overline{Student}$ refer to the same Role Type, the former to this Role Type in general and the latter the Role Type as part of $\mathcal{A}$ in the

filling dimension. However, the basis set of Role Types $\mathcal{A} = \{\overline{Student}, \overline{StudentAssistant}, \dots\} \cup \{\underline{Student}, \underline{StudentAssistant}, \dots\}$, which are basically all available Role Types.

At first, the Configuration is determined for each Dynamic Tuple resulting in:

$$c_{John} = (Person, \{Student, StudentAssistant, TeamMember, Captain\}, \emptyset) \text{ and}$$
$$c_{Tim} = (Person, \{Student\}, \emptyset).$$

Next, both Dynamic Tuples have the core type $t = Person$, which is identical to the operator's core type t_{cex}. Thus, $DT_{example}$ is a valid input set.

To determine $\Sigma_\alpha(DT_{example})$ the fourth case of 4.13 is applied, such that

$$\Sigma_\alpha(DT_{example}) = \Sigma_\beta(DT_{example}) \cap \Sigma_\gamma(DT_{example})$$

with $\beta = \overline{Student}$ and $\gamma = \overline{StudentAssistant}$. Consequently, both, β and γ are atomic and rule one is applied, because both Role Types are part of the set $\overline{rt}$. This results in the following sets.

$$\Sigma_{\overline{Student}}(DT_{example}) = \{dt_{John}, dt_{Tim}\}$$
$$\Sigma_{\overline{StudentAssistant}}(DT_{example}) = \{dt_{John}\}$$

Finally, these sets are intersected to provide the final result $\Sigma_\alpha(DT_{example}) = \{dt_{John}\}$, which is the output of this simple example at the same time.

In sum, the configuration selection Σ filters all Dynamic Tuples of a given input set based on their compliance to a certain Configuration. Invalid, in terms of not being in a valid Configuration, Dynamic Tuples will not pass this operator.

4.4.3 Configuration Projection Π_α

A *configuration projection* cuts out non-queried sets of Roles of a Dynamic Tuple. It consumes a set of Dynamic Tuples and a propositional formula α consisting of Role Types and logical operations. Only these sets of Role will be contained in the output for which the Role Type is also part of α. If a Dynamic Tuple's Configuration and α do not share any Role Type, the output is the Dynamic Tuple's player without any Roles in both dimension. Formally, this operator is defined in Definition 10.

Definition 10 (Configuration Projection)**.** *Let $cex = (t_{cex}, \alpha)$ be a tuple holding a core type t_{cex} and a propositional formula α consisting of Role Types, and the logical operations $\vee$, $\wedge$, and $\neg$. Moreover, let DT be the input set of Dynamic Tuples for which $\forall dt = (e, F, P) \in DT : type(e) = t_{cex}$ holds and F as well as P are the sets of Role sets that are played and featured, respectively. $\mathcal{A} = \{\overline{rt} \mid rt \in RT\} \cup \{\underline{rt} \mid rt \in RT\}$ is the alphabet for the propositional formula. Furthermore, the functions $f\_atoms(\alpha)$ and $p\_atoms(\alpha)$ provide a set of all Role Types in α. $f\_atoms(\alpha)$ for Role Types that are filled and $p\_atoms(\alpha)$ for those that are in the participating dimension. They are defined as:*

$$f\_atoms(\alpha) = \begin{cases} \{rt\} & if \alpha \equiv \overline{rt} \\ \emptyset & if \alpha \equiv \underline{rt} \\ f\_atoms(\beta) \cup f\_atoms(\gamma) & if \alpha \equiv (\beta \wedge \gamma) \text{ or } \alpha \equiv (\beta \vee \gamma) \\ f\_atoms(\beta) & if \alpha \equiv \neg\beta \end{cases} \tag{4.14}$$

$$p\_atoms(\alpha) = \begin{cases} \{rt\} & if \alpha \equiv \underline{rt} \\ \emptyset & if \alpha \equiv \overline{rt} \\ p\_atoms(\beta) \cup p\_atoms(\gamma) & if \alpha \equiv (\beta \wedge \gamma) \; or \; \alpha \equiv (\beta \vee \gamma) \\ p\_atoms(\beta) & if \alpha \equiv \neg\beta \end{cases} \tag{4.15}$$

Moreover, a Dynamic Tuple's Configuration is given by $c_{dt} = (t, FT, PT)$.

The operator $\Pi_\alpha : DT \rightarrow DT$ *consumes a set of Dynamic Tuples and returns such set. Formally, it is defined as:*

$$\begin{aligned} \Pi_\alpha(DT) = & \{dt' = (e, F', P') \mid (e, F, P) \in DT \wedge \\ & F' = \{F^i \in F \mid \exists r \in F^i : type(r) \in f\_atoms(\alpha)\} \wedge \\ & P' = \{P^i \in P \mid \exists r \in P^i : type(r) \in p\_atoms(\alpha)\}\} \end{aligned} \tag{4.16}$$

In detail, F' and P' hold only these sets of Roles for which a Role Type is part of α. To demonstrate the configuration projection a small example is employed. Imagine the Dynamic Tuples dt_{John} and dt_{Tim} and cex as introduced in the previous configuration selection's example and cex is given with $cex = (Person, \overline{Student} \wedge \overline{StudentAssistant})$. Hence, $f\_atoms(\alpha)$ (4.14) returns $\{\overline{Student}, \overline{StudentAssistant}\}$ and $p\_atoms(\alpha) = \emptyset$ (4.15). Furthermore, $DT_{example} = \{dt_{John}, dt_{Tim}\}$ is a valid input set.

At first, dt_{John} is processed. Applying 4.16 results in $F' = \{\{s_1\}, \{sa_1\}\}$. In detail, the Roles s_1 and sa_1 remain in the newly constructed Dynamic Tuple, because their Role Type is part of $f\_atoms(\alpha)$. In contrast, the Roles $\{tm_1\}$ and $\{cap_1\}$ are cut out, because their respective Role Type is not in this set. Furthermore, P is empty and so is P', because $p\_atoms(\alpha)$ returns an empty set. Finally, the resulting Dynamic Tuple is $dt'_{John} = (John, \{\{s_1\}, \{sa_1\}\}, \emptyset)$

Next, dt_{Tim} is reconstructed to dt'_{Tim}. The set F' is populated with set $\{s_3\}$ only, because the Student Role Type is queried. Even if $f\_atoms(\alpha)$ provides more or additional Role Types than FT_{Tim} contains, the resulting F' will always be shrunk, but not extended. Thus, Dynamic Tuples having a different resulting Configuration may pass this operator, as seen in this example. This is useful in case of disjunctive combined Role Types in cex. However, dt_{Tim} does not feature any Roles in P, so $P' = \emptyset$. The resulting Dynamic Tuple is $dt'_{Tim} = (Tim, \{\{s_3\}\}, \emptyset)$.

Finally, the example's operator output is $DT' = \{dt'_{John}, dt'_{Tim}\}$. Notably, as long as DT is a valid input set $|DT| = |DT'|$ holds. Each input Dynamic Tuple will pass this operator either way. Consequently, $\Pi_\alpha(DT)$ does not eliminate any Dynamic Tuple, in fact it shrinks the sets of played and featured Roles of a certain Dynamic Tuple to the provided Role Types in α.

4.4.4 Role Matching κ_α

The *Role matching* operator checks if Dynamic Tuples share some Roles of a certain Role Type rt, which is a parameter of this operator. If a Dynamic Tuple does not find a partner that shares some Roles, it will not pass this operator. This holds for both input sets. In this sense, this operator cuts out all Roles of a queried Role Type that do not find a partner in the opponent input set. Roles of other Role Types than the input Role Type will be fully carried over to the output Dynamic Tuple and are not touched at all. As result, κ provides two output sets consisting of manipulated Dynamic Tuples. The formal definition is presented and explained in Definition 11. This is followed by a detailed explanation using a small example.

Definition 11 (Role Matching)**.** *Let α be a propositional formula consisting of Role Types and logical operation. Additionally, DT_a as well as DT_b are two input sets of Dynamic Tuples, where a Dynamic Tuple $dt_a \in DT_a$ and $dt_b \in DT_b$. On the type level, κ_α is defined for:*

$$\kappa_\alpha : 2^{DT} \times 2^{DT} \rightarrow 2^{DT} \times 2^{DT}$$

On the instance level, κ_{rt} is specified as:

$$\kappa_\alpha : DT_a \times DT_b \rightarrow DT'_a \times DT'_b$$

To compute DT'_a and DT'_b, the Cartesian product of DT_a and DT_b is produced in the first place.

$$DT_{a\times b} := DT_a \times DT_b \tag{4.17}$$

Based on this product a set of triples DT_{share} is defined that contains the set of Roles, which are shared in the Dynamic Tuples dt_a and dt_b.

$$\begin{aligned} DT_{share} :=& \{(dt_a, dt_b, R_{share}) \mid (dt_a, dt_b) \in DT_{a\times b} \wedge \\ & dt_a = (e_a, F_a, P_a) \wedge dt_b = (e_b, F_b, P_b) \wedge \\ & R_{share} := \{r \mid r \in F_a^i \wedge F_a^i \in F_a \wedge r \in P_b^j \wedge P_b^j \in P_b\}\} \end{aligned} \tag{4.18}$$

Next, each tuple is filtered according to α to check either both Dynamic Tuples share Roles as specified in α or not. The resulting set is a triple with both Dynamic Tuples and the shared Roles that comply to α.

$$DT^\alpha_{filter} := \{(dt_a, dt_b, \overline{R_{share}}) \mid (dt_a, dt_b, R_{share}) \in DT_{share} \wedge \overline{R_{share}} := filter_\alpha(R_{share})\} \tag{4.19}$$

The function $filter_\alpha$ is specified for $filter_\alpha : 2^R \rightarrow 2^R$. It consumes a set of Roles and produces another one, such that $filter_\alpha(R) \rightarrow R'$. Formally, it is defined as:

$$filter_\alpha(R_{share}) = \begin{cases} \{r \in R_{share} \mid type(r) = rt\} & \alpha \equiv rt \\ filter_\beta(R_{share}) \cup filter_\gamma(R_{share}) & cond_1 \\ filter_\beta(R_{share}) \cup filter_\gamma(R_{share}) & cond_2 \\ \emptyset & else \end{cases} \tag{4.20}$$

Generally, the Roles coming from the filter process of β and γ are united for one conditions. For readability purposes they are split into two conditions and are explained individually. The first condition $cond_1$ is applied for:

$$\alpha \equiv (\beta \wedge \gamma) \ \wedge filter_\beta(R_{share}) \neq \emptyset \ \wedge filter_\gamma(R_{share}) \neq \emptyset$$

The specifies, the formula in β and γ are conjunctive and both filter processes must not return an empty set. The second condition $cond_2$ is defined as:

$$\alpha \equiv (\beta \vee \gamma) \ \wedge (filter_\beta(R_{share}) \neq \emptyset \ \vee filter_\gamma(R_{share}) \neq \emptyset)$$

This defines that β and γ are combined disjunctively and at most one sub-result can be empty.

DT^{α}_{filter} contains all pairs of dt_a and dt_b in combination with their shared Roles of Role Types in α. Additionally, all pairs that do not match α have been removed from the set. However, a Dynamic Tuple dt_a can find several partners it shares Roles of Role Types in α with. This information is distributed over several tuples in DT^{α}_{filter}. Next, the Roles that find a partner and are distributed over several tuples, are united in separate sets DT^{all}_{a} and DT^{all}_{b}. The former is for Dynamic Tuples of DT_a and the latter for those of DT_b.

$$DT^{all}_{a} := \{(dt_a, R^{all}_{share}) \mid (dt_a, \_, \_) \in DT^{\alpha}_{filter} \wedge R^{all}_{share} := \bigcup_{(dt_a, \_, R_{share}) \in DT^{\alpha}_{filter}} R_{share}\} \tag{4.21}$$

$$DT^{all}_{b} := \{(dt_b, R^{all}_{share}) \mid (\_, dt_b, \_) \in DT^{\alpha}_{filter} \wedge R^{all}_{share} := \bigcup_{(\_, dt_b, R_{share}) \in DT^{\alpha}_{filter}} R_{share}\} \tag{4.22}$$

These two sets contain all tuples of Dynamic Tuples and their respective Roles that find a partner. Next, the output sets are created and the Roles that do not find a partner are cut out of a Dynamic Tuple.

$$DT'_a := \{dt'_a = (e_a, F'_a, P_a) \mid (dt_a, R^{all}_{share}) \in DT^{all}_{a} \wedge dt_a = (e_a, F_a, P_a) \wedge F'_a := reduce(F_a, R^{all}_{share}, rt)\} \tag{4.23}$$

$$DT'_b := \{dt'_b = (e_b, F_b, P'_b) \mid (dt_b, R^{all}_{share}) \in DT^{all}_{b} \wedge dt_b = (e_b, F_b, P_b) \wedge P'_b := reduce(P_b, R^{all}_{share}, rt)\} \tag{4.24}$$

The function $reduce$ is defined for $reduce : 2^{2^R} \times 2^R \rightarrow 2^{2^R}$, consumes a set of Role sets and a set of Roles. It produces a new set of Role sets, such that $reduce(L, R) \rightarrow L'$. It is not limited to any dimension, so it can handle both, F and P as input. Thus, the input set of Role sets is denoted as L.

$$reduce(L, R^{all}_{share}) = \{\overline{reduce}(L^i, R^{all}_{share}) \mid L^i \in L\} \tag{4.25}$$

$$\overline{reduce}(L^i, R^{all}_{share}) = \begin{cases} L^i & \text{if } r \in L^i \cap R^{all}_{share} = \emptyset \\ L^i \cap R^{all}_{share} & \text{else} \end{cases} \tag{4.26}$$

The function $\overline{reduce}$ is defined for $\overline{reduce} : 2^R \times 2^R \rightarrow 2^R$ and consumes two sets of Roles. As output it produces a new set of Roles, such that $\overline{reduce}(L^i, R) \rightarrow L^j$

To explain the operator in detail, we employ an extended example. Imagine both, dt_{John} and dt_{Max} being two Dynamic Tuples of DT_{Person} that are specified by:

$$dt_{John} = (John, \{\{s_1, s_4\}, \{sa_1\}, \{tm_1\}\}, \emptyset)$$
$$dt_{Max} = (Max, \{\{s_2\}\}, \emptyset)$$

Additionally, there exists two Dynamic Tuples dt_{TUD} and dt_{TUC} as elements of $DT_{University}$. These hold the following Roles.

$$dt_{TUD} = (TUD, \emptyset, \{\{s_1, s_3\}, \{sa_1\}\})$$
$$dt_{TUC} = (TUC, \emptyset, \{\{s_4\}\})$$

Furthermore, let $\alpha = Student \wedge StudentAssistant$ be the input formula. The resulting role matching operator is written as:

$$\kappa_{Student \wedge StudentAssistant}(DT_{Person}, DT_{University})$$

For readability reasons the Role Types in α are abbreviated as s and sa, respectively.

To determine the result, the Cartesian product $DT_{Person} \times DT_{University}$ is produced, resulting in the four following tuples.

$$DT_{Person \times University}\{(dt_{John}, dt_{TUD}), (dt_{John}, dt_{TUC}), (dt_{Max}, dt_{TUD}), (dt_{Max}, dt_{TUC})\}$$

Next, DT_{share} is produced resulting in:

$$\begin{aligned} DT_{share} =\{&(dt_{John}, dt_{TUD}, \{s_1, sa_1\}), (dt_{John}, dt_{TUC}, \{s_4\}),\\ &(dt_{Max}, dt_{TUD}, \emptyset), (dt_{Max}, dt_{TUC}, \emptyset)\} \end{aligned}$$

For each tuple, the $filter_{s \wedge sa}(R_{share})$ function is applied, producing the set $DT^{s \wedge sa}_{filter}$.

$$DT^{s \wedge sa}_{filter} = \{(dt_{John}, dt_{TUD}, \{s_1, sa_1\})\}$$

At first, $(dt_{John}, dt_{TUD}, \{s_1, sa_1\})$ is checked for $filter_s(\{s_1, sa_1\})$ and $filter_{sa}(\{s_1, sa_1\})$. Both sets are non-empty, so the Roles s_1 and sa_1 are added to $\overline{R_{share}}$. Next, $(dt_{John}, dt_{TUC}, \{s_4\})$ is checked, which fails, because the check on a **StudentAssistant** Role fails and the returned set is empty. Both Role Types are conjunctively combined, thus, $cond_1$ is applied. Consequently, the whole tuple is discarded and not part this filter step's result. Moreover, dt_{Max} has no shared Roles, neither with dt_{TUD} nor dt_{TUC}, thus, it is completely eliminated.

Based on these results, the pairs of dt_a and dt_b in combination with their shared Roles are split into the two distinct sets DT^{all}_{Person} and $DT^{all}_{University}$. Applying 4.21 and 4.22 on $DT^{s \wedge sa}_{filter}$ results in:

$$\begin{aligned} DT^{all}_{Person} =&\{(dt_{John}, \{s_1, sa_1\})\}\\ DT^{all}_{University} =&\{(dt_{TUD}, \{s_1, sa_1\})\} \end{aligned}$$

As it can be seen, dt_{John} is included only once, but with all Roles united that found a partner in $DT_{University}$. In a final step, the output sets DT'_{Person} and $DT'_{University}$ are generated by applying 4.23 and 4.24.

$$\begin{aligned} DT'_{Person} =&\{dt'_{John} = (John, \{\{s_1\}, \{sa_1\}, \{tm_1\}\}, \emptyset)\}\\ DT'_{University} =&\{dt'_{TUD} = (TUD, \emptyset, \{\{s_1\}, \{sa_1\}\})\} \end{aligned}$$

The output set DT'_{Person} only contains one Dynamic Tuple, because dt_{Max} has been eliminated in the filter step. Additionally, dt'_{John} is shrunk to Roles of Role Types in α that find a partner. In this case, one **Student** and **StudentAssistant** Role of dt'_{John} found a partner. The Role s_4 has been eliminated, because the corresponding university does not feature a **StudentAssistant** Role that is played by $John$.

The same holds for the universities, dt'_{TUD} has lost the **Student** Role s_3, because no partner could be found that match this Role. In detail, $reduce(P_b = \{\{s_1, s_3\}, \{sa_1\}\}, R^{all}_{share} = \{s_1, sa_1\})$ is performed for dt_{TUD}. At first, $\overline{reduce}(L^i = \{s_1, s_3\}, R^{all}_{share} = \{s_1, sa_1\})$ is performed as

subfunction. In particular, the intersection is not empty, thus, the intersection itself is returned. In this case, it is the Role s_1 only, which is going to be part of the output Dynamic Tuple. The same procedure applies to the set of **StudentAssistant** Roles. sa_1 is returned, because the intersection is not empty. As there are no other subsets in this Dynamic Tuple, the $reduce$ procedure terminates.

In sum, the Role matching consumes two sets of Dynamic Tuples and produces two of them. During the process, it manipulates the Dynamic Tuples in a way that only Roles of a queried Role Types in α are part of the output, which find a partner in the opponent set. Dynamic Tuples that do not find a partner in the queried format, are eliminated as result of the failing match.

4.4.5 Relationship Matching Ω_{rst}

The *Relationship matching* operator checks if Dynamic Tuples are related to each other in a certain Compartment by a certain Relationship Type. Additionally, it filters all Roles that participate in a certain Relationship. Hence, unrelated Roles of the participating Role Types are cut out of the corresponding Dynamic Tuples.

It consumes three sets of Dynamic Tuples and the $links$ function for a certain Relationship Type. The first two input sets provide the Dynamic Tuples that need to relate to each other and the third one specifies the Compartment in which the relation has to take place. The fourth input provides the information which Roles are actually related to each other by a particular Relationship. As output is produces three manipulated sets of Dynamic Tuples and an adapted $links$ function. The formal definition and all construction rules for the output sets are presented in the following Definition 12.

Definition 12 (Relationship Matching)**.** *Let DT_a, DT_b, and DT_c be three input sets of Dynamic Tuples and $links$ the function holding the information about connected Roles. The first two input sets provide the Dynamic Tuples that are checked for being in relation to each other. The third one provides the Compartment in which both, the Dynamic Tuples of DT_a and DT_b, are required to be related to each other. This ensures the evaluation Compartment-dependent relationships only. Being in the same Compartment and playing Roles of Role Type that are part of a certain Relationship Type does not mean being in relationship to each other. This particular information in provided by the $links$ function. However, on the type level, Ω_{rst} is defined for:*

$$\Omega_{rst} : 2^{DT} \times 2^{DT} \times 2^{DT} \times links \rightarrow 2^{DT} \times 2^{DT} \times 2^{DT} \times links$$

On the instance level Ω_{RST} is a function that consumes two sets of Dynamic Tuples and a tuple of connected Roles.

$$\Omega_{rst} : DT_a \times DT_b \times DT_c \times links \rightarrow DT'_a \times DT'_b \times DT'_c \times links'$$

To determine DT'_a, DT'_b, and DT'_b the Cartesian product of these Dynamic Tuple input sets is generated as:

$$DT_{a\times b\times c} = DT_a \times DT_b \times DT_c \tag{4.27}$$

Based on this, all Dynamic Tuples and their respective connected Roles in a certain Compartment are identified and collected in DT^{rel}.

$$\begin{aligned} DT^{rel} :=& \{(dt_a, dt_b, dt_c, R_a, R_b) \mid (dt_a, dt_b, dt_c) \in DT_{a\times b\times c} \wedge \\ & dt_a = (e_a, F_a, P_a) \wedge dt_b = (e_b, F_b, P_b) \wedge dt_c = (e_c, F_c, P_c) \wedge \\ & type(e_c) \in C \wedge R_a := linkage(dt_a, dt_b, dt_c, left) \wedge \\ & R_b := linkage(dt_a, dt_b, dt_c, right) \wedge R_a \neq \emptyset \wedge R_b \neq \emptyset\} \end{aligned} \tag{4.28}$$

The linkage function is defined for $DT \times DT \times DT \times (left, right) \rightarrow 2^R$. On the instance level it is defined as $dt \times dt \times dt \times (left, right) \rightarrow R$. It consumes three Dynamic Tuples as well as a marker for the left or right Roles of a Relationship Type. The output is a set of Roles, where each Role is part of the links function and in the playing dimension of the first two Dynamic Tuples. Formally it is defined as:

$$linkage(dt_a, dt_b, dt_c, x) = \begin{cases} \{r_1 \mid (r_1, r_2) \in links(rst, e_c) \wedge \\ r_1 \in F_a^i \wedge F_a^i \in F_a \wedge r_2 \in F_b^j \wedge F_b^j \in F_b\} & \text{if } x = left \\ \{r_2 \mid (r_1, r_2) \in links(rst, e_c) \wedge \\ r_1 \in F_a^i \wedge F_a^i \in F_a \wedge r_2 \in F_b^j \wedge F_b^j \in F_b\} & \text{if } x = right \end{cases} \tag{4.29}$$

The set DT^{rel} contains tuples of Dynamic Tuples that are connected by a certain Relationship in a certain Dynamic Tuple that represents the Compartment, and the Roles they are connected by. Additionally, all not connected Dynamic Tuples have been excluded by checking for non-empty sets in R_a and R_b. A single Dynamic Tuple dt_a may have different Relationships to other Dynamic Tuples dt_b. This information is distributed over several tuples in DT^{rel}. To determine all Roles of a Dynamic Tuple in a certain Relationship the sets DT_a^{all} and DT_b^{all} are created. The Compartments in DT_c featured both Role sets, hence, R_c^{all} is computed as union of both, R_a^{all} and R_b^{all}.

$$\begin{aligned} DT_a^{all} :=& \{(dt_a, R_a^{all}) \mid (dt_a, \_, \_, \_, \_) \in DT^{rel} \wedge \\ & R_a^{all} := \bigcup_{(dt_a, \_, \_, R_a, \_) \in DT^{rel}} R_a\} \end{aligned} \tag{4.30}$$

$$\begin{aligned} DT_b^{all} :=& \{(dt_b, R_b^{all}) \mid (\_, dt_b, \_, \_, \_) \in DT^{rel} \wedge \\ & R_b^{all} := \bigcup_{(\_, dt_b, \_, \_, R_b) \in DT^{rel}} R_b\} \end{aligned} \tag{4.31}$$

$$\begin{aligned} DT_c^{all} :=& \{(dt_c, R_c^{all}) \mid (\_, \_, dt_c, \_, \_) \in DT^{rel} \wedge \\ & R_c^{all} := \bigcup_{(\_, \_, dt_c, R_a, \_) \in DT^{rel}} R_a \cup \bigcup_{(\_, \_, dt_c, \_, R_b) \in DT^{rel}} R_b\} \end{aligned} \tag{4.32}$$

All sets contain a tuple for each Dynamic Tuple that is linked by a certain Relationship to another Dynamic Tuple, and the Roles it is connected by. The final Dynamic Tuple will only hold these Roles that found a partner in a certain Relationship, hence, each Dynamic Tuple is reduced. This holds not only for the Dynamic Tuples that play a certain Role, but also for the Compartment containing these Roles. Thus, DT_a' and DT_b' are manipulated in F_a and F_b, respectively. In contrast, DT_c' is manipulated in P_c.

$$\begin{aligned} DT_a' :=& \{dt_a' = (e_a, F_a', P_a) \mid (dt_a, R_a^{all}) \in DT_a^{all} \wedge dt_a = (e_a, F_a, P_a) \wedge \\ & F_a' := reduce(F_a, R_a^{all})\} \end{aligned} \tag{4.33}$$

$$\begin{aligned} DT_b' :=& \{dt_b' = (e_b, F_b', P_b) \mid (dt_b, R_b^{all}) \in DT_b^{all} \wedge dt_b = (e_b, F_b, P_b) \wedge \\ & F_b' := reduce(F_b, R_b^{all})\} \end{aligned} \tag{4.34}$$

$$\begin{aligned} DT_c' :=& \{dt_c' = (e_c, F_c, P_c') \mid (dt_c, R_c^{all}) \in DT_c^{all} \wedge dt_c = (e_c, F_c, P_c) \wedge \\ & P_c' := reduce(P_c, R_c^{all})\} \end{aligned} \tag{4.35}$$

Additionally, the links information has to be preserved for the result.

$$links_{rst}' := \{(dt_c, R_a, R_b) \mid (\_, \_, dt_c, R_a, R_b) \in DT^{rel}\} \tag{4.36}$$

To demonstrate this operator in detail, a small example is discussed. Assume $DT_{Person1}$, $DT_{Person2}$, and $DT_{University}$ be the input sets of Dynamic Tuples. Additionally, the Relationship Type **supervises** between the Role Types **Student** and **Professor** is queried. Let the sets be populated as following:

$$\begin{aligned} DT_{Person1} =& \{dt_{John} = (John, \{\{s_1, s_4\}, \{sa_1\}\}, \emptyset), dt_{Max} = (Max, \{\{s_2\}\}, \emptyset)\} \\ DT_{Person2} =& \{dt_{Gert} = (Gert, \{\{p_1, p_2\}\}, \emptyset), dt_{Marge} = (Marge, \{\{p_3\}\}, \emptyset)\} \\ DT_{University} =& \{dt_{TUD} = (TUD, \emptyset, \{\{s_1, s_2\}, \{p_1, p_3\}\{sa_1\}\}), \\ & dt_{TUC} = (TUC, \emptyset, \{\{s_4\}\{p_2\}\})\} \end{aligned}$$

Additionally, the links function provides the following tuples:

$$links(supervises, TUD) = \{(p_1, s_1), (p_1, s_3)\}; links(supervises, TUC) = \{(p_4, s_4)\};$$

This corresponding relationship matching operator is specified by:

$$\Omega_{supervises}(DT_{Person1}, DT_{Person2}, DT_{University}, links)$$

At first, the Cartesian product of $DT_{Person1}$, $DT_{Person2}$, and $DT_{University}$ is created.

$$\begin{aligned} DT_{Person1 \times Person2 \times University} =& \{(dt_{John}, dt_{Gert}, dt_{TUD}), \quad (dt_{John}, dt_{Gert}, dt_{TUC}), \\ & (dt_{John}, dt_{Marge}, dt_{TUD}), \quad (dt_{John}, dt_{Marge}, dt_{TUC}), \\ & (dt_{Max}, dt_{Gert}, dt_{TUD}), \quad (dt_{Max}, dt_{Gert}, dt_{TUC}), \\ & (dt_{Max}, dt_{Marge}, dt_{TUD}), \quad (dt_{Max}, dt_{Marge}, dt_{TUC})\} \end{aligned}$$

Next DT^{rel} is produced according to 4.28 as well as 4.29 and consists of the following tuple.

$$DT^{rel} = \{(dt_{John}, dt_{Gert}, dt_{TUD}, \{s_1\}, \{p_1\})\}$$

At most there could be three tuples, because links features three tuples in total. In this scenario only (p_1, s_1) found partner Roles, hence, all others are eliminated. (p_1, s_3) is eliminated because there is no s_3 Role in the input set of $DT_{Person1}$, and (p_4, s_4) because there is no p_4 Role in $DT_{Person2}$.

Determining the sets $DT^{all}_{Person1}$, $DT^{all}_{Person2}$ and $DT^{all}_{University}$ is the next step. These sets are produced following the rules introduced in 4.30, 4.31, and 4.32.

$$\begin{aligned} DT^{all}_{Person1} =& \{(dt_{John}, \{s_1\})\} \\ DT^{all}_{Person2} =& \{(dt_{Gert}, \{p_1\})\} \\ DT^{all}_{University} =& \{(dt_{TUD}, \{s_1, p_1\})\} \end{aligned}$$

$DT^{all}_{Person1}$ contains only dt_{John} with Role s_1 because no other Role found a partner in a Relationship. The same holds for $DT^{all}_{Person2}$. Likewise, $DT^{all}_{University}$ contains only one Dynamic Tuple as well, but with the union the Roles of dt_{John} and dt_{Gert}. Next, the output Dynamic Tuples are generated by

reducing the input Dynamic Tuples to the Role that found a partner. This results in the three output sets $DT'_{Person1}$, $DT'_{Person2}$, and $DT'_{University}$ as described in 4.33, 4.34, and 4.35.

$$\begin{aligned} DT'_{Person1} =& \{dt_{John} = (John, \{\{s_1\}, \{sa_1\}\}, \emptyset)\} \\ DT'_{Person2} =& \{dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\} \\ DT'_{University} =& \{dt_{TUD} = (TUD, \emptyset, \{\{s_1\}, \{p_1\}, \{sa_1\}\})\} \end{aligned}$$

Finally, the links information is preserved as output according to 4.36.

$$links'_{supervises} = \{(dt_{TUD}, \{s_1\}, \{p_1\})$$

In sum, the Relationship matching operator filters Dynamic Tuples by their participation in a certain Relationship. It reduces matching Dynamic Tuples to the Roles that find a partner. Moreover, it directly leverages the links function of the instance model to filter these Dynamic Tuples. All Dynamic Tuples that do not find a partner in a certain Relationship will be eliminated of the output sets.

4.4.6 Dynamic Data Type Union τ

To unite Dynamic Tuples of different Dynamic Data Types in a shared stream, especially if the Dynamic Data Types share certain Role Types in a particular dimension, the τ operator is introduced. It consumes two sets of Dynamic Tuples and the *type* function. To perform the streams union, a new union type is created, which is used to overwrite the *type* function for the corresponding Dynamic Tuple cores.

Definition 13 (Dynamic Data Type Union)**.** *Let DT_a and DT_b two input streams of Dynamic Tuples of different Dynamic Data Types. These Dynamic Data Types are defined by $DDT_a := (t_a, FT_a, PT_a)$ and $DDT_b := (t_b, FT_b, PT_b)$. Moreover, the type of the base data is required. The operator τ is on the type level defined for:*

$$\tau : 2^{DT} \times 2^{DT} \times type \rightarrow 2^{DT} \times type$$

On the instance it is defined as:

$$\tau : DT \times DT \times type \rightarrow DT \times type$$

At first, the union of both input streams in a combined set, denoted as $DT_{a \cup b}$, is created.

$$DT_{a \cup b} := DT_a \cup DT_b \tag{4.37}$$

This unites both streams, but with different player types as core. To align these types, the union type consisting of attributes of both player types is created as t'.

$$t' := t_a \cup t_b \tag{4.38}$$

Each type is defined by a name and a set of attributes. Thus, the union type represents the union of both, t_a and t_b, attribute sets.

To compute the output, the t' has to be assigned to each core of the Dynamic Tuple union set.

$$DT' := \{dt \mid dt = (e, F, P) \in DT_{a \cup b} \wedge type(e) := t'\} \tag{4.39}$$

The core of a Dynamic Tuple is not manipulated, but a new type is assigned to it. Moreover, the Role Types in both dimensions remain untouched, hence the Roles are not manipulated at all.

To demonstrate this operator, assume the two input sets of Dynamic Tuples $DT_{Lecture}$, and $DT_{Seminar}$.

$$DT_{Seminar} = \{dt_{sem1} = (sem1, \{\{sc_1\}\}, \emptyset)\}$$
$$DT_{Lecture} = \{dt_{l1} = (l_1, \{\{sc_2\}\}, \emptyset)\}$$

At first the union of both sets is created according to 4.37 resulting in $DT_{Seminar \cup Lecture}$.

$$DT_{Seminar \cup Lecture} = \{dt_{sem1}, dt_{l1}\}$$

Next, the union type t' is created by applying rule 4.38.

$$Sem\_L' = \{Credits : Byte, Date : Date, Time : String, Room : String\}$$

Notably, the attribute $Credits$ is included only, because it is shared of both player types. This union type is now assigned to each entity core, as defined in rule 4.39. This is also the last step within the τ operator.

$$DT' = \{dt_{sem1}, dt_{l1}\}$$
$$type(sem_1) = Sem\_L'$$
$$type(l_1) = Sem\_L'$$

In sum, the τ operator consumes two input sets of Dynamic Tuples, unites them in one set, and overwrites the *type* function for the corresponding Dynamic Tuple cores. The new type is the union of both player type attribute sets. Empty attributes of a core, especially those that have not been present in the original type, are represented as `NULL` values.

4.4.7 Dynamic Tuple Difference Without Role Difference $\setminus_{R-}$

The difference between two sets of Dynamic Tuples, without considering the Role at all, are these Dynamic Tuples of the first input set that do not share the core with an entity of the second input set. The formal definition is presented in Definition 14.

Definition 14 (Dynamic Tuple Difference Without Role Difference)**.** *Let DT_a and DT_b be two input sets of Dynamic Tuples. The operator $\setminus_{R-}$ is defined as $\setminus_{R-} : 2^{DT} \times 2^{DT} \rightarrow 2^{DT}$ on the type level. On the instance level it consumes two sets of Dynamic Tuples and produces one set, such that $\setminus_{R-} : DT \times DT \rightarrow DT$.*

The output set DT' is determined by the following equation.

$$DT' := \{dt_a \mid dt_a = (e_a, F_a, P_a) \in DT_a \wedge dt_b = (e_b, F_b, P_b) \in DT_b \wedge \nexists dt_b : e_b = e_a\} \quad (4.40)$$

For example, imagine the two input sets $DT_{Person1}$ and $DT_{Person2}$ that are populated as follows.

$$DT_{Person1} = \{dt_{John} = (John, \ldots, \emptyset), dt_{Gert} = (Gert, \ldots, \emptyset), dt_{Max} = (Max, \ldots, \emptyset)\}$$
$$DT_{Person2} = \{dt_{Tim} = (Tim, \ldots, \emptyset), dt_{Kai} = (Kai, \ldots, \emptyset), dt_{John} = (John, \ldots, \emptyset)\}$$

The corresponding operator is specified as: $DT_{Person1} \setminus_{R-} DT_{Person2}$. The output DT' consists of two Dynamic Tuple, dt_{Gert} and dt_{Max}. The other Dynamic Tuple of $DT_{Person1}$, dt_{John}, is eliminated according to 4.40, because there is a Dynamic Tuple in DT_b that shares the same core. Obviously, dt_{John} is part of both sets, hence, it must be excluded from the result.

In sum, this operator follows the traditional set theory definitions of a difference, but has slightly different semantics. In detail, the reference point in each set is the core of an entity and not the element itself. Hence, it checks for the existence of the same core in the second input set.

4.4.8 Dynamic Tuple Difference With Role Difference $\setminus_R$

This operator produces the difference between two input sets of Dynamic Tuples, but also considers the difference of Role Types. Thus, Dynamic Tuples sharing the same entity core are not completely eliminated, rather the difference of their Roles is produced. This lifts the semantics of a difference to the level of Roles.

Definition 15 (Dynamic Tuple Difference With Role Difference)**.** *Let DT_a and DT_b be two input sets of Dynamic Tuples. The operator $\setminus_R$ is defined as $\setminus_R : 2^{DT} \times 2^{DT} \rightarrow 2^{DT}$ on the type level. On the instance level it consumes two sets of Dynamic Tuples and produces one set, such that $\setminus_R : DT \times DT \rightarrow DT$.*

At first, the Cartesian product of DT_a and DT_b is generated.

$$DT_{a\times b} := DT_a \times DT_b \tag{4.41}$$

On this basis the set of equal entities is collected in the set DT_{equal}.

$$\begin{aligned} DT_{equal} :=& \{(dt_a, dt_b) \mid (dt_a, dt_b) \in DT_{a\times b} \wedge \\ & dt_a = (e_a, F_a, P_a) \wedge dt_b = (e_b, F_b, P_b) \wedge e_a = e_b\} \end{aligned} \tag{4.42}$$

This builds the foundation to determine the sets of Roles sets in both dimension, but only for Dynamic Tuples that share the core.

$$\begin{aligned} DT'_{equal} :=& \{dt' = (e_a, F', P') \mid (dt_a, dt_b) \in DT_{a\times b} \wedge dt_a = (e_a, F_a, P_a) \wedge dt_b = (e_b, F_b, P_b) \wedge \\ & F' := group(flat(F_a) \setminus flat(F_b)) \wedge P' := group(flat(P_a) \setminus flat(P_b))\} \end{aligned} \tag{4.43}$$

The function flat unpacks sets of Role sets into one set of Roles, which simplifies the intersection of both sets of Role sets. Hence, it is defined for $flat : 2^{2^R} \rightarrow 2^R$ and consumes a set of Role sets and produces a set of Roles. On the instance level it is defined as $flat : 2^R \rightarrow R$. The group function is the opposite of flat and consumes a set of Roles and produces a set of Role sets, grouped by their Role Type. Consequently, the type of this function is $group : 2^R \rightarrow 2^{2^R}$ and the instance definition $group : R \rightarrow 2^R$. These functions work for both dimension, thus, the input is denoted as L to avoid confusion. Formally, these functions are defined as follows.

$$flat(L) := \{r \mid r \in L^i \wedge L^i \in L\} \tag{4.44}$$

$$group(L) := \{\{r \mid (r, rt) \in \overline{L}\} \mid rt \in RT\} \text{ with } \overline{L} := \{(r, rt) \mid r \in L \wedge rt = type(r)\} \tag{4.45}$$

The final result DT' is specified as all Dynamic Tuples of DT_a that do not share their core with a Dynamic Tuple of DT_b, plus all Dynamic Tuples of DT_a that share the core, but manipulated in a way that only the difference of Roles in both dimensions is returned.

$$DT' := (DT_a \setminus_{R-} DT_b) \cup DT'_{equal} \tag{4.46}$$

As demonstration the following two input sets $DT_{Person1}$ and $DT_{Person2}$ are given.

$$\begin{aligned} DT_{Person1} =& \{dt_{John} = (John, \{\{s_1, s_4\}\{sa_1\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\} \\ DT_{Person2} =& \{dt_{Marge} = (Marge, \{\{p_3\}\}, \emptyset), dt_{John} = (John, \{\{s_1\}\}, \emptyset), \} \end{aligned}$$

This example operator is specified by $DT_{Person1} \setminus_R DT_{Person2}$. At first the Cartesian product of both input sets creates four tuples.

$$\begin{aligned} DT_{Person1 \times Person2} =& \{(dt_{John}, dt_{Marge}), (dt_{John}, dt_{John}), \\ & (dt_{Gert}, dt_{Marge}), (dt_{Gert}, dt_{John})\} \end{aligned}$$

Only one tuple consists of Dynamic Tuples sharing a core. Hence, equation 4.42 produces $DT_{equal} = \{(dt_{John}, dt_{John})\}$. For this the manipulated output Dynamic Tuple is created in DT'_{equal} with respect to 4.43.

$$DT'_{equal} = \{dt'_{John} = (John, \{\{s_4\}\{sa_1\}\}, \emptyset)\}$$

The function $flat(\{s_1, s_4\}\{sa_1\})$ returns $F_a^{flat} = \{s_1, s_4, sa_1\}$ (see equation 4.44). $F_b^{flat} = \{s_1\}$ is created in the same way. The difference of both, F_a^{flat} and F_b^{flat}, is $\{s_4, sa_1\}$. These Roles are now regrouped by their Role Types, which is processed by the *group* function according to 4.45. Consequently, F' is populated by $\{\{s_4\}\{sa_1\}\}$. $P' = \emptyset$, because the inputs are also empty sets.

The final output set consists of the regular Dynamic Tuple difference without Roles united with DT'_{equal}, as defined in 4.46.

$$DT' = \{dt'_{John} = (John, \{\{s_4\}\{sa_1\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\}$$

The regular difference provides dt_{Gert} only. Additionally, dt'_{John} is included in this output set. In total, this operator provides functionality to determine a difference on the level of Roles.

4.4.9 Dynamic Tuple Intersection $\cap_{\circ}^{RT_a, RT_b}$

The *Dynamic Tuple intersection* searches for shared Dynamic Tuples between the two input sets and produces one output stream consisting of shared Dynamic Tuples only. These shared Dynamic Tuples may occur in different versions, which means they have different Roles in their dimensions. To construct only a single output stream, the Roles of both Dynamic Tuples have to be processed and united in new dimensions. This processing is controlled by the three parameters this operator features. At first, there are two parameters having a set of Role Types each, RT_a and RT_b. All Role having a Role Type in RT_a are directly taken from the Dynamic Tuple coming from the first input and put into the output Dynamic Tuple. In contrast, Roles with a Role Type in RT_b are directly transferred from the second input stream's Dynamic Tuple to the output one. These Role Type sets are used to protect previous dimension manipulations, for instance by a Role matching operator, from being overwritten. Secondly, all other Roles are processed according to the operation provided by the third parameter $\circ$, which can be a union or intersection. The formal definition is given in the following Definition 16.

Definition 16 (Dynamic Tuple Intersection)**.** *Let DT_a and DT_b be two input sets of Dynamic Tuples. This operator has three parameters, $\circ = \{\cup, \cap\}$, RT_a, and RT_b. The first one specifies the operation on the level of Roles, which becomes important in case the same Dynamic Tuple is included in both sets, but in different versions. RT_a specifies Role Types that are used to copy Roles from the Dynamic Tuple included in DT_a into the united and output Dynamic Tuples. In contrast, RT_b defines those Role Types that are utilized to transfer the Roles from the Dynamic Tuple coming from DT_b. Additionally, both sets of Role Types distinguish between Role Types in the playing and featuring dimension such that those of the playing dimension are denoted with an overline and those of the featuring dimension with an underline.*

However, this operator is defined for $\cap_{\circ}^{RT_a,RT_b} : 2^{DT} \times 2^{DT} \rightarrow 2^{DT}$. It consumes two sets of Dynamic Tuples and produces one set of those, such that $\cap_{\circ}^{RT_a,RT_b} : DT \times DT \rightarrow DT$.

At first, the Cartesian product of DT_a and DT_b is generated to find the Dynamic Tuples included in both sets.

$$DT_{a\times b} := DT_a \times DT_b \tag{4.47}$$

On this basis the set of equal entities is determined in the set DT_{equal} to represent the intersection on the Dynamic Tuple level.

$$\begin{aligned} DT_{equal} :=& \{(dt_a, dt_b) \mid (dt_a, dt_b) \in DT_{a\times b} \wedge \\ & dt_a = (e_a, F_a, P_a) \wedge dt_b = (e_b, F_b, P_b) \wedge e_a = e_b\} \end{aligned} \tag{4.48}$$

The output set DT' is generated on the foundation of DT_{equal}. For each pair, a new Dynamic Tuple is constructed with the entity of dt_a and the union of various sets of Roles in both dimensions. This unification is represented in F' and P'. To protect overwriting for certain Roles of certain Role Types, each dimension is constructed of three sets of Roles. At first, these Roles that neither have a Role Type represented in RT_a nor in RT_b. Secondly, the Role having a Role Type included in RT_a and thirdly, all Roles being of a Role Type contained in RT_b.

$$\begin{aligned} DT' :=& \{dt' = (e', F', P') | (dt_a, dt_b) \in DT_{equal} \wedge \\ & dt_a = (e_a, F_a, P_a) \wedge dt_b = (e_b, F_b, P_b) \wedge e' = e_a \wedge \\ & F' := group(F_- \cup F_a^{RT} \cup F_b^{RT}) \wedge P' := group(P_- \cup P_a^{RT} \cup P_b^{RT})\} \end{aligned} \tag{4.49}$$

The corresponding subsets of Roles are determined as follows. Please note, RT describes the complete set of available Role Types. Moreover, the flat and group function are applied as defined in 4.44 and 4.45, respectively.

$$F_- := \{r \in (flat(F_a) \circ flat(F_b)) \mid type(r) \in (RT \setminus RT_a \setminus RT_b)\} \tag{4.50}$$

$$F_a^{RT} := \{r \in flat(F_a) \mid type(r) \in \overline{RT_a}\} \tag{4.51}$$

$$F_b^{RT} := \{r \in flat(F_b) \mid type(r) \in \overline{RT_b}\} \tag{4.52}$$

$$P_- := \{r \in (flat(P_a) \circ flat(P_b)) \mid type(r) \in (RT \setminus RT_a \setminus RT_b)\} \tag{4.53}$$

$$P_a^{RT} := \{r \in flat(P_a) \mid type(r) \in \underline{RT_a}\} \tag{4.54}$$

$$P_b^{RT} := \{r \in flat(P_b) \mid type(r) \in \underline{RT_b}\} \tag{4.55}$$

F_- and P_- consists of Roles that neither have a Role Type represented in RT_a nor RT_b. In contrast, F_a^{RT} and P_a^{RT} include only Roles coming from the first input's Dynamic Tuple and have a proper Role Type as specified in RT_a. Likewise, F_b^{RT} and P_b^{RT} contain only Roles of Role Types included in RT_b. Additionally, these Roles have to come from the Dynamic Tuple of the second input. Finally, these sets of Roles are united and grouped by their Role Types to represent the output Dynamic Tuples.

To demonstrate this operator and its variety in parameter assignments as well as the consequences of these assignments, three different examples are employed. However, all three examples are based on the same input sets, thus, only the output DT' changes from example to example.

Assume two input sets $DT_{Person1}$ and $DT_{Person2}$ having the following Dynamic Tuples.

$$\begin{aligned} DT_{Person1} =& \{dt_{John} = (John, \{\{s_1, s_4\}\{sa_1\}\}, \emptyset), dt_{Max} = (Max, \{\{s_2\}\}, \emptyset)\} \\ DT_{Person2} =& \{dt_{John} = (John, \{\{s_1\}, \{sa_3\}\{tm_1\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\} \end{aligned}$$

The Cartesian product of both input sets creates four tuples.

$$\begin{aligned} DT_{Person1 \times Person2} =& \{(dt_{John}, dt_{John}), (dt_{John}, dt_{Gert}), \\ & (dt_{Max}, dt_{John}), (dt_{Max}, dt_{Gert})\} \end{aligned}$$

Next, DT_{equal} selects all tuples of the Cartesian product that have equal cores. Hence, only one tuple is contained in this set and all others are eliminated.

$$DT_{equal} = \{(dt_{John}, dt_{John})\}$$

Unrestricted Role Union

The first example represents an unrestricted Role union. Unrestricted means no Role Types are provided, neither in RT_a nor in RT_b. The parametrized operator is represented as

$$\cap_{\cup}^{\emptyset,\emptyset}(DT_{Person1}, DT_{Person2})$$

Next, the three Role sets for each dimension are determined. F_- contains all Roles of Role Types that are not part of RT_a and RT_b according to 4.50 In this example these Role Type sets are empty, hence, all Role Types are qualified for this set. The parameter $\circ$ is populated with a $\cup$, thus, the Roles are united. Consequently, $F_- = \{s_1, s_4, sa_1, sa_3, tm_1\}$. Because RT_a and RT_b are empty F_a^{RT} and F_b^{RT} are empty as well (see rule 4.51 and 4.52). The featuring dimension of the example's Dynamic tuple is empty in both version, so the output dimension is and P_-, P_a^{RT}, and P_b^{RT} return empty sets.

The resulting output DT' consists of one Dynamic Tuple.

$$DT' = \{dt'_{John} = (John, \{\{s_1, s_4\}\{sa_1, sa_3\}, \{tm_1\}\}, \emptyset)\}$$

Restricted Role Union

The second example explains this operator as restricted Role union. This means, at least one set out of RT_a and RT_b is not empty. In this example, $RT_a = \{\overline{StudentAssistant}\}$ and $RT_b = \emptyset$. The union specifies that $\circ$ is populated with a $\cup$. The parametrized operator is specified by

$$\cap_{\cup}^{\{\overline{StudentAssistant}\},\emptyset}(DT_{Person1}, DT_{Person2})$$

To determine dt'_{John}, the dimension have to be computed with respect to the rules 4.50 to 4.55. At first F_- is determined and includes all Roles, except of Roles of the Role Type $\overline{StudentAssistant}$. Consequently, $F_- = \{s_1, s_4, tm_1\}$. Secondly, all Roles with Role Types included in RT_a are collected in F_a^{RT} resulting in $F_a^{RT} = \{sa_1\}$. Finally, F_a^{RT} and the whole featuring dimension are empty according to rules 4.52 to 4.55.

The final output DT' consists of dt'_{John} only, which has modified dimensions. F' is the unification of F_-, F_a^{RT}, and F_b^{RT}. P' is calculated likewise, but with sets of the featuring dimension. This results in the following output.

$$DT' = \{dt'_{John} = (John, \{\{s_1, s_4\}\{sa_1\}, \{tm_1\}\}, \emptyset)\}$$

Restricted Role Intersection

The last example of this operator demonstrates the restricted intersection on the level of Roles. Assume $RT_a = \{\overline{StudentAssistant}\}$ and $RT_b = \{\overline{TeamMember}\}$. The parameter $\circ$ is specified with a $\cap$. Thus, the operator is denoted as:

$$\cap_{\cap}^{\{\overline{StudentAssistant}\},\{\overline{TeamMember}\}}(DT_{Person1}, DT_{Person2})$$

At first F_- is determined by intersecting all Roles that are not of a Role Type in RT_a and RT_b. This applies for the Roles $\{s_1, s_4\}$ of the dt_a's version of dt_{John} and $\{s_1\}$ for the dt_b's version. The intersection results in $F_- = \{s_1\}$. Secondly, F_a^{RT} is calculated as $F_a^{RT} = \{sa_1\}$ and $F_b^{RT} = \{tm_1\}$. The unification of these sets results in the F' dimension for dt'_{John} with $F' = \{\{s_1\}, \{sa_1\}, \{tm_1\}$. As in the example discussed previously, the featuring dimension is empty, such that $P' = \emptyset$.

The output of this parametrized Role intersection operator is given in DT'.

$$DT' = \{dt'_{John} = (John, \{\{s_1\}\{sa_1\}, \{tm_1\}\}, \emptyset)\}$$

In sum, this operator unites two sets of Dynamic Tuples into a single one while only shared Dynamic Tuples are consolidated into the new one. To provide control over handling the Roles in the different dimension, it offers three parameters. A parameter that specifies which Roles should come from the Dynamic Tuple of the first input only and one for those coming from the second input. Finally, all other Roles are treated as denoted in the third parameter. This fine-grained control mechanism allows for protecting previous Role selections to be overwritten by unfiltered Roles coming from the other input set.

4.4.10 Dynamic Tuple Union $\cup_{\circ}^{RT_a,RT_b}$

The *Dynamic Tuple union* operator takes two input streams of Dynamic Tuples and unite them into a single one. For those Dynamic Tuples that are shared between both input sets several options exist to unite them. However, this operator is similar to the Dynamic Tuple intersection operator, but additionally includes the Dynamic Tuples of both input streams that are not shared between these sets. Hence, it is build on the Dynamic Tuple intersection operator. The formal definition is presented as follows.

Definition 17 (Dynamic Tuple Union)**.** *Let DT_a and DT_b be two input sets of Dynamic Tuples. Moreover, there exist three parameters for this operator; $\circ = \{\cup, \cap\}$ that defines the operation on the Role level, RT_a to specify the Roles that come from the Dynamic Tuple in DT_a, and RT_b to describe the Roles that are transfers from the Dynamic Tuple in DT_b. Both Role Type sets distinguish between Role Types in the two dimensions. Overlined Role Types specify the filling dimension and underlined ones the participating dimension.*

However, this operator is defined for $\cup_{\circ}^{RT_a,RT_b} : 2^{DT} \times 2^{DT} \rightarrow 2^{DT}$. It consumes two sets of Dynamic Tuples and produces one set of those, such that $\cup_{\circ}^{RT_a,RT_b} : DT \times DT \rightarrow DT$.

The output of this operator is defined as:

$$DT' := \{DT'_{equal} \cup (DT_a \setminus_{R-} DT'_{equal}) \cup (DT_b \setminus_{R-} DT'_{equal})\} \tag{4.56}$$

with DT'_{equal} being the Dynamic Tuple intersection of both input streams.

$$DT'_{equal} := \cap_{\circ}^{RT_a,RT_b}(DT_a, DT_b) \tag{4.57}$$

This intersection determines the shared Dynamic Tuples and manipulates the dimensions according to the operation given by $\circ$. All other Dynamic Tuples are not shared between both input sets and are passed to the output stream without any manipulation.

Two examples are discussed to show the operator's functionality. Assume two input sets $DT_{Person1}$ and $DT_{Person2}$ having the following Dynamic Tuples.

$$DT_{Person1} = \{dt_{John} = (John, \{\{s_1, s_4\}\{sa_1\}\}, \emptyset), dt_{Max} = (Max, \{\{s_2\}\}, \emptyset)\}$$
$$DT_{Person2} = \{dt_{John} = (John, \{\{s_1\}, \{sa_3\}\{tm_1\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\}$$

These sets are the same as in the example of the Dynamic Tuple intersection operator (see 4.4.9).

Unrestricted Role Union

For the first example additionally assume $\circ = \cup$ and RT_a as well RT_b to be empty. The corresponding operator is denoted as $\cup_{\cup}^{\emptyset,\emptyset}(DT_{Person1}, DT_{Person2})$.

For creating the output set DT' three subsets are computed and united as defined in 4.56. The first subset is DT'_{equal}, which is determined by using the Dynamic Tuple intersection operator according to the rule defined in 4.57. The parameters of the Dynamic Tuple union are directly passed to the Dynamic Tuple intersection operator, such that $DT'_{equal} = \cap_{\cup}^{\emptyset,\emptyset}(DT_{Person1}, DT_{Person2})$. The output is the same as in the first example of the intersection operator. Thus, $DT'_{equal} = \{dt'_{John} = (John, \{\{s_1, s_4\}\{sa_1, sa_3\}, \{tm_1\}\}, \emptyset)\}$.

As a next step, the Dynamic Tuple difference of DT_a and DT'_{equal} is determined as second subset. This results in $DT_a \setminus_{R-} DT'_{equal} = \{dt_{Max} = (Max, \{\{s_2\}\}, \emptyset)\}$. Finally, the third subset is the Dynamic Tuple difference between DT_b and the shared Dynamic Tuples collected in DT'_{equal}. This results in $DT_a \setminus_{R-} DT'_{equal} = \{dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\}$.

These subsets are united to represent DT'.

$$\begin{aligned} DT' = &\{dt'_{John} = (John, \{\{s_1, s_4\}\{sa_1, sa_3\}, \{tm_1\}\}, \emptyset), \\ &dt_{Max} = (Max, \{\{s_2\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\} \end{aligned}$$

Restricted Role Intersection

The second example demonstrates a restricted Role intersection for the Role Type sets $RT_a = \{\overline{StudentAssistant}\}$ and $RT_b = \{\overline{TeamMember}\}$. Due to the intersection, $\circ = \cap$. The resulting Dynamic Tuple union operator is specified as $\cup_{\cap}^{\overline{StudentAssistant}, \overline{TeamMember}}$. The only difference to example discussed previously is the Dynamic Tuple included in DT'_{equal}.

However, DT'_{equal} is the output of a Dynamic Tuple intersection operator with the same parameters as this union operators has. Hence, $DT'_{equal} = \cap_{\cap}^{\overline{StudentAssistant}, \overline{TeamMember}}(DT_{Person1}, DT_{Person2})$. This represents the same result as the third example for the Dynamic Tuple intersection operator (see 4.4.9). Precisely, $DT'_{equal} = \{dt'_{John} = (John, \{\{s_1\}\{sa_1\}, \{tm_1\}\}, \emptyset)\}$ The subsets for the difference of DT_a to DT'_{equal} as well as DT_b to DT'_{equal} remain the same.

Consequently, this example's output is the following.

$$\begin{aligned} DT' = &\{dt'_{John} = (John, \{\{s_1\}\{sa_1\}, \{tm_1\}\}, \emptyset), \\ &dt_{Max} = (Max, \{\{s_2\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset)\} \end{aligned}$$

Concluding this operator, it provides functionality to unite two sets of Dynamic Tuples and for shared Dynamic Tuples it falls back on the Dynamic Tuple intersection operator. The parameters provided to this operator are directly passed to the intersection one. This guarantees a fine-grained control mechanism for uniting the Role dimensions of shared Dynamic Tuples in this operator as well. All non-shared input Dynamic Tuples are included in the output, too, but without any dimension manipulation.

4.4.11 Attribute Selection $\sigma_{predicate}^{t, RT_{overlap}}$

To filter entire Dynamic Tuples or only Roles of them based on their values, the *attribute selection* operator is introduced. A Dynamic Tuple is filtered in case the parameter t references a core type. In contrast, if t relates to a Role Type, Roles of this Role Type are filtered. Moreover, an arbitrary predicate is provided, which is evaluated for the core or the Role. This predicate may be related the type to be filtered or it may be not. This totally depends on the query. Additionally, a set ov Role Type $RT_{overlap}$ has to be provided to the operator, that specifies in which Role Type different input streams should overlap. In case of no overlap, an empty set can be provided. However, the attribute selection operator consumes as many inputs as necessary to evaluate the predicate. For instance, in case a $address$ of a **Person** is compared to the $location$ of a **University** that share the Role Type **Student**, an input consisting of Dynamic Tuples with **Persons** and an input for **University** is required. The more complex the predicate is the more inputs are required. The formal definition of this operator is presented in 18 and explained afterwards by using an example operator.

Definition 18 (Attribute Selection). *Let DT_a be the input set of Dynamic Tuples that is going to be manipulated and $DT_b \dots DT_n$ additional sets that are required to evaluate the predicate. In case the attribute selection predicate involves Dynamic Tuples of DT_a only, no additional sets are provided. Thus, this operator consumes a set of Dynamic Tuple sets and returns only one manipulated set of Dynamic Tuples. As parameters, it consumes a type t that is filtered, a set of Role Types $RT_{overlap}$ that specifies the overlapping point of several input sets, and an arbitrary predicate $predicate$. The set of overlapping Role Types may be empty, specifying that the input sets are not related to each other. The operator $\sigma^{t}_{predicate}$ is on the type level defined for $\sigma^{t,RT_{overlap}}_{predicate} : 2^{DT_1} \times \dots \times 2^{DT_n} \rightarrow 2^{DT}$. The concrete operator instance is defined as $\sigma^{t,RT_{overlap}}_{predicate} : DT_1 \times \dots \times DT_n \rightarrow DT$.*

At first, the Dynamic Tuples of all inputs are flatted and put together as n-ary Cartesian product. The flattening is described by the operator $\chi(DT)$. It is defined as follows:

$$\chi(DT) := \left\{ e \times \prod_{i=1}^{n} F_i \times \prod_{j=1}^{m} P_j \mid dt = (e, F, P) \in DT \wedge F_i \in F \wedge P_j \in P \right\} \tag{4.58}$$

This produces the Cartesian product of an entity with all its Roles for $n = |F|$ and $m = |P|$. The Cartesian product on the level of Roles as a regular n-ary Cartesian product, which is defined as follows.

$$\prod_{i=1}^{n} R_i = R_1 \times \dots \times R_n := \{(r_1, \dots, r_n) \mid r_i \in R_i\} \tag{4.59}$$

Using these two definitions, the Cartesian product on all Dynamic Tuple input sets B is defined as follows.

$$B := \{\chi(DT_a) \times \dots \times \chi(DT_n)\} \tag{4.60}$$

On this B, the predicate is evaluated resulting in the set of tuples B_{pred}, which consists of qualified tuples only.

$$\begin{aligned} B_{pred} := \{(e, r_{a_1}, \dots, r_{a_n}) \mid b = (e_a, r_{a_1}, \dots, r_{a_n}, \dots, e_h, r_{h_1}, \dots, r_{h_m}) \in B \wedge \\ overlap(b, RT_{overlap}) \wedge pred(b, predicate)\} \end{aligned} \tag{4.61}$$

The $overlap$ function checks if a tuple in B holds the same Role of a certain Role Type twice, which indicates an overlap. This check is required to ensure that only the overlapping Dynamic Tuples are considered as potential predicate fulfiller. Furthermore, the function $pred$ evaluates the predicate on the given tuple b. It becomes true, in case the predicate is fulfilled and false otherwise.

$$overlap(b, RT) := \forall rt_i \in RT \; \exists b_{[r_x]} \wedge \exists b_{[r_y]} : r_x = r_y \wedge type(r_x) = rt_i \wedge x \neq y \tag{4.62}$$

The set B_{pred} includes all entity and Role information that correspond to the input set DT_a. This information is distributed of several tuple in B_{pred}, hence, we unite this distributed information in a new Dynamic Tuple. This is constructed depending on the filter parameter t.

$$DT_{pred} := \begin{cases} \{dt = (e_a, \emptyset, \emptyset) \mid b \in B_{pred}\} & if\, t \equiv nt \vee ct \\ \{dt = (e_a, F, \emptyset) \mid b \in B_{pred} \wedge F := collect(e_a, B_{pred}, t)\} & if\, t \equiv \overline{rt} \\ \{dt = (e_a, \emptyset, P) \mid b \in B_{pred} \wedge P := collect(e_a, B_{pred}, t) & if\, t \equiv \underline{rt} \end{cases} \tag{4.63}$$

The $collect$ function collects all Roles of a given Role Type rt for a given entity core e out of a set B.

$$collect(e, B_{pred}, rt) := \bigcup_{(e,\_,r_{a_i},\_) \in B_{pred}} r_{a_i} \wedge type(r_{a_i}) = rt \tag{4.64}$$

Finally, the computed set of qualified Dynamic Tuples with respect to a given predicate is intersected with input set DT_a to filter only the valid ones in the output set DT'_a. This step depends on the input parameter t, too. Additionally, to keep Role of unfiltered Role Types as they are, we define $RT' := \overline{RT} \cup \underline{RT}$ a union of all Role Types, once in the filling and once in the participating dimension, which is denoted by the overline and underline annotations, respectively.

$$DT'_a := \begin{cases} \cap_{\cap}^{RT',\emptyset}(DT_a, DT_{pred}) & if\, t \equiv nt \vee ct \\ \cap_{\cap}^{RT' \setminus t,\{t\}}(DT_a, DT_{pred}) & if\, t \equiv \overline{rt} \vee \underline{rt} \end{cases} \tag{4.65}$$

In detail, in case the entity is filtered as whole data structure, a Dynamic Tuple intersection by using all Roles from the DT_a input set. This excludes all Dynamic Tuples of DT_a that do not match the predicate and for the matching ones the Role dimensions remain unmodified. In case Roles are filtered, only the Roles of the corresponding Role Type are changed. This is ensured by keeping all Role of the Dynamic Tuple in DT_a except for Roles of the filtered Role Type t, which is specified by the first Role Type parameter $RT_a = RT' \setminus t$. The Role of Role Type t will come from the Dynamic Tuple in DT_{pred}. Thus, the Dynamic Tuple intersection parameter RT_b is set to $\{t\}$.

As example imagine the two input sets DT_{Person} and $DT_{University}$ that are populated as follows and an operator parametrized as $\sigma^{Person,\{Student\}}_{length(p.address) \geq (2 * length(u.location))}$. This predicate filters **Person** Dynamic Tuples that overlap with a **University** Dynamic Tuple by the Role Type **Student**, on the filter predicate that the **Person's** *address* is at least *2 times longer* than the *location* attribute of the **University**. Moreover, the address of $John$ is *123 Fake Street* and has a length of 15. The address of Max is *197 Foo Way* and features a length of 11. Finally, the TUD location is *Dresden*, which has a length of 7. Thus, multiple Dynamic Tuple input streams are required to evaluate the predicate, especially Dynamic Tuples of **Persons** and those of **Universities**.

$$\begin{aligned} DT_{Person} =& \{dt_{John} = (John, \{\{s_1, s_4\}\{sa_1\}\}, \emptyset), dt_{Max} = (Max, \{\{s_2\}\}, \emptyset)\} \\ DT_{University} =& \{dt_{TUD} = (TUD, \emptyset, \{\{s_1, s_2\}, \{sa_1\}\})\} \end{aligned}$$

At first, B is produced according to 4.60. Thus, each Dynamic Tuple of both input sets is flattened by using the rules defined in 4.58 and 4.59. This results in the following two sets.

$$\begin{aligned} \chi(DT_{Person}) =& \{(John, s_1, sa_1), (John, s_4, sa_1), (Max, s_2)\} \\ \chi(DT_{University}) =& \{(TUD, s_1, sa_1), (TUD, s_2, sa_1)\} \end{aligned}$$

These sets are used two build the overall Cartesian product that represents B.

$$\begin{aligned} B =& \{(John, s_1, sa_1, TUD, s_1, sa_1), (John, s_4, sa_1, TUD, s_1, sa_1) \\ & (Max, s_2, TUD, s_1, sa_1), (John, s_1, sa_1, TUD, s_2, sa_1) \\ & (John, s_4, sa_1, TUD, s_2, sa_1), (Max, s_2, TUD, s_2, sa_1)\} \end{aligned}$$

On this basis, the $overlap$ and $pred$ function are applied on each tuple b in B, resulting in B_{pred}, as defined in 4.61. At first, the address of $John$ is compared to the length of $TUD's$ address, which evaluates to *true*, because $15 \geq 2 * 7$. Moreover, the overlap on the Role Type **Student** evaluates to true, because s_1 is included twice and at different points in this tuple. Consequently, the first tuple is evaluated positively and marked as qualified by adding it to the set B_{pred}. The second tuple also evaluates to true in case of the predicate, but fails the overlap check. In detail, the Role s_4 is included once. This procedure continues for all tuples in B. Notably, the last tuple passes the overlap check, but fails the predicate check, because the address of Max is not long enough. Thus, only one tuple passed this detail check and is included in B_{pred}.

$$B_{pred} = \{(John, s_1, sa_1)\}$$

As it can be seen, all fields that do not belong to the input set DT_a are deleted from the tuple in this set. Next, the comparison Dynamic Tuples collected in DT_{pred} are constructed as specified in rule 4.63. As there is only one tuple in B_{pred}, DT_{pred} will also consists of only one Dynamic Tuple.

$$DT_{pred} = \{(John, \emptyset, \emptyset)\}$$

The filter type is specified as $Person$, hence the whole Dynamic Tuple is selected and the Roles in their corresponding dimensions are of no interest. Consequently, the Dynamic Tuples in DT_{pred} have empty dimensions. In contrast, a selection on the Role Type Student with the same Attribute would to the following set $DT_{pred}^{Student}$.

$$DT_{pred}^{Student} = \{(John, \{\{s_1\}\}, \emptyset)\}$$

As defined, only one input set will be manipulated, in this example it is DT_{Person}, which is finally intersected with DT_{text} by using the following operator.

$$\cap_{\cap}^{RT',\emptyset}(DT_{Person}, DT_{pred})$$

The output set DT'_{Person} consists of one Dynamic Tuple, too. As defined in 4.65, the dimensions are not modified at all, because the selection is specified on the core. Consequently, the Dynamic Tuple dt_{John} is passed to the output set and not manipulated.

$$DT'_{Person} = \{dt_{John} = (John, \{\{s_1, s_4\}\{sa_1\}\}, \emptyset)\}$$

In sum, this attribute selection operator enables the filtering of Dynamic Tuples as whole data structure or only parts of it, depending on the parameter t and for a given arbitrary predicate. Moreover, this evaluation process is performed with respect to an overlapping information that takes the Compartment-dependent structure into account. Depending on the predicate complexity, additional Dynamic Tuple streams are put into this operator to provide it the required data. However, only the first Dynamic Tuple input set is filtered, all others are used as auxiliary information provider.

4.5 SUMMARY

A proper database model that represents Roles, Compartments, and Relationships as first class citizen is the first step in the direction to role-based database system. In the beginning of the chapter nine requirements for such a database model were defined and discussed. This discussion builds the basis for the subsequent evaluation of related approaches, starting from the very first database model

featuring roles as concept, proposed by Bachman and Daya in the 1970s, over database models that use roles without context to extend an entity's structure during runtime, like DOOR and Fibonacci, to an approach like INM that intermingles roles with relationships and induces inheritance hierarchies. None of these approaches is able to satisfy the full list of requirements, though.

Hence, the evaluation is followed by a novel database model definition; the RSQL data model. It consists of Dynamic Data Types on the type level and Dynamic Tuples on the instance level. The former combines the notion of an entity type with its Role Types, but Role Type are required to be present in at least two Dynamic Data Types where one is represented by a Compartment Type. Furthermore, this definition allows to specify the structure an instance of this type may acquire during runtime, hence, it allows combining instances having different structures, but a common core, to be of the same overall type, especially of the same Dynamic Data Type. The latter combines an entity with its actually played and featured Roles in an integrated data structure. By allowing these Dynamic Tuple to change their Configuration dynamically during runtime, the system is able to represent entities that are able to change their structure within the boundaries set by the Dynamic Data Type, but without changing their overall type.

Next, several formal operators on the RSQL data model are defined to describe the possible operations on the data model. This is of importance especially for optimizations purposes. As defined, Dynamic Tuples represent the integrated data structure on the instance level, hence, the operators are tailored to process and manipulate such. However, there are three classes of operators available. The first class defines operators that operate on the structural level of Dynamic Tuples, like filtering two Dynamic Tuple sets on overlaps. The second class of operators enables usual set operations for Dynamic Tuples, for instance intersecting two sets of Dynamic Tuples while specifying the operation performed on the level of Roles. The last category features one operator only, which processes and manipulates Dynamic Tuples based on the values of attributes. In total, a list of ten operators is defined. To conclude this chapter, the database model definitions as well as the operators of this chapter are evaluated against the requirements defined in Section 4.1. This evaluation is presented in Table 4.4.

Requirement	Bachman	ORM	DOOR	Fibonacci	INM	RSQL DM
Notion of Roles	■	⊞	■	■	■	■
Notion of Compartments	□	□	□	□	□	■
Notion of Relationships	□	■	□	□	⊞	■
Roles in Compartments	□	□	□	□	□	■
Multiple Player Types	■	□	■	□	□	■
Multiple Roles simultaneously	□	■	□	■	■	■
Integrated Data Structure	■	□	■	■	■	■
Formal Operators	⊞	□	■	⊞	□	■
Mathematical Closure	⊞	□	■	■	□	■

■: yes, ⊞: partial, □: no

Table 4.4: Evaluation of Related Data Model Approaches and RSQL's Data Model

In sum, the RSQL data model satisfies all posed requirements. It features an explicit notion of Roles, Compartments, and Relationships, as well as it embeds each Role into a certain Compartment such that isolated Roles are prohibited. Moreover, a Role Type can have multiple player types and a single entity can start playing several Roles of the same Role Type simultaneously. Dynamic Data Types and Dynamic Tuples represent the integrated data structures, the former on the type level and the latter one on the instance level. Finally, the operators defined in Section 4.4 build the mathematical foundation for query processing and Dynamic Tuple manipulations. Additionally, these operators have a mathematical closure on the operational data model. In total, this chapter builds the mathematical base for a role-based database system as well as a role-based query language.

QUERY LANGUAGE AND PROCESSING

There exists a large variety of languages to communicate with a database system; each having a certain purpose and functionality goal. This ranges from conceptual query language to provide implementation independent functionality like SQL/EER [42], over languages that describe relational schema evolution, as CODEL does [38], to database programming languages like PL/SQL [69]. A query language in the narrow sense provides statements to retrieve data from a database only. In a broader sense, statements to define the database schema and to populate the database are associated to a query language as well. Hence, the functionality of an external database system interface is collected under the term query language. To provide a sophisticated database interface on the basis of RSQL's database model, this chapter introduces the RSQL query language including three language parts to define, manipulate and retrieve role-based data objects. Consequently, the term query language in its broader sense is assumed.

The main body of this chapter, especially the syntax description and result discussion, is based on the works presented in [50, 49, 51]. At first, we define requirements posed to a query language supporting the notion of Roles and Compartments. These requirements are mostly non-functional, because each query language is tailored to a certain database model and the database models are evaluated in Section 4.2. In fact, these requirements reflect goals for a clear definition and rigid connection to the underlying database model. Secondly, we discuss related database interface approaches and evaluate them to the previously defined requirements. Query languages supporting a notion roles as entity internal separation of concerns are rare, only two approaches exist. Next, RSQL's sublanguages to create a role-based database schema, to populate and manipulate the database, and to query role-based data are presented in the Sections 5.3, and 5.4, respectively. All parts feature a formal syntax description in Extended Backus–Naur form (EBNF) and several examples to demonstrate the connection between the language elements and the underlying database model and operators. Based on the syntax description of the query language, we discuss the query processing of Dynamic Tuples in detail. Finally, this chapter discusses the result representation of various Dynamic Tuple sets, a typical RSQL query result, and the options to navigate and iterate between these sets. A conclusion and evaluation of RSQL in comparison to the two related approaches completes this chapter.

5.1 REQUIREMENTS

A query language acts as external interface of a database system. Usually, it is designed to represent the concepts of the underlying database model. Consequently, SQL [45] is designed to query relational tables, and Cypher [82] for data stored graph-wise with vertexes and edges. In contrast to database model tailored query language, there exist conceptual query languages, like SQL/EER [42] for the extended entity relationship data model or CABLE [75] for the traditional entity relationship model. This type of languages implements the concepts of a conceptual model as first class citizen in a query language. Usually, these languages are available for users only and are not actually implemented as database query language. Thus, conceptual query languages are transferred into a query language that is executable in a traditional database system. By transferring the external interface characteristic of a query language to a role-based database system, various requirements can be identified. In total, five requirements are defined and listed in Table 5.1.

At first, query languages implemented in a DBS are tailored to the underlying logical database model. To provide role semantics and the entity internal **metatype distinction** on this interface as well, the query language has to feature this distinction, too. This metatype distinction requirement is represented in QL.1.

Secondly, a proper query language needs language statements to create schema objects and interrelations between these objects. These parts of the external database interface are referred to as **data**

Requirement	Description
QL.1	Metatype distinction on the basis of the data model
QL.2	Data definition language to create the schema
QL.3	Data manipulation language to populate the database
QL.4	Data query language to retrieve stored data
QL.5	Syntax description to connect the operators to the query language
QL.6	Adapted Result Representation

Table 5.1: Overview of Requirements Posed to Role-based Database Query Language

definition language. This language part has to represent the metatype distinction in combination with data model constraints, too. This might be a prohibition of Role Types to exist in isolation, or Relationship Types can be established between Role Types of the same Compartment Type only. In sum, these type level requirements are collected in requirement QL.2.

The instance level representation of requirements for a role-based query language are summarized in QL.3. The database interface is required to provide statement to create and manipulate instances, most likely in a **data manipulation language**. This includes the implementation of database model constraints, like avoiding isolated Roles, too. Generally, this requirement aims for statements to populate a role-based database.

To retrieve data from the database system, the actual **query language** has to be adopted to the underlying database model as well, which is stated in requirement QL.4. This is the most important part of the external database interface, because it is used by applications and users to retrieve information. In the best case, the metatype distinction is directly reflected in the statements.

Moreover, a **syntax description** is required as stated in the requirement QL.5. In the first place, this defines how a query is written and secondly which expressions are part of certain statements and where they can be applied. On the one hand side, this helps user to writer and debug their queries more efficiently by giving them a clear definition how to write a query. On the other hand, this builds the bridge between the expressions of the query language and the operators performed during query processing.

Finally, a proper result representation is required to preserve the database model semantics in the query results. This completes a fully integrated role-based database solution and is specified in the requirement QL.6. Moreover and depending on the result representation, novel navigation and browsing options have to be defined to provide the applications functionalities to process the result.

5.2 RELATED WORK

As aforementioned, there exist a lot of query languages, but only two among them support a notion of roles. These approaches are ConQuer and IQL, which are discussed in the following. This discussion outlines the design goals of the corresponding approach and details each language by utilizing a small example. Moreover, these approaches are evaluated with respect to their ability to represent a separation of concerns within a query.

5.2.1 ConQuer

The first query language to be analyzed is ConQuer, an Object Role Model (ORM) based conceptual query language. In 1996 the first version of ConQuer has been published in [13], followed by a second version in 1997 [14]. It is characterized as conceptual query language, because it allows users to formulate queries against a DBS on the basis of the conceptual model [33]. In the background this type of query languages maps to the logical database model. Thus, it provides a database interface to the users and applications, which is not based on the database model, but on the conceptual data model. Moreover, ConQuer can be categorized into the class of mapping engines as discussed in Section 3.4.2.

Notably, ConQuer is a query language in a narrower sense. It is a query language only and does not feature statements to create or manipulate data stored in a DBS. In fact, it assumes the actual data to be stored in a relational format. Generally, a ConQuer query describes facts and evaluates predicates for these facts. Roles are not distinguished explicitly, rather they are used as named places in relations between facts. This is caused by the ORM data model and is reflected directly in the query language.

```
Q2    ✓Academic
          +- is Professor
          ¦         +- holds Chair = Informatics
          +- not was awarded Degree in Year
                                +- is from University = UQ
```

Figure 5.1: ConQuer Example Query; Extracted From [14]

In Figure 5.1 an example ConQuer query is presented. It queries for an **Academic** fact that is related to a **Professor** fact over an **isa** relationship. Additionally, this **Professor** fact has to be related to **Chair** fact by a **holds** relationship. Moreover, the **Academic** itself must not hold a degree from a certain **University** UQ.

However, there is no (formal) syntax description available for ConQuer. In each of their published works on ConQuer, they describe the language using example queries and possible mappings [13, 14, 33]. In [13] the authors present a screenshot of a ConQuer implementation in the *InfoAssistant* product of the company *Asymetrix*. Today, neither the product nor the company does exist anymore. However, the implementation utilizes an exploratory approach, which presents the query writer various options on how to complete the query. These options are extracted from the ORM-based schema descriptions as shown in [13, Figure 1].

Unfortunately, ConQuer is a conceptual query language only. Based on the background mapping, the database system cannot utilize the ORM specific semantics for optimization purposes. Hence, there are no special operators that process ORM data objects, rather relational operators are used. Moreover, the authors do not explain the storage of their metadata, especially how they represent the interrelations between the facts and how they generate the queries. In detail, they claim for more stable ConQuer queries, which is motivated by abstraction to ORM data model. However, the SQL query generator is required to have knowledge about the actual mapping and where it will find the metadata in the relational schema. Especially normalized relational schemata distribute the data object's attributes differently from ORM schemata. Furthermore, a discussion on the result representation is totally missing. It is assumed that ConQuer provides a relational query result based on the background mapping.

In sum, ConQuer provides the users an ORM abstracted view on data in a relational database. It can be seen as additional interface along with a relational SQL interface. It features, caused by the

underlying data model, weak role semantics, but is a good example to show the benefits of query languages located on the conceptual layer. ConQuer fulfills the requirement QL.4 only, because there is the aforementioned weak notion of roles, and it does not come with a separate language to create schema objects as well as manipulate the data in an ORM-based way. Moreover, a syntax description and actual implementation is not available.

5.2.2 Information Networking Model Query Language

The information networking model approach provides a query language as well. In detail, the authors describe a language that features a definition language (IDL), a manipulation language (IML), and a query language (IQL) [43]. All language parts are based on and tailor to the information networking model. Hence, it can be seen as full external database interface for an information networking model database system.

The IQL is an X-Path [12] inspired query language, thus, the expressions within a query form a tree [43]. This is reasonable, because their underlying data structure forms a tree as well (see Section 4.2.5). In general, the IQL is separated into two parts, a query and a construction part [43]. Typically, both parts are handled within one statement, for instance in the projection clause of a SQL select statement. The query part describes bindings of objects in the database to variables used in the query. In contrast, the construction part defines the tree-based representation of the bindings for the result. Figure 5.2 illustrates both, the query and construction part, of an IQL statement.

```
1 query TUD//Prof:$x//supervises:$y[//student\_id:$z,//supervisor:$u,age:$v]
2 construct Prof:$x[supervises:$y[student\_id:$z,supervisor:$u,age:$v], avgAge:avg($v)]
```

Figure 5.2: Example IQL Query Statement Illustrating the Query and Construction Part; According to [43, p. 529]

This example queries for a **Professor** x at *TUD* who *supervises* a set of students y. Moreover, the *student_id* is bound to the variable z, the supervisor to u and the age of the student to v. In addition, the construction part defines the result representation omitting the root *TUD*, rather each the **Professor** bound to x forms a separate root, indicated by []. Next, each **Professor** root has a set of *supervised* students, as specified in the query part. Additionally, this query constructs an additional variable for the *average age* of the supervised students.

As it can be seen in the example, a metatype distinction is not present, which is consequent with respect to the underlying database model. In INM role relationships are represented as traditional classes. Moreover, actual relationships, like *supervises*, are handled as class or objects as well. As a consequence, it does not become clear within a query, which variable refers to a class, which to role relationship, and which to a normal relationship. Basically, each metatype can occur in each part of an expression. Consequently, a metatype distinction is absent in the construction part as well.

The IDL and IML are only mentioned and not explained in detail. Hence, it remains unclear which metatypes they actually distinguish. In case of the definition language IDL, the definition statements could be similar to the class definitions of the database model, as explained in Section 4.2.5. For the IML, neither functionality specifications nor a syntax description is given. However, the IQL has a formal syntax description for both, the query and construction part [77, Appendix A].

Like the IQL's data definition and manipulation language, the result representation is not discussed very well. The INM data structure forms a tree and so the query language implements tree structures

as query concepts. It is assumed that the result is a tree as well, consisting of classes and their interrelations. However, the result returned from an IQL query would hold classes instantiated on the database side, in this situation. In which way the result can be processes by applications and which options to navigate between the classes are available, is not mentioned at all.

In sum, the IQL is a powerful query language for the tree-based INM data structure. The separation of a query description and result construction part is a nice feature and results in a well-structured and organized query statement. However, a metatype distinction is absent, rather each metatype is equally handled as class or object on the instance side. It fully satisfies the requirements QL.4, and QL.5, even if there are no operators, but they relate their query language with the defined search strategies. QL.1 and QL.2 are partially fulfilled, because the data model provides a metatype distinction that is weakly transformed into the query language and it is assumed that the definition language is specified as the class specifications given in [62, 43]. The requirement QL.3 and QL.6 are not fulfilled at all, because they claim to have defined an IML, but a syntax description is completely missing and assumption based on their class definitions cannot be made. Moreover, the result representation is unknown and an implemented INM database system is not available.

5.2.3 Discussion

Query languages that take advantage of roles or support a notion of flexible entities are rare. In fact, there are two approaches only, **ConQuer** [14, 13] on the basis of Object Role Model [34] and **IQL** [43, 77] for the Information Networking Model [61, 44]. However, evaluating and rating competing or at least related query languages is unfair. At first, these languages are based on and tailored to a certain database model that has been designed to satisfy possibly different criteria than the approach discussed in this thesis. Secondly, the query language is second step to take, right after defining the database model. Weaknesses in the database model cannot be compensated by the query language. If there is no discrimination between the desired metatypes and no explicit notion of a context in the database model, like it is in the case of INM, the query language is not going to support this type of distinction. For these reasons, the requirements are as general as possible. However, the evaluation presented in Table 5.2 does not rate the general language design, rather the functionality aspects are evaluated.

Requirement	ConQuer	IQL
Metatype distinction	☐	⊞
Data definition language	☐	⊞
Data manipulation language	☐	☐
Data query language	■	■
Syntax description	☐	■
Result representation	☐	☐

■: yes, ⊞: partial, ☐: no

Table 5.2: Evaluation of Related Database Interface Approaches

At first, none of the approaches provides an adequate data manipulation language. In case of ConQuer this is by design, because they put an additional ConQuer layer over the traditional SQL layer to simplify querying. This does not hold for the INM manipulation language. Basically, they do not provide a syntax or any language example. Additionally, it does not become clear in related published works how such a manipulation language could look like. For instance, how it is stated to which entity a role

relationship has to be added to or which constraints apply to an insert or update statement. Furthermore, the result representation of a query is not discussed at all by these approaches. Consequently, the options to process the result remain unclear.

Secondly, a data definition language is not discussed very well. As aforementioned, ConQuer is a query language in the narrow sense, hence, it is applied upon another language. Consequently, all schema definitions and data manipulations are not covered by this approach, at all. In case of INM, the data definition language could look like the class definitions in their discussed example [63, Example 1, p. 299]. However, they do not explicitly state that these statements belong to their data definition language.

Thirdly, the metatype distinction is not satisfied. The IQL is based on the INM, which actually provides a metatype distinction between classes and role relationships, but this is not consequently transferred to the query language. In case of ConQuer, the data model has a weak metatype distinction, so the corresponding query language inherits this weakness.

Fourthly, a syntax description with respect to the query description underneath, is provided by the IQL only. They do not define formal operators, but four search strategies which are coupled with their query language. For the query and construction part, a syntax in Backus–Naur form is available [77, Appendix A]. Unfortunately, ConQuer is explained by example only and does not feature an available syntax description.

Finally, both approaches provide a proper query language for their desired purposes, but on a weak data model, with respect to an explicit Role and Compartment notion. In total, none of the discussed approach provides a satisfying foundation for the requirements posed to a role-based and context-dependent query language.

5.3 RSQL DATA DEFINITION AND MANIPULATION LANGUAGE

Database systems manage their data and check their integrity with respect to a given schema. A schema definition depends on the database model's data structures. Nowadays, these structures are very diverse. Relational database systems manage their data by means of tables [19], a key-value store simply associates an arbitrary value to a certain key [66], and graph database relate vertexes by edged [5]. A database system based upon RSQL's database model manages its data structures by means of Dynamic Data Types (see Section 4.3.1). At first, we explain RSQL's data definition language syntax, to create such schema objects, and demonstrate it by employing several statements. To populate a database and manipulate the actual data, RSQL's data manipulation language is introduced and discussed. According to the database model (see Section 4.3.2), these manipulation procedures are applied on Dynamic Tuples, the instance representation of Dynamic Data Types and the main instance data structure. At first, we explain discuss the syntax of this language in detail. Based on this, we create a small role-based database by using RSQL's data manipulation language, This demonstrates the relation between the statements, the database model, and the resulting data structures.

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

⟨*create-ct*⟩ ::= `CREATE COMPARTMENTTYPE` ⟨*ct-name*⟩ `(` ⟨*attribute-definition*⟩ `(` `,` ⟨*attribute-definition*⟩ `)*` `)`

⟨*create-rt*⟩ ::= `CREATE ROLETYPE` ⟨*rt-name*⟩ `(` ⟨*attribute-definition*⟩ `(` `,` ⟨*attribute-definition*⟩ `)*` `)`
`PLAYED BY (` ⟨*rigid-name*⟩ `(` `,` ⟨*rigid-name*⟩ `)*` `) PART OF` ⟨*ct-name*⟩

⟨*create-rst*⟩ ::= `CREATE RELATIONSHIPTYPE` ⟨*rst-name*⟩
`CONSISTING OF` ⟨*rst-participating-expression*⟩ `AND` ⟨*rst-participating-expression*⟩

⟨*rst-participating-expression*⟩ ::= `(`⟨*rt-name*⟩ `BEING (` `0` | `1` `) . . (` `1` | `*` `)`

⟨*attribute-definition*⟩ ::= ⟨*attribute-name*⟩ ⟨*data-type*⟩ (⟨*constraint*⟩)*

⟨*constraint*⟩ ::= `PRIMARY KEY` | `NOT NULL` | `UNIQUE`

⟨*drop-nt*⟩ ::= `DROP NATURALTYPE` ⟨*nt-name*⟩

⟨*drop-ct*⟩ ::= `DROP COMPARTMENTTYPE` ⟨*ct-name*⟩

⟨*drop-rt*⟩ ::= `DROP ROLETYPE` ⟨*rt-name*⟩

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

Figure 5.3: RSQL's Data Definition Statements (excerpt)

5.3.1 Data Definition Language Syntax

Creating an RSQL schema focuses on defining the building blocks of Dynamic Data Types, for instance, Natural Types and Role Types. Thus, the core elements, rather than the Dynamic Data Type as logical data structure, are the focus of the data definition language statements. The syntax of this language is represented in Figure 5.3. However, only the statements to create and drop schema objects are presented. Other statements, for example to rename attributes or change Relationship Type participants, are omitted, because they focus on evolving a database schema rather than creating one and this schema evolution is out of scope.

The schema definition in Section 4.3.1 specifies four different metatypes to achieve an entity internal separation of concerns. Consequently, the data definition language features four different statements to create these schema objects, the ⟨*create-nt*⟩, ⟨*create-ct*⟩, ⟨*create-rt*⟩, and ⟨*create-rst*⟩ statement. As the names indicate, the first one creates a new Natural Type in the schema, the second one a new Compartment Type, the third a new Role Type and the last one a new Relationship Type. From a database model perspective, the specified sets, relations, and functions are populated by these statements. Each of these statements starts with an initial `CREATE` followed by the type reference.

Creates

Afterwards, the syntax of the statements differs. In case of ⟨*create-nt*⟩ and ⟨*create-ct*⟩ the attribute definition follows the initial phrases. An ⟨*attribute-definition*⟩ consists of an ⟨*attribute-name*⟩ and a data type specification like `INT` or `varchar`. Moreover, ⟨*constraints*⟩ like a primary key, not null, or uniqueness can be optionally defined for each attribute. Generally, all data types implemented in a

traditional relational database system are available. Hence, both statements only differ in the referenced metatype.

In contrast to these two statements for specifying a rigid type, the ⟨*create-rt*⟩ statements additionally defines its player types and the respective Compartment Type. After the attribute definition of a Role Type, the phrase `PLAYED BY` indicates a list of rigid types that are able to fill this Role Type. As both, Natural Types and Compartment Types, are able to fill Role Types, both can be referenced in this clause. The list of these types is pointed off. Additionally, this phrase populates the schema's *fills* relation, as defined in Section 4.3.1. Moreover, each Role Type has to be part of exactly one Compartment Type. To specify this, the `PART OF` phrase references a Compartment Type in which the Role Type is contained in. This information is stored in the *parts* function of schema.

To create a proper Relationship Type, the name and the participating Role Types including their cardinality constraints is required. A participating Role Type is specified by a ⟨*rst-participating-expression*⟩. Since there are always two Role Types involved in a Relationship Type, two of these expressions are required in total. A Compartment Type specification is not required, because both Role types have to be in the same Compartment Type anyway, thus, it can be derived implicitly. However, such an expression references a Role Type by naming it by using a ⟨*rt-name*⟩ followed by the phrase `BEING` and the cardinality constraints. Furthermore, the information which Role Type is relationship to which other one by which Relationship Type is stored in the *rel* function of the database model's schema. These constraints include a lower and an upper bound. For simplicity the lower bound is limited to an `0` or `1` and the upper to a `1` or `*`. However, any combination of numbers in this cardinality specification is possible in general, but for explaining the syntax, these options suffice. The actual cardinality information for a Relationship Type is stored in the function *card*.

Drops

The drop statements for each of the metatypes is straight forward. A `DROP` in conjunction with the metatype name and the corresponding type name specifies a proper drop statement. Depending on which metatype is referenced the effects on the schema vary.

Natural Type Dropping a Natural Type results in deleting it from the set NT and a drop of all its references from the *fills* relation.

Compartment Type In case a Compartment Type is dropped, the corresponding element in the set CT is deleted. Additionally, all references in the *fills* relation are discarded and the *parts* function is adapted.

Role Type Dropping a Role Type results in deleting the element from the set RT, deleting all references in the *fills* relation, adapting the *parts* function for the Compartment Type the respective Role Type is contained in, and adjusting the *rel* function for all affected Relationship Types.

Relationship Type If a Relationship Type is dropped from the system, the corresponding element in the set RST is deleted, the *rel* function for this Relationship Type is deleted, and consequently, the *card* function is adapted by deleting all cardinality constraints associated to this Relationship Type.

However, dropping types may result in cascading drop sequences. For instance, dropping a Compartment Type has to result in the drop of all contained Role Types as well. Generally, there are two option to implement this: (i) dropping all related types as well or (ii) only allow to drop unrelated types, such as empty Compartment Types. The first option may result in cascading drop sequence and unintended drops, but allows for dropping many types with only few statements. In contrast, the second option requires an order within the drop statements. At first Relationship Types have to be dropped, afterwards the Role Types and finally the Compartment and Natural Types. Moreover, this option requires an individual statement for dropping a type and denies invalid drop statements. This gives the developer the most control over dropping types, but demands more effort. No matter which of these options is implemented, the syntax is orthogonal to this problem, which exist in relational database systems as well. For instance, in case there are foreign key constraints between two tables and the referenced table should be dropped.

In sum, the data definition language syntax discriminates between the four metatypes defined in the database model and manipulates the database model's relations and functions as well.

5.3.2 Creating And Extending Dynamic Data Types

To demonstrate the creation and extension of a Dynamic Data Type and the underlying schema elements, a small example based on the conceptual model illustrated in Figure 2.3 on page 21, is employed. This example shows the creation of two Dynamic Data Types, which share certain Role Types and a Relationship Type, in five statements. Moreover, the population of the sets, relations and functions is given for each of these steps. In particular, a conceptual representation of the model state after each step is depicted in Figure 5.4. In the end, we model the situation depicted in step five; a **Person** who can be **Student** and **Professor** within a **University** and a certain **Professor supervises** some **Students**.

Next, each statement is discussed in detail by representing the actual textual statement, the population of the database model's schema sets, its functions, and relations. Finally, the resulting Dynamic Data Types are explained.

Statement 1 At first, the Natural Type **Person** is created.

```
1 CREATE NATURALTYPE Person (name varChar(128) PRIMARY KEY,
2   lastname varChar(128) PRIMARY KEY, birthday date PRIMARY KEY,
3   address varChar(256));
```

The schema $\mathcal{U}$ after executing this statement has the following state.

$$\begin{aligned}\mathcal{U} =&(NT = \{Person\}, CT = \emptyset, RT = \emptyset, RST = \emptyset,\\ &fills = \emptyset, parts = \emptyset, rel = \emptyset, card = \emptyset)\end{aligned}$$

Moreover, the following Dynamic Data Type is generated, because a new Natural Type is created. Creating a new Natural Type in the system always results in a new Dynamic Data Type.

$$ddt_{Person} = (Person, \emptyset, \emptyset)$$

Statement 2 Secondly, the Compartment Type **University** is created in the system.

```
1 CREATE COMPARTMENTTYPE University (name varcChar(128), address varChar(255),
2   location varChar(255));
```

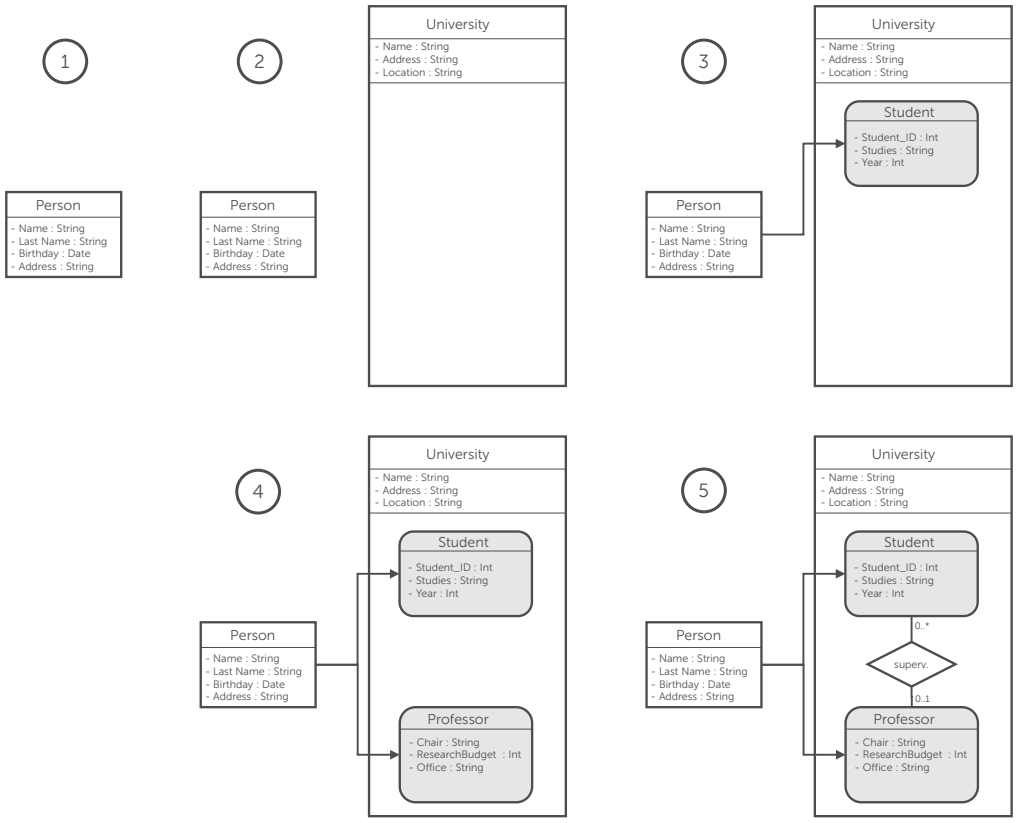


Figure 5.4: Model Evolution While Creating a Role-based Database Schema

The schema $\mathcal{U}$ after executing this statement has the following population. As it can be seen, the **University** is added to the set CT and to the *parts* function, but with an empty set.

$$\mathcal{U} = (NT = \{Person\}, CT = \{University\}, RT = \emptyset, RST = \emptyset,$$
$$fills = \emptyset, parts = \{University \rightarrow \emptyset\}, rel = \emptyset, card = \emptyset)$$

Moreover, the creation of a Compartment Type always creates a new Dynamic Data Type.

$$ddt_{Person} = (Person, \emptyset, \emptyset)$$
$$ddt_{University} = (University, \emptyset, \emptyset)$$

Statement 3 As there is a Natural Type and a Compartment Type in the schema, Role Types can be added. The first Role Type to be created in the schema is **Student**.

```
CREATE ROLETYPE Student (Student_ID int, Studies varChar(255), year int)
  PLAYED BY (Person) PART OF University;
```

The schema $\mathcal{U}$ after executing this statement has the following elements.

$$\mathcal{U} = (NT = \{Person\}, CT = \{University\}, RT = \{Student\}, RST = \emptyset,$$
$$fills = \{(Person, Student)\}, parts = \{University \rightarrow \{Student\}\},$$
$$rel = \emptyset, card = \emptyset)$$

Inserting a new Role type into the schema always results in extending at least two Dynamic Data Types. All referenced rigid types in the `PLAYED BY` phrase are extended in the filling dimension.

Additionally, the Compartment Type in the `PART OF` phrase is enhanced in the participating dimension. As it can be seen, the **Student** Role Type has been added to the Dynamic Data Type **Person** as well as **University**.

$$ddt_{Person} = (Person, \{Student\}, \emptyset)$$
$$ddt_{University} = (University, \emptyset, \{Student\})$$

Statement 4 Next, the Role Type **Professor** is inserted into the schema.

```
CREATE ROLETYPE Professor (Chair varChar(256), ResearchBudget int,
  Office varChar(128)) PLAYED BY (Person) PART OF University;
```

The schema $\mathcal{U}$ after executing this statement has the following elements.

$$\begin{aligned}\mathcal{U} =&(NT = \{Person\}, CT = \{University\}, RT = \{Student, Professor\}, RST = \emptyset,\\ &fills = \{(Person, Student), (Person, Professor)\},\\ &parts = \{University \rightarrow \{Student, Professor\}\}, rel = \emptyset, card = \emptyset)\end{aligned}$$

By creating this Role Type, the corresponding sets, relations, and functions are updated. The Dynamic Data Types are adapted to the new Role Type, too.

$$ddt_{Person} = (Person, \{Student, Professor\}, \emptyset)$$
$$ddt_{University} = (University, \emptyset, \{Student, Professor\})$$

Statement 5 Finally, the Relationship Type **supervises** between the **Professor** and **Student** Role Types is created.

```
CREATE RELATIONSHIPTYPE supervises CONSISTING OF
  (Professor BEING 0 .. *) AND (Student BEING 0 .. 1);
```

The schema $\mathcal{U}$ after executing this statement has the following elements.

$$\begin{aligned}\mathcal{U} =&(NT = \{Person\}, CT = \{University\}, RT = \{Student, Professor\},\\ &RST = \{supervises\}, fills = \{(Person, Student), (Person, Professor)\},\\ &parts = \{University \rightarrow \{Student, Professor\}\},\\ &rel = \{supervises \rightarrow (Professor, Student)\}, card = \{supervises \rightarrow (0..\infty, 0..1)\})\end{aligned}$$

The creation of this Relationship type results in the population of the set RST and both functions, rel and $card$. However, the Dynamic Data Types are not affected at all, because they describe the structure of an entity only and not the relations between entities.

$$ddt_{Person} = (Person, \{Student, Professor\}, \emptyset)$$
$$ddt_{University} = (University, \emptyset, \{Student, Professor\})$$

These statements create and extend Dynamic Data Types and populate the schema. In contrast, all drop statements reduce the schema and shrink Dynamic Data Types.

5.3.3 Data Manipulation Language Syntax

As the schema is created by the base elements and inferring Dynamic Data Types, the same approach is applied on the instance level as well. Hence, the data manipulation language creates, updates, and deletes Naturals, Compartments, and Roles, according to the metatype distinction defined in the database model. The relationships are stored in the *links* function and do not feature an individual instance set. Moreover, the *plays* relation is the instance level representation of *fills* in combination with *parts* and connects the cores with played Roles in Compartments. In addition, the *type* function stores data about the type information of each instance. The data manipulation language is presented in Figure 5.5.

Inserts

Inserting cores into the system is as easy as creating them, because they do not have any constraints. In general, a core can be Natural or Compartment and the insert statements ⟨*insert-nt*⟩ and ⟨*insert-ct*⟩ start with `INSERT INTO NATURALTYPE` and `INSERT INTO COMPARTMENTTYPE`, respectively. This is followed by the respective name and a list of attributes for an explicit linkage between the attributes and values. Afterwards, the literal `VALUES` initializes the value specifications by several ⟨*value-expressions*⟩. However, an insert on a Natural Type and Compartment Type populates the corresponding sets and the *type* function. Moreover, for each of these statements a new Dynamic Tuple is created in the system.

Creating a new Role demands more effort in writing the statement, because the relations to other elements have to be denoted and established. The ⟨*insert-rt*⟩ statement starts with the phrase `INSERT ↪ INTO ROLETYPE` followed by a Role Type name. This specifies the Role Type of the new Roles. Next, the explicit assignment of attributes and values is specified by using the same mechanism as the statements to insert a new core. So far, this statement do not differ from statements inserting a core. In addition to this base description, the ⟨*insert-rt*⟩ statement defines by which core it is played and in which Compartment it is going to be featured in. The former is declared by the phrase `PLAYED BY` followed by a ⟨*config-expression*⟩ and an optional ⟨*where-clause*⟩.

The idea behind a ⟨*config-expression*⟩ is to describe the schema a Dynamic Tuple has to meet. Hence, it describes Configurations of a Dynamic Data Type. According to the database model and the Dynamic Data Type definition, it consists of a core with a mandatory alias and Role Type descriptions in two dimensions, the playing and featuring dimension. A dimension is indicated by its corresponding term `FEATURING` and `PLAYING`, respectively. To specify the Role Types in the dimensions and their logical relation to each other, the ⟨*log-expression*⟩ is used, consisting of a single Role Type definition or a Role Type definition in logical combination with another ⟨*log-expression*⟩. Consequently, a ⟨*log-expression*⟩ is set up recursively, having the recursion breakpoint at a single Role Type definition. However, a Role has to have exactly one player, hence, this ⟨*config-expression*⟩ has to specify exactly one Dynamic Tuple. The same applies for the featuring Dynamic Tuple, which is specified by the phrase `FEATURED BY` in combination with another ⟨*config-expression*⟩ and an optional ⟨*where-clause*⟩. Again, the second ⟨*config-expression*⟩ has to reference one Dynamic Tuple only, because a Role is featured in exactly one Compartment. In sum, this statement adds new entries to the set of Roles R, extends the *type* function, and extends two Dynamic Tuples.

The ⟨*insert-rst*⟩ has a different syntax again. It is designed to specify three Dynamic Tuples, one holding the Role of the first Role Type, one for the second Role Type participating in this Relationship Type, and finally the last one defines the Compartment these Roles will be connected in. At first the `INSERT INTO RELATIONSHIPTYPE` phrase in combination with a proper name initializes a ⟨*insert-rst*⟩

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

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

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

⟨*insert-rst*⟩ ::= INSERT INTO RELATIONSHIPTYPE ⟨*rst-name*⟩
CONNECT ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?
WITH ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?
AT ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?

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

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

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

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

⟨*delete-ct*⟩ ::= DELETE COMPARTMENTTYPE FROM (⟨*config-expression*⟩) (WHERE ⟨*where-clause*⟩)?

⟨*delete-rt*⟩ ::= DELETE ROLETYPE ⟨*rtname*⟩
PLAYED BY ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?
FEATURED BY ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?

⟨*delete-rst*⟩ ::= DELETE RELATIONSHIPTYPE ⟨*rstname*⟩
DISCONNECT ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?
FROM ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?
AT ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?

⟨*config-expression*⟩ ::= ⟨*rigid-name*⟩ ⟨*rigidAlias*⟩ (FEATURING ⟨*logical-expression*⟩)?
(PLAYING ⟨*logical-expression*⟩)?

⟨*logical-expression*⟩ ::= ⟨*rt-def*⟩ | ⟨*rt-def*⟩ ⟨*junctor*⟩ ⟨*logical-expression*⟩

⟨*rt-def*⟩ ::= ⟨*rt-name*⟩ ⟨*rtAlias*⟩

⟨*assignment-expression*⟩ ::= ⟨*attribute*⟩ = ⟨*newValue*⟩

Figure 5.5: RSQL's Data Manipulation Statements

statement. By denoting the Relationship Type name, the participating Role Types and the Compartment Type these Role Types are located in, is uniquely identifiable. Next, three Dynamic Tuples have to be described, which is performed by several ⟨*config-expressions*⟩. The first one is introduced by a CONNECT, the second by a WITH and the last one by an IN. All of these expressions can be combined with an individual ⟨*where-clause*⟩ to filter Dynamic Tuples on the attribute level. From a database mode perspective, this statement adds entries in the *links* function.

Updates

The update statements ⟨*update-nt*⟩, ⟨*update-ct*⟩, and ⟨*update-rt*⟩ are used to change value of certain attributes. There is no particular ⟨*update-rst*⟩ statement, because Relationship Types do not have attributes. In detail, the former two statements differ only in the referenced metatype. They start with `UPDATE NATURALTYPE` and `UPDATE COMPARTMENTTYPE`, respectively, followed by a proper name. Next, the phrase `SET` introduces the ⟨*assignment-expression*⟩, which references an ⟨*attribute*⟩ and a new value. Optional, the affected Dynamic Tuples can be filtered by an attribute filter in the ⟨*where-clause*⟩.

The ⟨*update-rt*⟩ statement differs from these by having two additional ⟨*config-expressions*⟩. One is initialized by `PLAYED BY` and the other by `FEATURED BY`, to specify these Dynamic Tuples, in which the affected Roles have to be played and featured by, respectively. These two expressions take the duality of a Role into account, by specifying corresponding player and Compartment. However, this update statements starts with `UPDATE ROLETYPE`, a Role Type name followed by a `SET` and an ⟨*assigment-expression*⟩. This expression references an attribute of the Role Type and sets the new value for all Roles of the updated Role Type. However, updates do not affect the sets and functions of the instance data model at all, because changes of attribute values are described only.

Deletes

To discard data from the database, four different delete statements are available. One for each metatype. The ⟨*delete-nt*⟩ and ⟨*delete-ct*⟩ statement specify Dynamic Tuples that have to be deleted from the database. The start with the phrase `DELETE NATURALTYPE FROM` and `DELETE COMPARTMENTTYPE FROM`. Afterwards, a ⟨*config-expression*⟩ specifies the schema of the affected Dynamic Tuples and the optional ⟨*where-clause*⟩ defines an attribute filter. A dedicated name for the Natural Type or Compartment Type is not required, because the ⟨*config-expression*⟩ specifies only one player type. Hence, there will not be any ambiguity about the core. However, each Dynamic Tuple that is qualified by the ⟨*config-expression*⟩ and the attribute filter will be erased from the database. In contrast, the ⟨*delete-rt*⟩ statement discards only parts of a Dynamic Tuple, especially the Roles of the specified Role Type. This statement starts with a `DELETE ROLETYPE` and a certain Role Type name. Note, three is no `FROM`, because Role are part of two instead of one instances, i.e. Dynamic Tuples. Consequently, the phrases `PLAYED BY` and `FEATURED BY` in conjunction with a ⟨*config-expression*⟩ and a ⟨*where-clause*⟩ specify the Dynamic Tuples, a qualified Role is played in and featured by, respectively. The qualifying Roles will be deleted from both Dynamic Tuples. This can be only one Role, or multiple Roles of the specified Role Type. Finally, the ⟨*delete-rst*⟩ statement disconnects Roles from a certain Relationship Type. The statement specifies three sets of Dynamic Tuples, like the contrary ⟨*insert-rst*⟩ does. It starts with the phrase `DELETE RELATIONSHIPTYPE` followed by a particular Relationship Type name ⟨*rst-name*⟩. Next, the three ⟨*config-expressions*⟩ in combination with an optional ⟨*where-clause*⟩ define the three sets of Dynamic Tuples. The first one defines the first Role Type of the Relationship Type and second ⟨*config-expression*⟩ the second Role Type. Finally, the third ⟨*config-expression*⟩ defines the Compartments the Relationships have to take place. However, the Dynamic Tuple specifications are introduced with `DISCONNECT`, `FROM`, and `AT`, respectively. In total, these statements are the inverses of the insert statements, hence, the same sets and functions are affected, but in an inverse way. Consequently, elements are erased from the instances' sets and functions are shrunk.

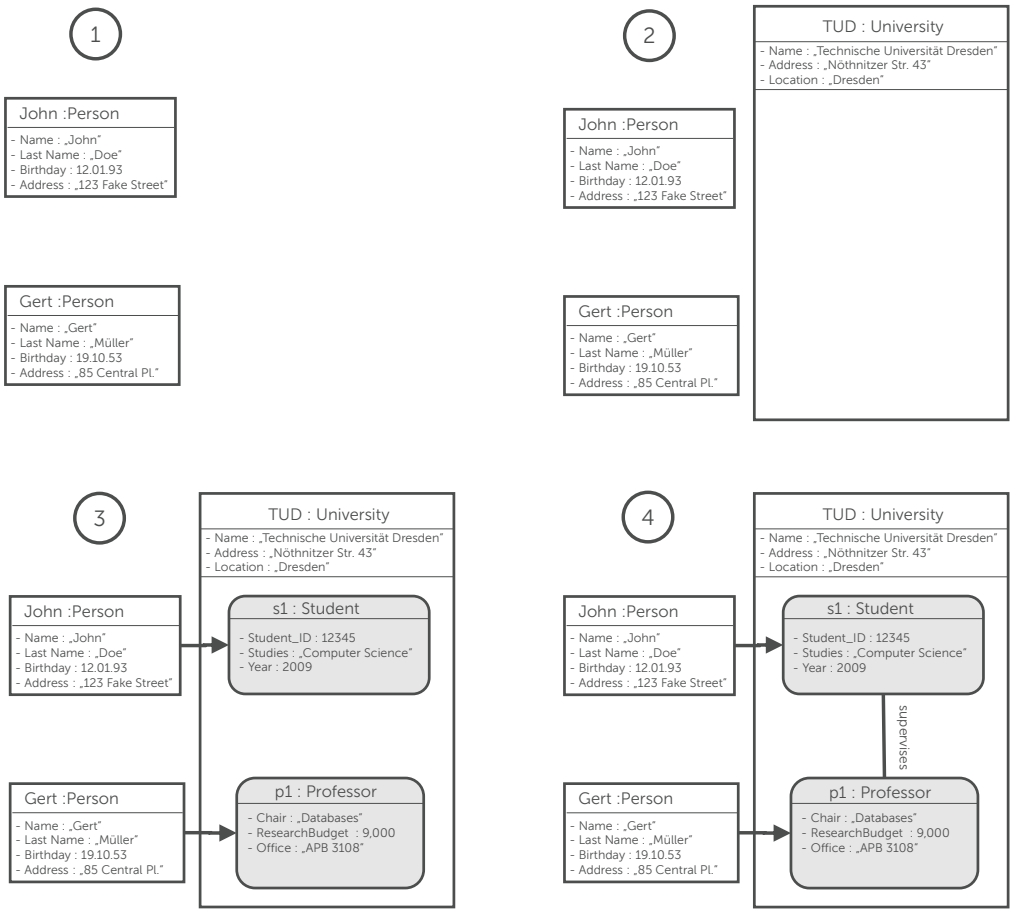


Figure 5.6: Dynamic Tuple Evolution While Creating a Role-based Database

5.3.4 Creating and Extending Dynamic Tuples

To explain the creation and extension of Dynamic Tuples in detail, a small example based on the instance presented in Figure 2.4 is employed. This example creates three Dynamic Tuples, add Roles to these, and connects them by a certain Relationship. In particular, four steps are required to build a database as conceptually shown in Figure 5.6. Each of these steps is discussed in detail by presenting the query, the effects on the data structure, and the database's set population.

Statement 1 At first, two Naturals are inserted in the database by executing the following two statements.

```
INSERT INTO NATURALTYPE Person (name, lastname, birthday, address)
  VALUES ("John", "Doe", 12.01.93, "123 Fake Street");
INSERT INTO NATURALTYPE Person (name, lastname, birthday, address)
  VALUES ("Gert", "Mueller", 19.10.53, "85 Central Pl.");
```

The instance $\mathfrak{u}$ of $\mathcal{U}$ after executing these two statements has the following state.

$$\begin{aligned}\mathfrak{u} =&(N = \{John, Gert\}, C = \emptyset, R = \emptyset,\\ &type = \{(John \rightarrow Person), (Gert \rightarrow Person)\}, plays = \emptyset, links = \emptyset)\end{aligned}$$

Moreover, the following Dynamic Tuples are generated, because two new Naturals are created. Creating a new Natural in the database always results in a new Dynamic Tuple.

$$dt_{John} = (John, \emptyset, \emptyset), dt_{Gert} = (Gert, \emptyset, \emptyset)$$

Statement 2 Secondly, the Compartment TUD is created, which builds the context for later Roles.

```
INSERT INTO COMPARTMENTTYPE University (name, address, location)
  VALUES ("TUD", "Noethnitzer Str. 43", "Dresden");
```

This affects the instance $\mathfrak{u}$ sets by adding a new Compartment to the Compartment set C and creating a new Dynamic Tuple.

$$\begin{aligned}\mathfrak{u} =&(N = \{John, Gert\}, C = \{TUD\}, R = \emptyset,\\ &type = \{(John \rightarrow Person), (Gert \rightarrow Person), (TUD \rightarrow University)\},\\ &plays = \emptyset, links = \emptyset)\end{aligned}$$

As inserting Naturals creates a new Dynamic Tuple, inserting a new Compartment has the same effect. Consequently, the database holds three Dynamic Tuples after executing the above statement.

$$dt_{John} = (John, \emptyset, \emptyset), dt_{Gert} = (Gert, \emptyset, \emptyset), dt_{TUD} = (TUD, \emptyset, \emptyset)$$

Statement 3 Next, Roles are created and bound to their respective player as well as to a Compartment. In detail, two Roles are inserted, a **Student** Role and a **Professor** Role by the following statements.

```
INSERT INTO ROLETYPE Student (Student_ID, Studies, year)
  VALUES (12345, "Computer Science", 2013)
  PLAYED BY PERSON p WHERE p WITH p.name="John"
  FEATURED BY University u WHERE u WITH u.name="TUD";
INSERT INTO ROLETYPE Professor (Chair, ResearchBudget, Office)
  VALUES ("Databases", 9000, "APB 3108")
  PLAYED BY PERSON p WHERE p WITH p.name="Gert"
  FEATURED BY University u WHERE u WITH u.name="TUD";
```

The base data are manipulated by adding both Roles to the set of Roles, adding the corresponding *types* function, and inserting the connection into the *plays* relation.

$$\begin{aligned}\mathfrak{u} =&(N = \{John, Gert\}, C = \{TUD\}, R = \{s_1, p_1\},\\ &type = \{(John \rightarrow Person), (Gert \rightarrow Person), (TUD \rightarrow University),\\ &(s_1 \rightarrow Student), (p_1 \rightarrow Professor)\},\\ &plays = \{(John, TUD, s_1), (Gert, TUD, p_1)\}, links = \emptyset)\end{aligned}$$

By adding Roles to the system, two Dynamic Tuples are extended for each inserted Role. In this example, the two **Person** Dynamic Tuples are extended once, and the **University** one twice, resulting in the following Dynamic Tuple state. Moreover, the Dynamic Tuples start to overlap by the newly created Roles. In detail, dt_{John} overlaps with dt_{TUD} by the Role s_1 and dt_{Gert} with dt_{TUD} in the Role p_1. In addition, each Dynamic Tuple changed its Configuration. For instance, dt_{John} changed from $c_{min} = (Person, \emptyset, \emptyset)$ to $c_1 = (Person, \{Student\}, \emptyset)$. However, the respective Dynamic Tuples have the following state.

$$dt_{John} = (John, \{\{s_1\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset), dt_{TUD} = (TUD, \emptyset, \{\{s_1\}, \{p_1\}\})$$

Statement 4 The last statement establishes a relationship between the previously created **Student** and **Professor** Roles. Consequently, the instance $\mathfrak{u}$ is manipulated by creating a new entry in the *links* function.

```
INSERT INTO RELATIONSHIPTYPE supervises
  CONNECT Person p PLAYING Student s WHERE p WITH p.name="John"
  WITH Person p2 PLAYING Professor prof WHERE p2 WITH p2.name="Gert"
  AT University u WHERE u WITH u.name="TUD";
```

$$
\begin{aligned}
\mathfrak{u} = & (N = \{John, Gert\}, C = \{TUD\}, R = \{s_1, p_1\}, \\
& type = \{(John \rightarrow Person), (Gert \rightarrow Person), (TUD \rightarrow University), \\
& (s_1 \rightarrow Student), (p_1 \rightarrow Professor)\}, \\
& plays = \{(John, TUD, s_1), (Gert, TUD, p_1)\}, \\
& links(supervises, TUD) = \{(p_1, s_1)\})
\end{aligned}
$$

Dynamic Tuples are not affected by establishing Relationships, hence, they remain the same.

$$dt_{John} = (John, \{\{s_1\}\}, \emptyset), dt_{Gert} = (Gert, \{\{p_1\}\}, \emptyset), dt_{TUD} = (TUD, \emptyset, \{\{s_1\}, \{p_1\}\})$$

5.4 RSQL DATA QUERY LANGUAGE

Generally, a query language specification defines the syntax of well-formed query statements to retrieve stored data objects. The concepts of the database are usually reflected in the query language as well. For instance, SQL as declarative query language for the mathematical concept of a relation specifies tables and tuples as instance of those. Transferring this onto a query language for RSQL's database model, it has to support the notion of Dynamic Data Types and Dynamic Tuples. By building a query language on the basis of these notions, the separation of concerns within an entity is guaranteed, because this characteristic is inherited from the base data structure to the query language concepts.

At first the formal syntax of RSQL's query language is introduced and explained. To demonstrate the connection from the query language's expressions to the base database model data structures, various examples are employed and detailed subsequently to the syntax explanations. Furthermore, the linkage to the database operators is detailed by using several examples. Parts of the following syntax description have been published in [50, 49, 51].

5.4.1 Data Query Language Syntax

The data query language of RSQL consists of one statement only, the ⟨*select*⟩ statement. It can be combined with other ⟨*select*⟩ statements by using set operations. To combine several ⟨*select*⟩ statements, the ⟨*query*⟩ element is introduced as root element. A set operation between two ⟨*selects*⟩ may be a difference, an intersection or a union. To define which operation has to be performed on the level of Role for these set operations, a union and an intersection are available, resulting in a two-staged set operation design (see 4.4).

Generally, a ⟨*select*⟩ statement has three levels. The first level is called **intra Dynamic Tuple level** and describes the schema a Dynamic Tuple has to have. The second one is denoted as **inter Dynamic Tuple level** and relates Dynamic Tuples to others by using overlapping information or Relationship Types. Finally, the third level is entitled as **attribute level** and aims at attributes of Dynamic Tuples. The syntax description is illustrated in Figure 5.7. As it can be seen, each select statement in RSQL's data query language consists of three parts. A ⟨*projection-clause*⟩, a ⟨*from-clause*⟩, and an optional ⟨*where-clause*⟩.

The first one specifies several attributes among all available Dynamic Tuple attributes. Consequently, only the specified attributes will be available in the result. If all attributes should be returned, then a $*$ simplifies the listing of all attributes. It is located on the attribute level.

⟨*query*⟩ ::= ⟨*select*⟩ | ⟨*select*⟩ ⟨*query-junctor*⟩ ⟨*query*⟩

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

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

⟨*config-expression*⟩ ::= ⟨*rigid-def*⟩ (`FEATURING` ⟨*logical-expression*⟩)?
(`PLAYING` ⟨*logical-expression*⟩)?

⟨*rigid-def*⟩ ::= ⟨*rigid-name*⟩ ⟨*alias*⟩ | `_` ⟨*alias*⟩

⟨*rigid-name*⟩ := ⟨*ct-name*⟩ | ⟨*nt-name*⟩

⟨*logical-expression*⟩ ::= ⟨*rt-def*⟩ | ⟨*rt-def*⟩ ⟨*junctor*⟩ ⟨*logical-expression*⟩

⟨*rt-def*⟩ ::= (⟨*rt-name*⟩)? ⟨*rtAlias*⟩

⟨*relation-clause*⟩ ::= `RELATING` ⟨*log-rel-expression*⟩

⟨*log-rel-expression*⟩ ::= ⟨*rel-def*⟩ | ⟨*rel-def*⟩ ⟨*junctor*⟩ ⟨*log-rel-expression*⟩

⟨*rel-def*⟩ ::= ⟨*rtAlias*⟩ `WITH` ⟨*rtAlias*⟩ `IN` ⟨*ctAlias*⟩ `USING` ⟨*rst-name*⟩

⟨*where-clause*⟩ ::= ⟨*where-filter*⟩ (`,` ⟨*where-filter*⟩)*

⟨*where-filter*⟩ ::= ⟨*alias*⟩ `WITH` ⟨*predicate*⟩

⟨*junctor*⟩ ::= `AND` | `OR`

⟨*query-junctor*⟩ ::= `DIFF DT` | `DIFF R` | ⟨*dt-set-op*⟩

⟨*dt-set-op*⟩ ::= (`INTERSECT` | `UNION`) ⟨*role-level-set-op*⟩

⟨*role-level-set-op*⟩ ::= `INTERSECT` | `UNION`

⟨*projection-clause*⟩ ::= `*` | ⟨*alias*⟩.⟨*attribute*⟩ (`,` ⟨*alias*⟩.⟨*attribute*⟩)*

Figure 5.7: RSQL's Data Query Language Syntax Definition

The ⟨*from-clause*⟩ is the most important and complex clause in a ⟨*select*⟩ statement. It describes the constructs of the intra and inter Dynamic Tuple level. Each ⟨*from-clause*⟩ consists of several ⟨*config-expression*⟩ and optional ⟨*relation-clause*⟩. The general syntax of a ⟨*config-expression*⟩ is discussed in Section 5.3.3. However, this expression is included in RSQL's data query language syntax, because there is one important difference. In the ⟨*rt-def*⟩ of a ⟨*logical-expression*⟩, the Role Type name ⟨*rt-name*⟩ is optional, to specify overlap points in a query. Skipping the Role Type name results in a re-usage of a previously defined Role Type by re-referencing it. Such a Role Type re-usage construct specifies an overlap over two ⟨*config-expression*⟩.

However, along with the schema specifications, there may exist ⟨*relation-clauses*⟩ to specify the Relationships between Role Types. The ideas of this clause is, to specify which Roles of a certain Role Type are in a certain Relationship to other Roles of a particular Role Type and in which Compartment this Relationship takes place. All referenced Role Types and Compartment Types must be specified previously, thus, only re-usages are valid in this clause. This clause is initialized by the term `RELATING` followed by a ⟨*log-rel-expression*⟩. Like the ⟨*log-expression*⟩, this expression is recursive as well, to logically combine various of these expressions. The recursion's breakpoint is a ⟨*rel-def*⟩, consisting of two

Role Type aliases connected by the term `WITH` followed by the term `IN` and a Compartment Type alias ⟨*ctAlias*⟩. To specify the actual Relationship Type, the literal `USING` is stated by a subsequent Relationship Type name.

Finally, each ⟨*select*⟩ statement has an optional ⟨*where-clause*⟩, initialized by the term `WHERE` and represents the attribute level. Generally, a ⟨*where-clause*⟩ specifies several ⟨*where-filter*⟩ consisting of an ⟨*alias*⟩ in combination with a ⟨*predicate*⟩. The alias can be any alias that has been defined previously. The predicate can be anything, an equation, a check for a certain value, a comparison of two attributes, or anything else. However, a predicate is applied to a certain type alias only, to distinguish the predicate evaluation process between the whole data structure or only parts of it. For instance, the whole Dynamic Tuple can be filtered on the basis of the predicate or it can be applied on the Roles only. In the former case, the rigid is referenced and in the latter one a Role Type alias is referenced.

5.4.2 From Syntax to Logical Operators

The syntax of this data query language is defined with a strong relation to the underlying database model concepts and operators in mind. To demonstrate this relation, all operators are discussed in detail by explaining which parts of a ⟨*query*⟩ result in which operator and operator parametrization. Moreover, this discussion is augmented by small query examples that focus on the operator characteristics.

Configuration Selection A configuration selection is executed for each ⟨*config-expression*⟩ in a ⟨*select*⟩ statement. Hence, the operator plan will feature as many configuration selections as base data inputs are in the query. This represents the base data filter for Dynamic Tuples in suitable Configuration. As input, a set of Dynamic Tuples under a queried Dynamic Data Type is provided to this operator. Additionally, the ⟨*rigid-name*⟩ represents t_{cex}. α is represented by the ⟨*log-expression*⟩, while each ⟨*rt-name*⟩ in combination with ⟨*junctor*⟩ is added to logical proposition. Is there only an alias for the Role Type, the respective ⟨*rt-name*⟩ is added from the ⟨*config-expression*⟩, which specifies this Role Type with a ⟨*rt-name*⟩. Depending on the dimension, either playing or featuring, the Role Type names are annotated with an overline and underline, respectively. For instance, imagine the following query.

```
1 SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa;
```

As input set, a set of Dynamic Tuples typed with the Dynamic Data Type **Person**, will be provided to the operator. The initial configuration selection will feature a $t_{cex} = Person$ and $\alpha = \overline{Student} \wedge \overline{StudentAssistant}$.

Configuration Projection A configuration projection is executed for each ⟨*config-expression*⟩ as well. It filters the input Dynamic Tuples for queried Role Types only. Thus, α is represented by a logical proposition consisting of all ⟨*rt-name*⟩ in logical combination with ⟨*junctor*⟩. Hence, as many ⟨*log-expression*⟩ are under a certain ⟨*config-expression*⟩, as many Role Types will appear in α. Moreover, the Role Types are annotated by their corresponding dimension they belong to.

For example, imagine the same query as in the configuration selection again. All Dynamic Tuples of a Person will be provided to this operator and α is populated with $\overline{Student} \wedge \overline{StudentAssistant}$. The output is a set of Dynamic Tuples with trimmed dimensions.

Role Matching A role matching operator is created within the operator plan for each **role matching unit**. In general, such a unit consists of ⟨*rt-defs*⟩ that do not specifying a ⟨*rt-name*⟩, but reference an alias only. Moreover, it is established between two ⟨*config-expressions*⟩, such that each reference points to the same ⟨*config-expression*⟩. If they do not point the same ⟨*config-expression*⟩, they form a separate role matching unit and thus, separate operators. This alias reference and ⟨*config-expression*⟩ overlap is based upon the overlap of Dynamic Data Types and Dynamic Tuples.

The most basic role matching unit is a single ⟨*rt-def*⟩ that does not specify a ⟨*rt-name*⟩. In case multiple ⟨*rt-def*⟩ point from one ⟨*config-expression*⟩ to same opponent one, the corresponding Role Types are collected in the role matching unit and combined with the ⟨*junctor*⟩ between them. However, such an alias only construct defines an overlap of two ⟨*config-expressions*⟩ at the point of this alias. In case the role matching unit consists of more than one Role Type, the two ⟨*config-expressions*⟩ overlap in multiple points. However, the parameter α is populated with the Role Types referenced by the alias and combined with the ⟨*junctor*⟩ between them. The affected ⟨*config-expressions*⟩ define the input sets of Dynamic Tuples for this operator. As example, imagine the following query.

```
SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa,
  Uni u FEATURING s AND sa;
```

The **Student** and **StudentAssistant** Role Types are referenced twice each, once in the ⟨*config-expression*⟩ for the Dynamic Data Type **Person** and once for the **Uni**. Additionally, both are conjunctively connected to each other. Consequently, the corresponding role matching unit consists of Student and StudentAssistant combined with and logical conjunction. Moreover, as it can be seen, both Role Type references are referenced once in the playing dimension and once in the featuring dimension. Otherwise, if there would be a reference in two identical dimensions, the corresponding Dynamic Tuples cannot overlap. However, the κ operator created for this query is $\kappa_{Student \wedge StudentAssistant}(DT_{Person}, DT_{Uni})$.

Relationship Matching A relationship matching operator is created for each ⟨*log-rel-expression*⟩ in the query. It filters three sets of Dynamic Tuples with respect to their relationships in a certain Compartment. The Relationship Type is specified by a ⟨*rst-name*⟩. Moreover, the Compartments in which a relationship has to take place, is defined by the ⟨*ctAlias*⟩. This information is combined to retrieve the corresponding relationship tuples in *links*. The first and second ⟨*rtAlias*⟩ specify Role Types previously defined in various ⟨*config-expressions*⟩. Thus, a ⟨*rel-def*⟩ reuses Role types only, but does not define new one. As input Dynamic Tuple sets, the corresponding ⟨*config-expressions*⟩, in which the Role Types are defined, are used. As example query, imagine the following statement.

```
SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa,
  Person p2 PLAYING Professor prof,
  Uni u FEATURING s AND prof,
  RELATING prof WITH s IN u USING supervises;
```

The ⟨*log-rel-expression*⟩ consists of a single ⟨*rel-def*⟩ only. It specifies the Relationship Type **supervises**. Moreover, the Role Type **p** has to be in relation to **s** in the Compartment **u**. All aliases have been previously specified in various ⟨*config-expressions*⟩, one for each Relationship Type participant. Thus, the input sets depend on these ⟨*config-expressions*⟩. The resulting relationship matching operator for this example query is $\Omega_{supervises}(DT_{Person2}, DT_{Person}, DT_{Uni})$

Type Union A Type union operator is triggered by a wildcard underscore in a ⟨*config-expression*⟩, in case the listed Role Types can be filled several times. This means, a wildcard on a Role Type that can be filled by one player type only, will not trigger a τ operator. In general, this operator combines various Dynamic Data Types, that have to be specified in individual ⟨*config-expressions*⟩. This operator merges two distinct Dynamic Data Types by combining their core type and assigns a new united core type to each core. Because there is no core type specified, the database schema is utilized to find the Dynamic Data Types this Role Type is filled and contained in, respectively. This is similar to a wildcard in the ⟨*projection-clause*⟩, but it automatically rolls out the Dynamic Data Types instead of attributes. The output will be used as standard input for other operators and do not need any individual management. Generally, an operator plan will feature $n - 1$ τ operators, with n being the amount of occurrences of the affected Role Types in the *fills* relation. As example query, imagine the following statement.

```
SELECT * FROM _ w PLAYING StudiesCourse;
```

This query defines a type union operator with the alias w on the Role Type **StudiesCourse**, which can be filled by both Natural Types, **Lecture** and **Seminar**. The affected Dynamic Data Types can be derived from the database schema using the *fills* relation. The Role Type **StudiesCourse** is included twice, hence, there will be one τ operator to combine both player types. Consequently, the operator input sets are $DT_{Lecture}$ and $DT_{Seminar}$. Additionally, the *type* function is consumed and manipulated with the new union type. The resulting operator is $\tau(DT_{Lecture}, DT_{Seminar})$ The output is a single output stream of Dynamic Tuple comprising all entities of the first and second input set and the manipulated *type* function.

Intersection A standard intersection on the Dynamic Tuple level and an intersection on the Role level is triggered by each AND ⟨*junctor*⟩ that is not part of role matching unit. It intersects two sets of Dynamic Tuples having the same core and unites them within a new set. The parameter $\circ$ within a ⟨*select*⟩ will always be an intersection $\cap$. In case a union is required at this point, the set operations between several ⟨*select*⟩ statements has to be used. However, the Role Types included in the sets RT_a and RT_b depend on the placement of this operator in the operator plan. But, the directly connected ⟨*rt-names*⟩ will be definitely part of these. For instance, assume the successive query.

```
SELECT * FROM Person p PLAYING Student s AND TeamMember tm,
  Uni u FEATURING s,
  SportTeam st FEATURING tm;
```

As it can be seen, the **Student** and **TeamMember** Role Types are not part of the same role matching unit, because they have dependencies to different ⟨*config-expressions*⟩. Consequently, the AND will result in an intersection operator. In detail, RT_a contains the element $Student$ and RT_b the $TeamMember$. Depending on upstream processed operators, these sets may vary. However, in this case no such operators are assumed. Hence, the resulting operator is $\cap_{\cap}^{\{Student\},\{TeamMember\}}(DT_{Person1}, DT_{Person2})$. This operator will combine Dynamic Tuples of the Dynamic Data Type **Person** that play a Role **Student** with Dynamic Tuples of **Persons** that play a **TeamMember** Role. Moreover, all filtered **Student** Roles will come from the first input set and all **TeamMember** Roles from the second one. All other Roles are intersected.

Union A standard union on the Dynamic Tuple level and a union on the Role level is created for each OR ⟨*junctor*⟩ that is not part of a role matching unit. It unites two sets of Dynamic Tuples and for those included in both sets it performs the action specified in $\circ$. Within a certain ⟨*select*⟩ this action will always be a $\cup$. In case an intersection is required, set operations between several ⟨*select*⟩ statements have to be used. As in the intersection operator, the population of RT_a and RT_b depends on the operator placement within the operator plan, but the directly connected Role Type will definitely be part of these sets. As example, imagine the following query.

```
SELECT * FROM Person p PLAYING Student s OR TeamMember tm,
  Uni u FEATURING s,
  SportTeam st FEATURING tm;
```

The Role Types **Student** and **TeamMember** are connected by an `OR` and are not part of the same role matching unit. Hence, the union operator is created having the sets $RT_a = \{Student\}$ and $RT_b = \{TeamMember\}$. The overlapping Dynamic Tuples will be united to a new one, by keeping **Student** Roles from the Dynamic Tuple in DT_a and **TeamMember** ones from that one in DT_b. All other Role in both dimension will be united. However, the resulting operator for this query is $\cup_{\cup}^{\{Student\},\{TeamMember\}}(DT_{Person1}, DT_{Person2})$.

Set Operations A set operator is issued by a ⟨*query-junctor*⟩ that connects two ⟨*select*⟩ statements. As input, it consumes the output of both queries and performs the desired action, either a difference on the Dynamic Tuple or Role level, an intersection, or a union. For the latter ones, the Role level action can be chosen between an intersection or a union. The corresponding sets of Role Types in case of an intersection or union will be empty. A query can have multiple output sets of Dynamic Tuples and a set operator is created for each of those output sets. Moreover, it is assumed, both queries produce the same amount of output sets. However, the first output of the first query will be the first input of the first set operator and the first output of the second ⟨*select*⟩ the second input. As a consequence, both queries have to describe equally typed outputs. Otherwise, the correct input assignments for the operators cannot be made. In some cases, this assignment can be performed by analyzing the type of the output sets. For instance, an output set containing Dynamic Tuples of **Persons** can be intersected with other **Persons** only, but in case multiple output sets describe **Persons**, the assignment is ambiguous. As example, assume the following valid query.

```
SELECT * FROM Person p PLAYING Student s OR TeamMember tm;
  DIFF DT
SELECT * FROM Person p PLAYING StudentAssistant sa;
```

This query builds the difference between Dynamic Tuples of the Dynamic Data Type Person playing Roles of the Role Types **Student** and **TeamMember** and those Dynamic Tuples of the type Person that play a Role of a **StudentAssistant**. Rephrased this means, return all **Persons** who are a **Student** and a **TeamMember**, but not a **StudentAssistant**. Both queries produce an output of Persons, hence the query is valid. The resulting operator is $DT_{Person1} \setminus DT_{Person2}$. An invalid query in the sense of a Dynamic Tuple based set operation is present in the following.

```
SELECT * FROM Person p PLAYING Student s OR TeamMember tm,
  Person p2 PLAYING Professor prof,
  Uni u FEATURING s AND prof;
 DIFF DT
SELECT * FROM Person p PLAYING StudentAssistant sa;
```

It is not clear if the difference should be performed on the first or second output of the first ⟨*select*⟩ statement, because both describe Dynamic Tuples of **Persons**.

Attribute Filter An attribute filter operator is created for each ⟨*where-filter*⟩. The affected Dynamic Tuples or Roles are specified by the first ⟨*alias*⟩ with a subsequent `WITH`. After this `WITH`, the predicate is specified. This can be related to the Dynamic Tuple the ⟨*alias*⟩ is part of, or anything else. In particular, there is no constraint for the predicate. Depending on the predicate complexity and the referenced attributes, several other inputs are required along with these Dynamic Tuples that have to be manipulated. As example imagine the following query that asks for **Persons** that have the same name as the **Professor** that **supervises** them within a certain **University**.

```
SELECT * FROM Person p PLAYING Student s,
  Person p2 PLAYING Professor prof,
  Uni u FEATURING s AND prof;
  RELATING prof WITH s IN u USING supervises
  WHERE p WITH p.name = p2.name;
```

The corresponding attribute selection operator is specified by $\sigma^{Person,\{Student,Professor\}}_{p.name=p2.name}(DT_{Person}, DT_{Person2}, DT_{University})$. In detail, it defines only one ⟨*where-filter*⟩. This filters Dynamic Tuples of the first ⟨*config-expression*⟩, because the ⟨*alias*⟩ is `p`. Consequently, the first input set of this operator is DT_{Person}. The `WITH` defines the predicate `p.name = p2.name`, which states that both Person Dynamic Tuples have to be equal in their names. Moreover, the first and second ⟨*config-expression*⟩ are connected by a third one and an additional Relationship. They apply the filter with respect to this overlapping information. Thus, additional input sets are required, especially $DT_{Person2}$ and $DT_{University}$. Additionally, the Dynamic Tuples have to overlap by the Role Types **Student** and **Professor**, which is extracted from the overall statement and not explicitly stated. Finally, this statement produces a filtered version of the Dynamic Tuples specified in the first ⟨*config-expression*⟩. All other input sets remain unchanged.

5.4.3 Simple Config-Expression Example

Assume the following query as example for a simple and basic ⟨*config-expression*⟩.

```
SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa;
```

This example query searches for Dynamic Tuples under the Dynamic Data Type **Person** that additionally play Roles of the Role Type **Student** and **StudentAssistant**. This means, their Configuration has to have both Role Types in the filling dimension. There might be more, but these are mandatory. As it can be seen, a single ⟨*config-expression*⟩ describes Configurations of a certain Dynamic Data Type.

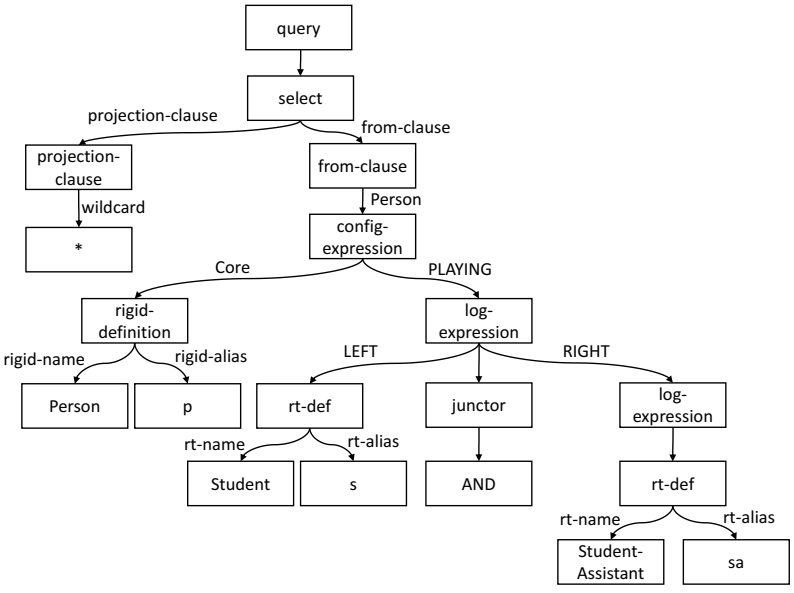


Figure 5.8: Abstract Syntax Tree For the Simple Config-Expression Example Query

The abstract syntax tree of this query is illustrated in Figure 5.8. On the top, there is the ⟨*query*⟩ element as root. This has only one connected node, the ⟨*select*⟩, which has two nodes connected,

the ⟨*projection-clause*⟩ and ⟨*from-clause*⟩. In the ⟨*projection-clause*⟩ the wildcard is referenced, which specifies all attributes will be available in the result. The ⟨*from-clause*⟩ has one ⟨*config-expression*⟩. This is split into a ⟨*rigid-def*⟩ and a ⟨*log-expression*⟩ the playing dimension. The former defines the ⟨*rigid-name*⟩ as *Person* and specifies the alias *p*. The latter has a ⟨*rt-def*⟩ in combination with a ⟨*junctor*⟩ and another ⟨*log-expression*⟩. This Role Type definition specifies the ⟨*rt-name*⟩ as *Student* in combination with the alias *s*. Furthermore, the ⟨*junctor*⟩ is an `AND` and the second ⟨*log-expression*⟩ terminates with another ⟨*rt-def*⟩, which specifies the ⟨*rt-name*⟩ *StudentAssistant* with an alias *sa*.

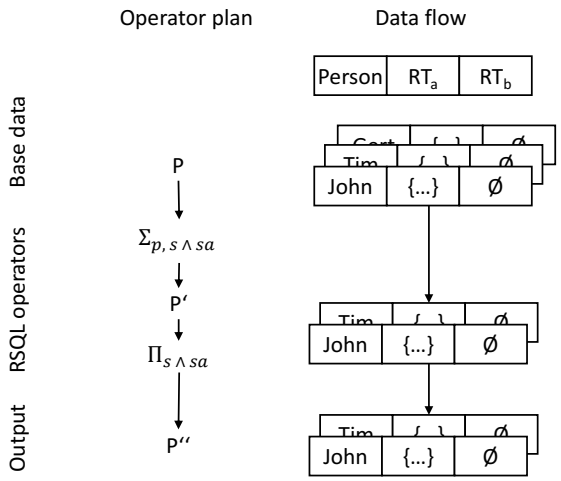


Figure 5.9: Operator Plan and Data Flow Chart For the Simple Config-Expression Example Query

Based on this syntax tree an operator plan can be created, as shown in Figure 5.9. It features one individual Dynamic Tuple stream only, because the query specifies only one ⟨*config-expression*⟩. Hence, there is neither an overlap nor a relationship involved. Moreover, it can be seen that the Dynamic Tuples in the stream are not valid with respect to the base data model, but to the operational data model. In the base one each Role has to be in two Dynamic Tuples, but in the operational one this constraint is relaxed. However, this query results in two operators. One to select the Dynamic Tuples based on their Configuration and one to trim these to the queried Role Types only. In detail, Σ is parametrized with $t_{cex} = Person$ and $\alpha = \overline{Student} \wedge \overline{StudentAssistant}$. All Dynamic Tuples that do not have Roles of both of these Role Types in their playing dimension will be eliminated. The output of this operator is denoted as P', which is the input for the subsequent operator. Π trims Dynamic Tuples to the queried Role Types, thus, α is parametrized with $\overline{Student} \wedge \overline{StudentAssistant}$. It consumes P' and produces P''. Dynamic Tuples are not eliminated in the operator.

5.4.4 Non-Overlapping Config-Expressions Example

A query with non-overlapping ⟨*config-expressions*⟩ has several of these expressions, but without any overlap. In fact, they form individual chains in the operator plan. Such non-overlapping ⟨*config-expressions*⟩ are syntactically characterized by having ⟨*rt-def*⟩ with a ⟨*rt-name*⟩ and ⟨*alias*⟩ only. This means, they do not utilize references of previously specified Role Types by having a ⟨*rt-def*⟩ with only an ⟨*alias*⟩. For instance, imagine the following query, which queries for **Persons** being a **Student** and **StudentAssistant**, and a **University** featuring **Students**. Additionally, the **Student** Role Types in both ⟨*config-expressions*⟩ may not share the same instance.

```
1 SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa,
2   University u FEATURING Student s2;
```

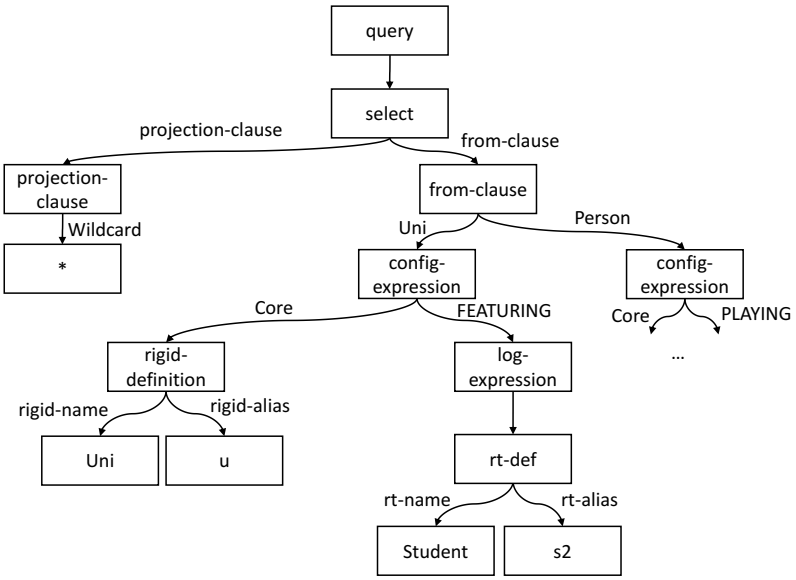


Figure 5.10: Abstract Syntax Tree For the Unrelated Config-Expression Example Query

The abstract syntax tree for this example is illustrated in Figure 5.10. As every RSQL query that does take usage of set operations, the ⟨*query*⟩ root element has only one ⟨*select*⟩ child. This is split into two subelements, the ⟨*projection-clause*⟩ and ⟨*from-clause*⟩. The former is a wildcard, which references all available attributes. The latter specifies two individual ⟨*config-expressions*⟩. One for the core **University** and one for the **Person**. Note, the subtree of the Person ⟨*config-expression*⟩ is the same as illustrated in Figure 5.8, but for arrangement purposes this subtree is omitted in the unrelated ⟨*config-expression*⟩ illustration. However, no ⟨*rt-def*⟩ utilizes the opportunity to reuse a previously specified Role Type. In the example, neither the University ⟨*config-expression*⟩ nor the Person one does. This situation results in an independent Dynamic Tuple processing of both referenced Dynamic Data Types as shown by the operator plan depicted in Figure 5.11.

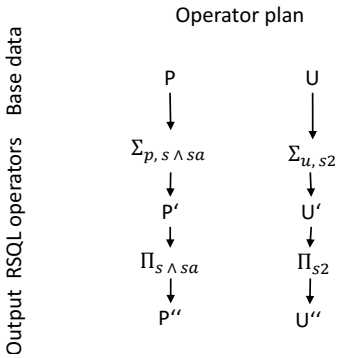


Figure 5.11: Operator Plan For the Unrelated Config-Expression Example Query

This operator plan has two input sets, one for each ⟨*config-expression*⟩. As outlined in the section from syntax to operators (see Section 5.4.2), each of these expressions issues a configuration selection and configuration projection on all specified Role Types. Consequently, the operator plan on the Person input set has a configuration selection with a $t_{cex} = p$ and an $\alpha = s \wedge sa$. This produces the output P', which is the input for the configuration projection. This projection is performed on the same α and produces the Person's operator thread output P'' For the university operator thread has a configuration selection for the core type $t_{cex} = u$ and an α populated with $s2$ only. This results in the intermediate result U'. Moreover, this acts as input for the configuration projection on $s2$. The final output of this operator chain is U''. Finally, the query output consists of P'' and U''.

5.4.5 Overlapping Config-Expressions Example

In contrast to non-overlapping ⟨*config-expressions*⟩, the overlapping ones reuse Role Type definitions. Additionally, such queries are characterized by featuring operators that combine two individual operator chains as input into one operator. The Role Type definition explicitly states that the Roles coming from one input stream have to included in the second one, too. In the abstract syntax tree, such queries are identified by a ⟨*rt-def*⟩ having an ⟨*alias*⟩ only. The corresponding Role Type can be found in another ⟨*config-expression*⟩. Moreover, an overlap from one ⟨*config-expression*⟩ to another one builds a role matching unit (see Section 5.4.2). Pointing multiple times from one ⟨*config-expression*⟩ to the same other one results in a role matching unit that logically combines several Role Types with each other. For instance, assume the following query asking for **Persons** being a **Student** and **StudentAssistant**, and the **University** these Roles are featured in.

```
1 SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa,
2   University u FEATURING s AND sa;
```

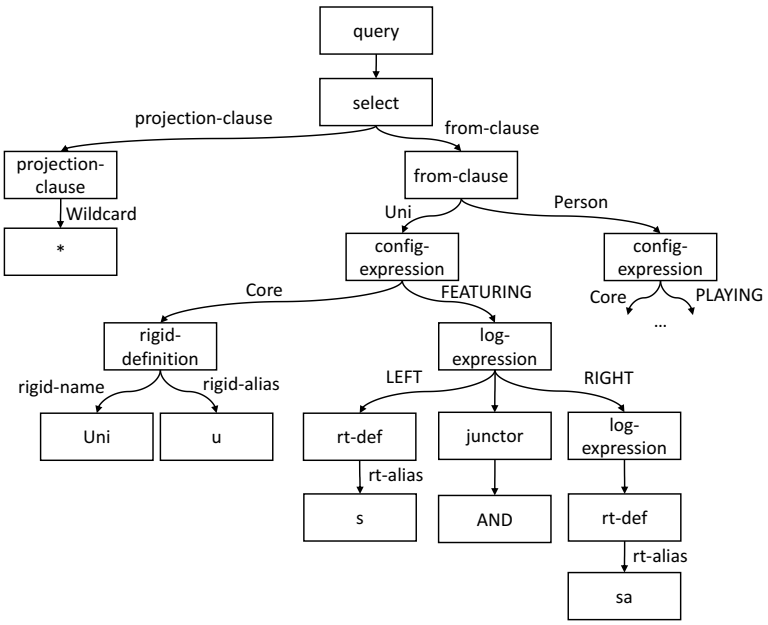


Figure 5.12: Abstract Syntax Tree For the Overlapping Config-Expression Example Query

The respective abstract syntax tree is presented in Figure 5.12. As root element, it has a ⟨*query*⟩ with only one ⟨*select*⟩ child element. The ⟨*projection-clause*⟩ references all available attribute by using the star (*) wildcard. The ⟨*from-clause*⟩ features two ⟨*config-expressions*⟩, one for the **Universities** and one for the **Persons**. Each specifies a core and a set of logically combined Role Type definitions. In case of the University these ⟨*rt-defs*⟩ reference an ⟨*alias*⟩ only, especially *s* and *sa*. Both relate to ⟨*rt-defs*⟩ in the Person ⟨*config-expression*⟩, which is not shown in this abstract syntax tree, but in Figure 5.8. Based on the fact that both refer to the same opponent ⟨*config-expression*⟩, they form a role matching unit with the ⟨*junctor*⟩ as logical operator between them. In this example, this ⟨*junctor*⟩ is a logical `AND`.

A possible resulting operator plan is shown in Figure 5.13. It features two distinct input sets, one for each ⟨*config-expression*⟩. Moreover, each ⟨*config-expression*⟩ produces a configuration selection and configuration projection operator, which is independent of the overlapping information. Hence, the input sets for Person and University go through their respective configuration selection and configuration projection operators, both filter on **Student** and **StudentAssistant** combined with a logical conjunction. These steps produce the Dynamic Tuple sets P'' for those of the Dynamic Data Type

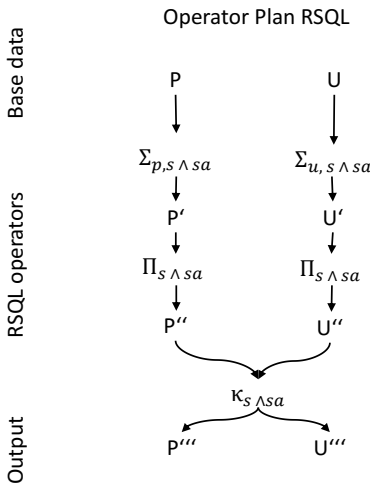


Figure 5.13: Operator Plan For the Overlapping Config-Expressions Example Query

Person and U'' for the **University** ones. These sets are the input for the overlapping operator κ that filters both Dynamic Tuple sets for overlapping Roles of the Role Type **Student** and **StudentAssistant**.

Because both Role Types references point to the same opponent ⟨*config-expression*⟩, they are processed as combined role matching unit. Hence, the role matching parameter for κ is $s \wedge sa$. Furthermore, each input set forms a new output sets, thus, P'' and U'' are processed and transformed to P''' and U''', respectively. Finally, these output sets represent the query output at the same time, because there is no operator left.

5.4.6 Relationships Example

Relationships are part of a query in case a ⟨*relation-clause*⟩ appears as child of a ⟨*from-clause*⟩. It requires at least two Role Type definitions, usually specified in different ⟨*config-expressions*⟩ and a ⟨*config-expression*⟩ acting having a Compartment as core. Additionally, the information held in the *links* function are necessary. As example, imagine the following query, which asks for **Persons** being a **Student** and **StudentAssistant**, other **Persons** playing the Role of a **Professor**, a **University** holding all these Roles and the **Professor** has to **supervise** the **Student** within this **University**.

```
1 SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa,
2   Person p2 PLAYING Professor prof,
3   University u FEATURING s AND sa AND prof,
4   RELATING prof WITH s IN u USING supervises;
```

An excerpt of the resulting abstract syntax tree is presented in Figure 5.14. As the previous examples, this one has a ⟨*query*⟩ element as root and a single ⟨*select*⟩ as child. The ⟨*projection-clause*⟩ references the wildcard. In contrast to the previous example, the ⟨*from-clause*⟩ in this example has three ⟨*config-expressions*⟩ and an additional ⟨*relation-clause*⟩. These three ⟨*config-expressions*⟩ are not fully shown in the abstract syntax tree illustration, in fact they are adumbrated. Assume, they are constructed following the rules in the previous examples. The import path in this example is the ⟨*relation-clause*⟩, which holds one ⟨*log-rel-expression*⟩. Because there is only one Relationship involved in this example, the ⟨*log-rel-expression*⟩ is not recursive and results in a ⟨*rel-def*⟩ only. This definition reuses Role Type definitions by referencing a ⟨*rtAlias*⟩ only. In this scenario it is *prof* and *s*, both previously defined in their respective ⟨*config-expression*⟩. The third component is the Compartment, in which Roles of these Role Types are located in. This is specified by a core reuse, especially a ⟨*ctAlias*⟩.

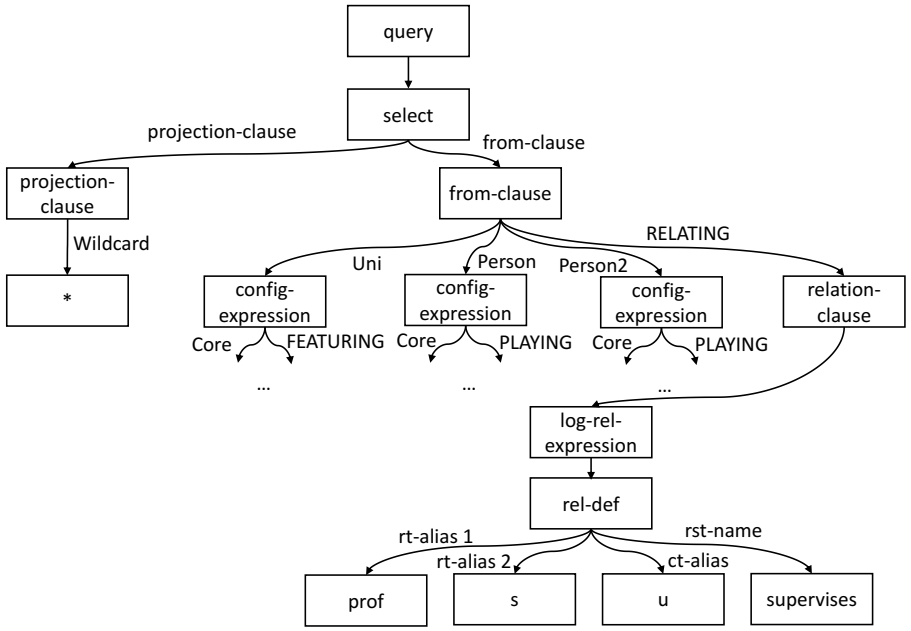


Figure 5.14: Abstract Syntax Tree For the Overlapping Config-Expression Example Query

Finally, the ⟨*rst-name*⟩ specifies the Relationship Type, in this case it is **supervises**. Note, multiple of these ⟨*log-rel-expressions*⟩ can be logically combined with each other. However, a possible operator plan for this example query is shown in Figure 5.15.

This example's operator plan has four inputs and four outputs. Three inputs are specified by the ⟨*config-expressions*⟩ and the last input is the $links$ function. In the first place, each of these expressions forms an individual operator path including a configuration selection and configuration selection on the corresponding Role Types. Next, there exist two role matching units, one between p and u and the other one between $p2$ and u. Consequently, there exist two κ operators. The first is applied on the **Person** Dynamic Tuples coming from p and the **Universities** that are part of the u stream. The respective outputs are denoted as P''' and U'''. P''' holds Dynamic Tuples with only these **Student** and **StudentAssistant** Roles that find a partner in the University Dynamic Tuples. U''' consists of Dynamic Tuples of **Universities** that find a partner Roles in the **Person** stream p. Moreover, the second role matching unit includes the Professor Role Type only and is applied on U'' and $P2''$ resulting in U^4 and $P2'''$.

After these steps there exist two different sets of University Dynamic Tuples, one filtered with respect to **Student** and **StudentAssistant** and the other one for **Professor**. To consolidate these two versions, the intersection operator is executed, because both role matching units are conjunctively combined. The resulting operator is parameterized with s, sa for RT_a because the left part of this operator is assumed to be U'''. Additionally, the second parameter RT_b is set with $prof$. Thus, the second input set is U^4. This intersection operator produces U^5, consisting of Dynamic Tuples that find a partner Roles in p and $p2$.

Finally, the Ω operator is executed on **supervises** while consuming P''' for the **Student** Roles, $P2'''$ for the **Professor** Roles, and U^5 as **University** Compartments. Moreover, $links$ is put into this operator. As output, it produces the query output P^4, $P2^4$, U^6, and $links'$. The first holds Dynamic Tuples of **Persons** being a **Student** and **StudentAssistant** at a certain **University** and **supervised** by a **Professor** at the same **University**. The second consists of **Persons** being a **Professor** at a certain **University** and **supervising Students** at this one. The output U^6 holds Dynamic Tuples of **Universities** that have **Students**, **StudentAssistants**, **Professors**, and these **Professors supervise** certain **Students** at this **University**. Finally, the $links'$ only includes these tuples of **Professors** and **Student** Roles that are also included the sets P^4 and $P2^4$ for the **Universities** in U^6.

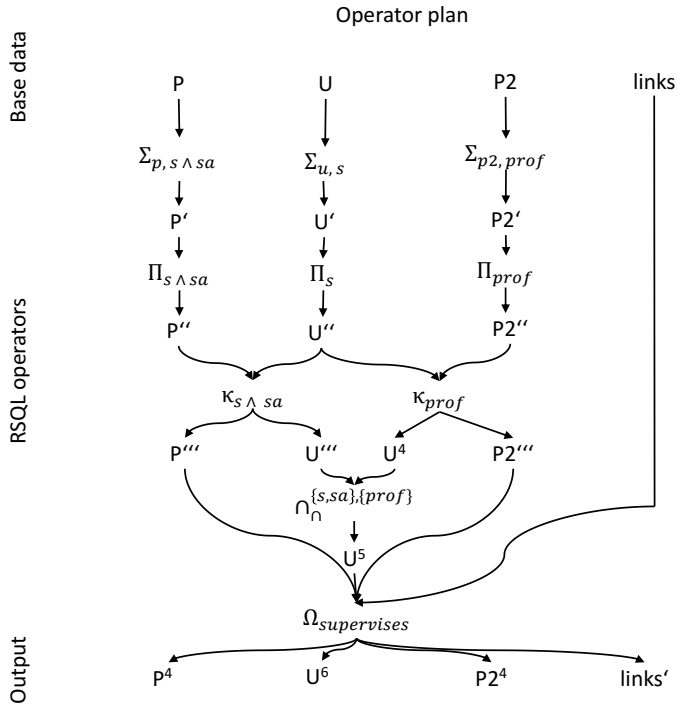


Figure 5.15: Operator Plan For the Relationship Example Query

5.4.7 Dynamic Tuple Attribute Selection Example

The ⟨*where-clause*⟩ is the third possible arm underneath the ⟨*select*⟩ element and the only one that is optional. It filters the whole Dynamic Tuple or only parts of it based on a given predicate. Generally, it is applied on a single Dynamic Tuple stream only, but may require additional streams to evaluate the predicate. In contrast to a traditional `WHERE`, like it executed in SQL, the attribute filter in RSQL explicitly specifies on which element is evaluated. This is caused by the underlying data structure and query execution, which does not merge several input streams into a single one, in fact the input streams remain separated. However, the filter element is defined by an ⟨*alias*⟩, which can be an alias of a core or a Role Type. Moreover, the filter predicate is totally independent of this alias and may not include this. As simple example, imagine the following query. It queries for **Persons** playing Roles of **Student** and **StudentAssistant** and additionally require a **Student's** start year later than 1985.

```
1 SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa,
2   WHERE p WITH s.year > 1985;
```

The corresponding abstract syntax tree is illustrated in Figure 5.16. As each example query, there exists only one ⟨*query*⟩ element as root. In this example it has only one child, a ⟨*select*⟩ statement, consisting of a ⟨*projection-clause*⟩, a ⟨*from-clause*⟩, and a ⟨*where-clause*⟩. The first references a wildcard, while the second features a single ⟨*config-expression*⟩. This expression is not fully shown in this example's abstract syntax tree illustration, but it is equal to that one shown in Figure 5.8. However, this example focuses on the ⟨*where-clause*⟩, which has only one child element, a ⟨*where-filter*⟩. This consists of an ⟨*alias*⟩ `p` and a ⟨*predicate*⟩ `s.year > 1985`. This predicate reuses the Role Type definition of the Student and a valid attribute. The reuse references the same ⟨*config-expression*⟩ as the ⟨*alias*⟩ does, hence, no additional information are required to evaluate the predicate.

Moreover, a Dynamic Tuple will pass this operator in case at least one of the Student Role fulfills the predicate. The Roles itself will not be filtered, because the filter references a core. In contrast,

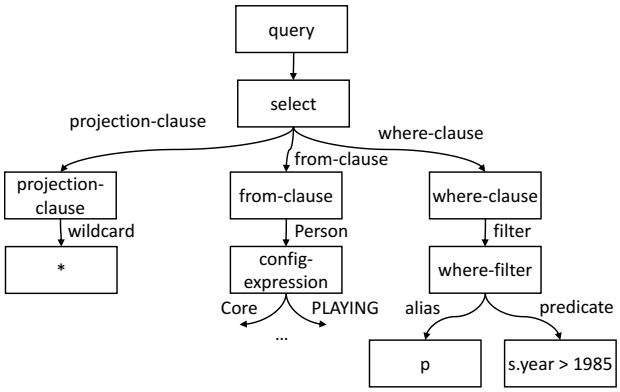


Figure 5.16: Abstract Syntax Tree For the Attribute Selection on a Core

an ⟨*alias*⟩ referring to a Role Type would filter the Role of this Role Type. That means, varying the input parameter for the filter type varies the output significantly, such that there will be a different output set in case the whole Dynamic Tuple is filtered than in a case only the Roles are filtered. The query's respective operator plan is presented in Figure 5.17. It consists of one stream only, because one ⟨*config-expression*⟩ is used. Moreover, the attribute selection operator is parametrized with `p` as filter type and the corresponding predicate. Overlapping are not provided, this, the respective set is empty.

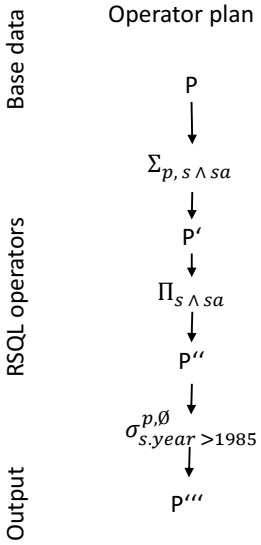


Figure 5.17: Operator Plan For the Attribute Selection on a Core

5.5 RSQL QUERY PROCESSING

Operator-based query processing pursues the goal of creating the same results independent from the individual operator placement in the query execution plan. This enables the logical optimization of operator plans and a lot of effort has been put into the research area, especially to optimize relational operator plans. An important prerequisite is the processing of coherent data structures. For instance, relational query processing uses the relational algebra to compute relational query results. It uses tables only, thus, each operator consumes one or more tables and produces one. Transferred to RSQL this prerequisite holds, too. It consumes sets of Dynamic Tuples and produces such. However, the

query processing is quite different to the well-known relational one, because it is built upon independent Dynamic Tuple streams and a query produces multiple output sets, instead of only one like a relational query.

For instance, assume the operator plans illustrated in Figure 5.18. On the left-hand side an RSQL operator plan for the overlapping config-expression example (see 5.4.5) is depicted. On the right-hand side a relational operator plan joining four base tables and a closing selection. As it can be seen in direct comparison, the relational query plan output consists of a single table, whereas the RSQL output has two sets of Dynamic Tuples. However, all tuple streams in the relational operator plan are united into a single output. In contrast, the streams remain independent of each other in the RSQL query plan. In fact, the input sets are manipulated using other input sets, but they are not united (except for the τ operator).

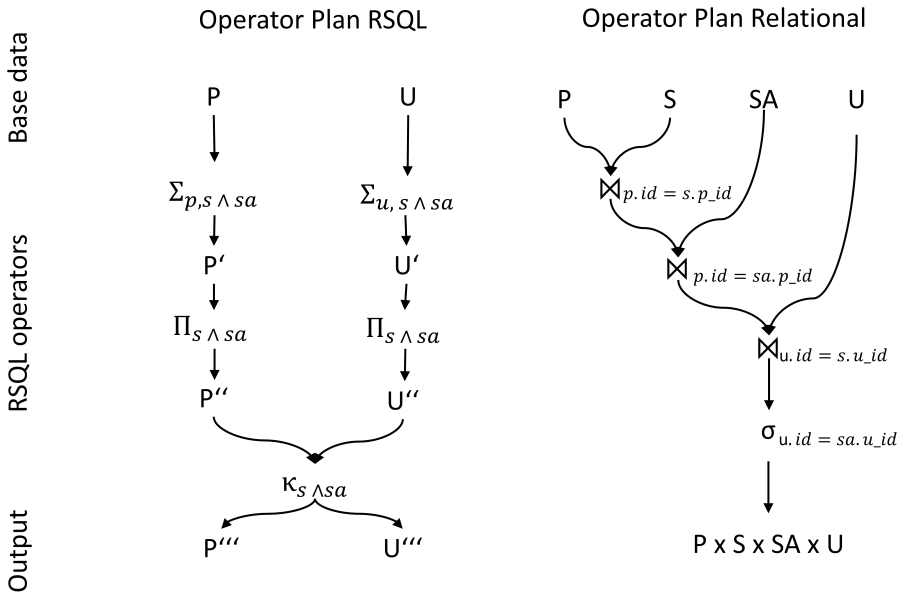


Figure 5.18: RSQL and Relational Operator Plan in Comparison

As a consequence, the relational result mixes several entity information into a single tuple and distributes a one entity over possibly multiple tuples. To restore an entity, several tuples have to be read. Moreover, intermediate results may grow rapidly, depending on the cardinality of the base tables and the interconnection ration of the individual tuples. However, RSQL's query processing may produce invalid intermediate result, caused by the independent input set processing.

5.5.1 Invalid Intermediate Results

Invalid intermediate result may be caused by each Dynamic Tuple manipulating operator, like the role matching or relationship matching operator. This situation is caused by the independent Dynamic Tuple input set processing and a Role's residence in two different Dynamic Tuples. Generally, each input set is handled independently of the other inputs, but several inputs are processed simultaneously in a certain operator, like the role matching operator. Hence, such an operation is performed on a certain state of the input streams and subsequently executed operators are not reflected in this state. This may causes invalid states of Dynamic Tuples and thus, invalid intermediate results. Especially, in case a Dynamic Tuple in combination with its Roles is eliminated in a later operator. The included Roles are deleted from the intermediate result of the currently processed Dynamic Tuples, but not in the Roles' other Dynamic Tuples. As example, imagine the following query.

```
SELECT * FROM Person p PLAYING Student s AND TeamMember tm,
  Uni u FEATURING s,
  SportsTeam st FEATURING tm;
```

This query comprises two role matching unit resulting in two role matching operators, one between **Person** and **University** by the overlapping **Student**, and one for **Person** and **SportsTeam** with the overlapping point **TeamMember**. This situation causes two versions of the Person input sets, which is united by an intersection operator. Each Dynamic Tuple that is present in one of these two versions only, will be eliminated by the intersection. However, the overlapping opponent, for instance the **University** Dynamic Tuples, will not reflect this elimination, especially the **Student** Roles of eliminated **Persons** will remain in the **University** Dynamic Tuples. As illustration assume the situation depicted Figure 5.19, which represents a query execution on a valid operator plan.

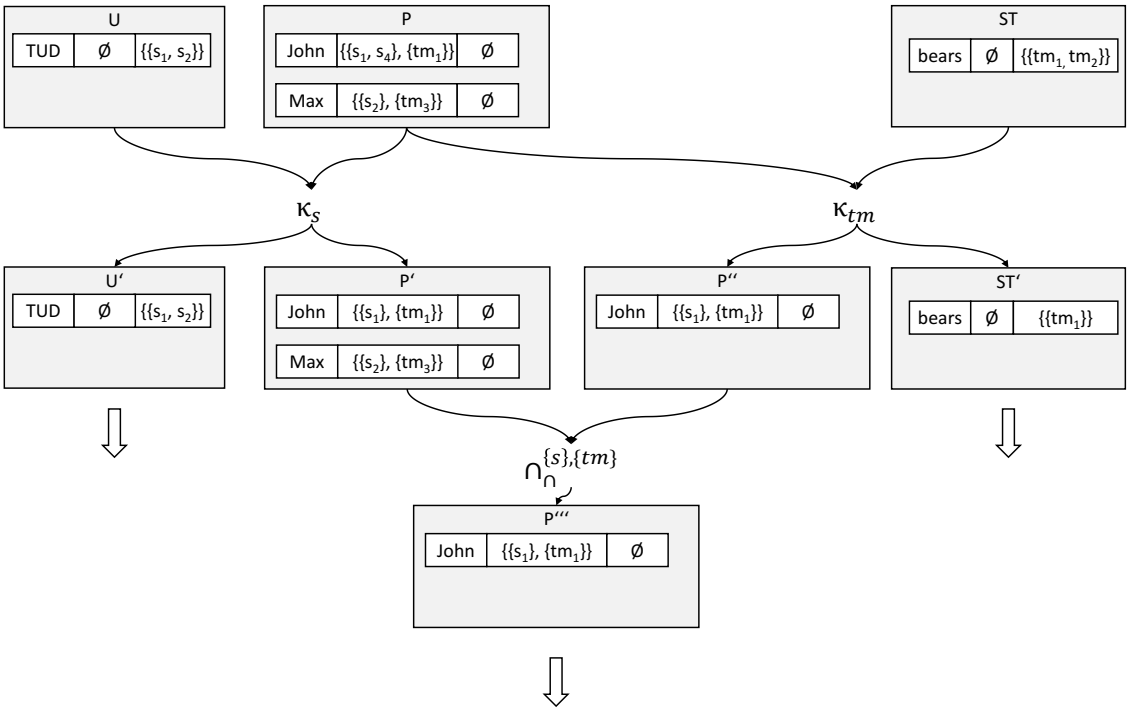


Figure 5.19: Unreflected Dynamic Tuple Elimination

There are three inputs, one for each ⟨*config-expression*⟩ and two role matching operators. At first, the role matching operator κ_s is executed producing U' and P'. Moreover, the Dynamic Tuple $John$ is manipulated, precisely, the Role s_4 is eliminated, because there is no **University** that features this Role. Next, the role matching is performed on **Person** and **SportsTeam** by κ_{tm} producing the intermediate results P'' and ST'. The Dynamic Tuple Max is eliminated from P'', because there is no **SportsTeam** in the input set ST that features a **TeamMember** Role of $John$. Thus, the Role tm_2 is deleted from the Dynamic Tuple $bears$. Finally, P' and P'' are intersected in combination with an intersection on the Role level, producing the output P'''. This set contains only one Dynamic Tuple, $John$, because Max does not pass the intersection operator. However, the query plan is fully completed, but the elimination of Max is not reflected in the **University** output set. As it can be seen, the Role s_2 is still part of TUD, although the corresponding Dynamic Tuple Max is not part of the **Person's** output.

To overcome these invalid results, two options exist. The first option implements multiple operator executions during the query processing. Precisely, in case a Dynamic Tuple is manipulated all previously executed operators have to be validated with the new updated sets of Dynamic Tuples. The

second option requires a different set of operators to unite the Dynamic Tuple streams in a way the relational algebra does it with tuples of different relations.

5.5.2 Multiple Operator Executions

As mentioned, the first option employs multiple operator executions to retrieve the correct result. This option preserves the Dynamic Tuple notion as defined and encapsulates one entity within one Dynamic Tuple. Generally, there are two options to realize multiple operator executions during runtime. At first, each operator that manipulates a Dynamic Tuple features a feedback loop to previous operators. Thus, the manipulating operator's output is directly validated by the previous operators by re-executing them with the new set of Dynamic Tuples. This process cascades the operator plan until there is not any manipulation anymore. The second option executes the operator plan as long as there are changes in the output sets. The output of each run is going to be the input of the next run, as long as there are changes in the results. Hence, this option terminates in case input and output sets are equal.

Feedback Loop

For the first option, imagine the query execution plan shown in Figure 5.20, in which the solid arrows represent the ordinary query execution and the dashed ones the feedback loops. This is the same RSQL operator plan as depicted in Figure 5.18 on the left-hand side, but enriched with a feedback loop from Dynamic Tuple manipulating operators to previous ones. Moreover, assume the inputs as depicted in Figure 5.19. Additionally, imagine the intersection operator is currently performed and eliminates a Dynamic Tuple. Based on this manipulation, both upstream operators are re-executed with the output of the intersection. Consequently, κ_s is executed on the sets U and P'''. As there is no Dynamic Tuple Max, the Role s_2 is eliminated. The results U' and P' are overwritten with the validated output sets. Furthermore, the operator κ_{tm} does no produce any changes in ST' and P''. Next, the intersection is executed once more and does not eliminate any Dynamic Tuple, thus, there is no manipulation and the query result is correctly computed. Note, in case there are more complex interrelations between the input sets, each manipulation and re-execution may issue a validation on the previous operators.

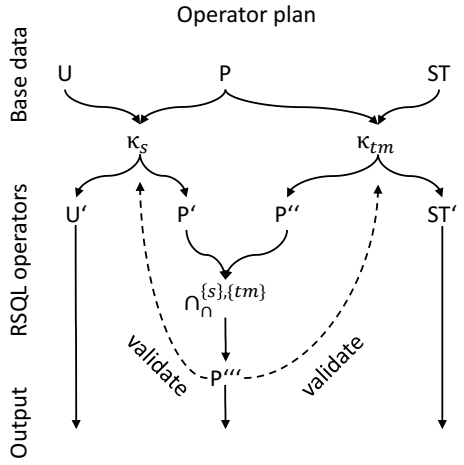


Figure 5.20: Feedback Loop for Manipulating Operators

Multiple Operator Plan Executions

The second option to re-execute the operators, is to executing the whole operator plan several times. This option does not require individual implementations of validity checks for each operator, rather the whole output is checked at once. Generally, the global query execution plan generation is equivalent to the feedback loop mode. Instead of creating local feedback loops for each operator, the single loop is created on the whole plan. This loop is stopped in case the input and output results are equivalent to each other, which means the operators do not change anything and all Dynamic Tuples passes these operators without being manipulated.

The query processing is adapted as shown in Figure 5.21. Note, the input and output details as well as the query execution plan are of minor interest, rather the overall process is focused. Generally, the whole plan execution is embedded in a loop. This loop continues as long as Dynamic Tuples are manipulated during the query plan execution. A query execution run's output represents the input of the consecutive run. As mentioned, the loop is interrupted in case the input equals the execution's output. Then, the query execution terminates and returns the final created output.

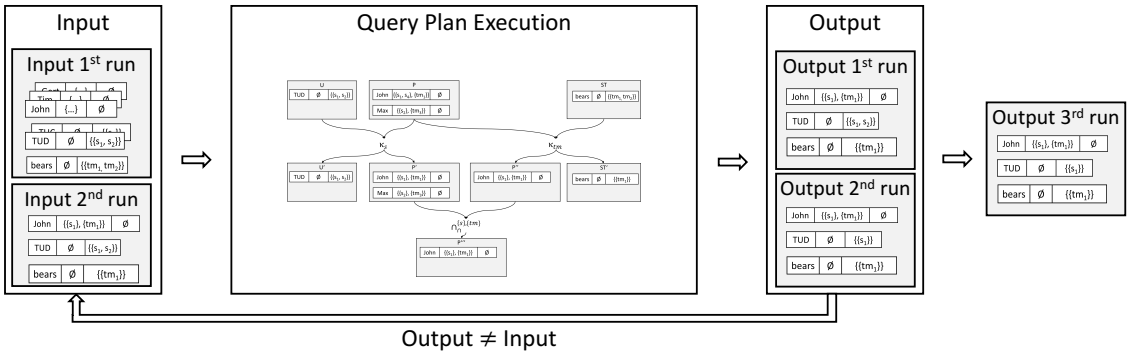


Figure 5.21: Process Overview of a Query Plan Re-execution

To demonstrate the multiple operator plan execution, imagine the example shown in Figure 5.19. After the first run, the University Dynamic Tuple TUD holds too many **Student** Roles, especially s_2 is invalid. However, there are three output sets that differ from the input ones, thus, the query result has to be validated. In detail, the **Person** Dynamic Tuple Max is eliminated by the intersection. Next, the three outputs are put into a second query execution run as input sets. This run will eliminate the s_2 Role in TUD, because there is no partner Dynamic Tuple that plays this Role. All other Dynamic Tuples remain unchanged. However, the inputs and outputs differ in the Dynamic Tuple TUD, thus, the query result is validated once more in a third run. The third run does not change anything in the corresponding sets. Consequently, the query processing computed the correct query result, which is finally sent to the querying application.

5.5.3 Fusing Dynamic Tuple Streams

The last option to avoid invalid intermediate results exploits the relational algebra idea. In detail, the operators fuse matching Dynamic Tuples into a single one. This includes the core as well as the dimension. As result, there will be a Dynamic Tuple that represents the union type of both input Dynamic Tuples, which is similar to the relational join semantics. For each operator that has more than one input set, the union type is built and the passing Dynamic Tuples are united under this type. This eliminates independent Dynamic Tuple streams completely.

However, as result the operators produce Dynamic Tuples that include and mix information of several entities. Moreover, each match is represented as individual Dynamic Tuple. As a consequence, a single entity is distributed over several Dynamic Tuples and mixed with other entity information. To create the several output sets of an RSQL query, the resulting Dynamic Tuple set has to be split into the original Dynamic Tuples. As a single entity is distributed, it cannot be directly extracted from a Dynamic Tuple in the result, rather it has to be built up subsequently.

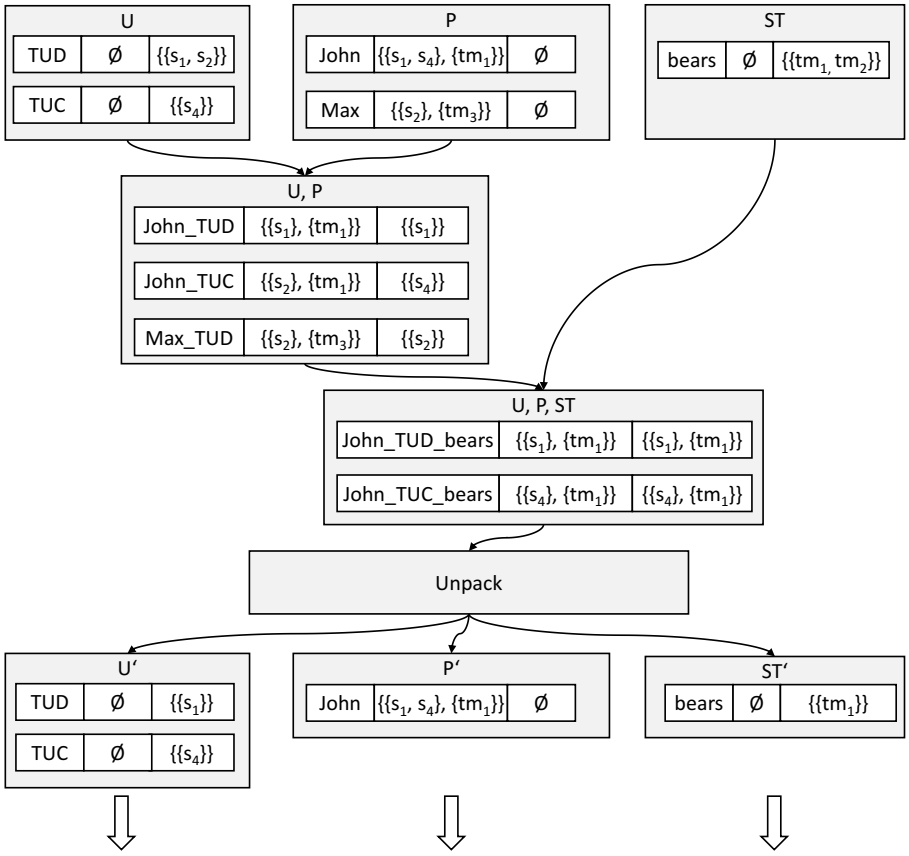


Figure 5.22: Operator Plan for a Dynamic Tuple Fusion Including an Unpack Step

Imagine the example represented in Figure 5.22. It shows the query plan for the query in Section 5.5.1, but with a slightly different input set. Instead of one **University**, there are two. In addition to the first one, there is a **University** that matches the Role s_4 of the Dynamic Tuple $John$. However, the first operator fuses **University** and **Person** Dynamic Tuples in case they share a Student Role. Each match is represented as new Dynamic Tuple comprising both matching ones. In this example, there are three resulting Dynamic Tuples, $John\_TUD$, $John\_TUC$, and Max_TUD

The next operator takes the University and Person (represented in the set University, Person) Dynamic Tuples and fuses these with matching **SportsTeam** ones. This results in new Dynamic Tuples, consisting of a **Person**, a **University** and a **SportsTeam**. Only two Dynamic Tuples are represented in this set, $John\_TUD\_bears$ and $John\_TUC\_bears$. Max_TUD is not included, because there is no fusion partner in the **SportsTeam** set. This represents the query result for now. As it can be seen, the entity information of $John$, for instance, is spread over two Dynamic Tuples in the result and additionally mixed with **University** and **SportsTeam** information.

To bring this fused Dynamic Tuples into the desired output format, an unpacking step is necessary. This step splits each union typed Dynamic Tuple into its atomic entity information. It is an iterative process to rebuild the original Dynamic Tuples, because the entity information is spread over several

Dynamic Tuples in the result. For instance, imagine the Student Role of the entity $John$. These are distributed over two Dynamic Tuples in the result and are required to reunited within the desired format.

However, such Dynamic Tuple processing is against the initial intention, especially to encapsulate all entity information, including the role-based one, within a single Dynamic Tuple. Moreover, it uses the Dynamic Tuple notion in a wrong way. Furthermore, a separate unpacking step is required to rebuild the original Dynamic Tuple structure. Additionally, such processing requires a totally different set of operators, a more relational algebra flavored set of operators. In sum, a fusion of Dynamic Tuples into a union type is contradicting to the Dynamic Tuple's design goals and does not eliminate the entity's information mixing and distribution as observed in the relational data representation.

5.6 RSQL RESULT NET

To preserve the role-based contextual semantics in a query result, we introduce the **R**SQL Res**u**lt **N**et (RuN). It enables users and applications to iterate over Dynamic Tuples and navigate along the Roles to connected Dynamic Tuples. In particular, the navigation leverages the overlapping Roles and relationship information. The query result itself is an instance of the previously defined operative data model. Thus, queries can be nested into each other. Moreover, the traditional view and sub-query mechanism can be applied on this result representation. At first, the general RuN architecture is explained. Based on this, the iteration and navigation paths within a RuN are discussed and detail in several examples. Generally, navigation steps can be performed on the relations between and within Dynamic Tuples, as defined in Section 4.3.2. Finally, a complex RuN example is discussed to provide an integrated and comprehensive overview on the navigation options. Moreover, parts of this result representation have been published in [51]

5.6.1 Architecture

The architecture of RuN is influenced by RSQL's database model and the query language. The database model defines Dynamic Data Types on the type level and Dynamic Tuples on the instance level. A query, especially the ⟨*config-expressions*⟩ specify the schema a Dynamic Tuple has to match for a certain Dynamic Data Type to pass the query operators. Thus, the query processing takes Dynamic Tuples as input, manipulates them with respect to the query, and returns these manipulated ones. Consequently, RSQL's result representation handles Dynamic Tuples and focuses on providing them to the application. However, Dynamic Tuples of different Dynamic Data Types are not merged into a single set, in fact multiple sets of Dynamic Tuples are available. They are organized by following the same structure as defined in the database model (see Section 4.3).

In general, a RuN consists of several result groups, one for each ⟨*config-expression*⟩ in the query. Such a group has a header and a set of Dynamic Tuples. The header defines which rigid type the included Dynamic Tuples have and Role Types in both dimension. This is equal to the Dynamic Data Type definition given in Section 4.3.1. It specifies the overall type of the included Dynamic Tuples int his result group. Moreover, each Dynamic Tuple has a core and sets of Role sets in each dimension, which is identical to the database model definitions in Section 4.3.2.

Furthermore, various Dynamic Tuples usually overlap, but the architecture is focused on the Dynamic Tuple itself. Thus, links between interconnected and overlapping Dynamic Tuples are established to

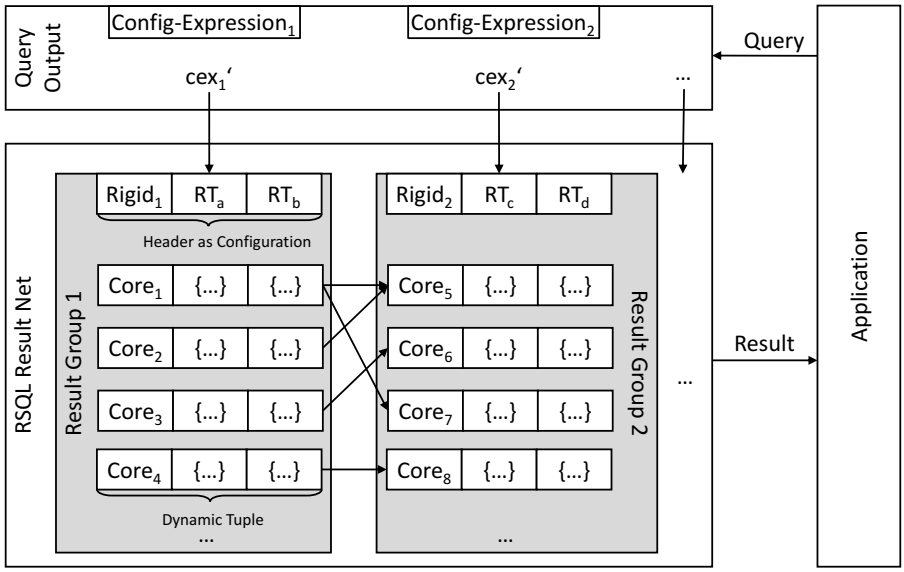


Figure 5.23: RSQL Result Net Architecture

navigate from one Dynamic Tuple to another one by using the overlapping information, for instance. Consequently, a navigation step usually directs from a Dynamic Tuple in a certain result group to another Dynamic Tuple in a different result group. An illustration of RuN's architecture is depicted in Figure 5.23.

In detail, imagine the the following query with overlapping ⟨*config-expressions*⟩.

```
1 SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa,
2   Uni u FEATURING s AND sa;
```

This will produce a query output with **Persons** being **Students** and **StudentAssistants**, and **Universities** featuring these Roles. However, there are two sets of Dynamic Tuples, one for the **Persons** and one for the **Universities**. Consequently, the resulting RuN will have two result groups, one for each of these sets. The header in the **Person** result group will have a **Person** as core and the **Student** and **StudentAssistant** Role Type in the filling dimension. In contrast, the second result group has the **University** as core and both Role Types in the participating dimension. Each of the query's output sets is assigned to its designated result group.

In sum, this architecture layout preserves the Dynamic Tuple notion, including the metatype distinction and metamodel semantics, within the result representation. Dynamic Tuples are gathered together under a certain type and linked to other Dynamic Tuples resulting in multiple available and interconnected sets of Dynamic Tuples.

5.6.2 Iteration and Navigation

As aforementioned, an RSQL result net consists of interconnected and overlapping sets of Dynamic Tuples [51]. These interconnections and overlapping information are utilized to enable navigation between several Dynamic Tuples. Additionally, the connected Dynamic Tuples may not be of the same result group, in fact usually they are of different ones. Generally, RuN offers two ways to options to navigate within the net of Dynamic Tuples. Firstly, there are endogenous navigation paths and secondly exogenous ones. The former provide access to Dynamic Tuple internal information, like

Roles in a certain set of Roles. The latter gives access to connected Dynamic Tuples by using the overlapping information or relationships.

However, the traditional and well-known principle of iterating over records in a query result is not touched. For instance, a query consisting of a single ⟨*config-expression*⟩ only returns a list of Dynamic Tuples. To jump from one Dynamic Tuple to another in this list, the well-known iteration principle is used. Furthermore, an overview of RuN's general functionality is presented in Table 5.3. A DT in the functionality overview refers to a set of Dynamic Tuples, whereas a dt only to a single one. Likewise, a R relates to a set of Roles and a r to a single Role. Moreover, a T is associated with either a set of Dynamic Tuple, a set of Role sets, or a set of Roles. This polymorphy is used for iterations purposes and omits specifying the same functionality multiple times. All navigation possibilities as well as the iteration are explained in detail in the following.

	Functionality	*Input*	*Output*
general	execute	arbitrary query	new Cursor(DT)
	switch	$rigid$	new Cursor(DT)
	next	Cursor(T)	Cursor(T) + 1
	close	Cursor(T)	-
endogenous	plays	dt	new Cursor($\{R\}$)
		dt, rt	new Cursor(R)
	features	dt	new Cursor($\{R\}$)
		dt, rt	new Cursor(R)
exogenous	played by	r	dt
	featured by	r	dt
	related to	r, rst	new Cursor(R)

Table 5.3: Functionality Overview of RSQL's Result Net

Iteration

The iteration is the most basic way to browse through a RuN; it jumps from one element in a certain list to another one. Generally, each navigation action in a RuN issues cursor operations and enables applications to iterate over certain data structures [51]. The initial cursor on a RuN is generated when accessing the result net the first time. It initially points to the firstly returned Dynamic Tuple of the first result group, which holds Dynamic Tuples of the query's first ⟨*config-expression*⟩. Hence, an arbitrary query generates a new cursor on a certain Dynamic Tuple. In addition to iterating Dynamic Tuples, Roles may be iterated, especially when a Dynamic Tuple's internal information is accessed. Moreover, the initially created cursor on the first result group can be switched between the result group by invoking the **switch** functionality in combination with a rigid name. This overwrites the old main cursors and places the new one on the first Dynamic Tuple of the targeted result group. The corresponding result group is identified by its rigid name in the header.

However, a cursor can be moved by utilizing the **next** functionality. It moves the cursor from its current position in a set to the next one. This functionality is not limited to Dynamic Tuples, because Roles may also be iterated. Thus, this functionality consumes a cursor on either a certain Dynamic Tuple, a set of Roles, or a Role, which is denoted as T in the functionality overview. However, there may exist several cursors on the same set. Each cursor is handled individually and in isolation, thus, terminating a cursor on a certain set does not affect all other cursors that may exist on the same set. Moreover, a cursor can be **closed** in case the iteration process has to be terminated. As example for a Dynamic Tuple iteration imagine the illustrations depicted in Figure 5.24 and Figure 5.25.

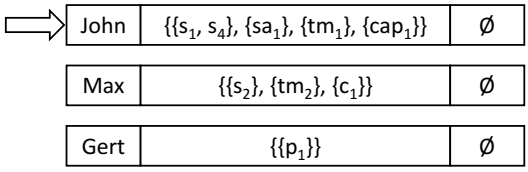


Figure 5.24: Initial Cursor Position at t_0

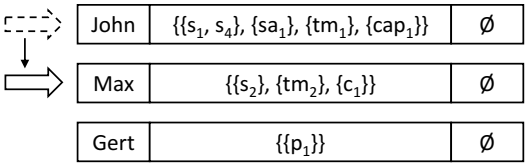


Figure 5.25: Moved Cursor Position at t_1

There are three Dynamic Tuples, *John*, *Max*, and *Gert*. Initially at point t_0, the cursor points to *John*. Such situation may appear in case the RuN is accessed the first time. After executing a next on this cursor, it is moved to *Max* as shown at point t_1.

However, the same functionality can be applied on sets of Roles, like the **Student** set of the Dynamic Tuple *John*, which consists of the Roles s_1 and s_4. A **next** moves the cursor from one Role to another one, but only within the available **Student** Roles and never outside of its scope. Hence, the Role sa_1 is not reachable by the **Student** Roles' cursor. This functionality is well-known in database systems, thus, a more detailed explanation is skipped.

Endogenous Navigation

A Dynamic Tuple by definition is a combination of a rigid type, the sets of played Roles, and sets of featured Roles. Hence, the information carried by the Roles are embedded in the two dimensions. To access this information special endogenous navigation paths are available on the basis of the result functionality presented in Table 5.3.

In detail, there exist the **plays** and features **functionality**. Both are based on the endogenous data model relations defined in Section 4.3.2 on page 81 and the work in [51]. As the name of the functionality implies, the former one provides access to Roles in the playing dimension and the latter to those in the featuring dimension. Hence, they only differ by their designated dimension.

In general, each of this Dynamic Tuple's internal navigation paths can be used with two different parameter sets. The first option consumes the Dynamic Tuple on which plays or features is performed on only. It returns a new cursor on the Role sets in the corresponding dimension. The second option additionally receives a Role Type. This provides direct access to Role of a certain Role Type in a particular dimension by creating a new cursor on the corresponding Roles set.

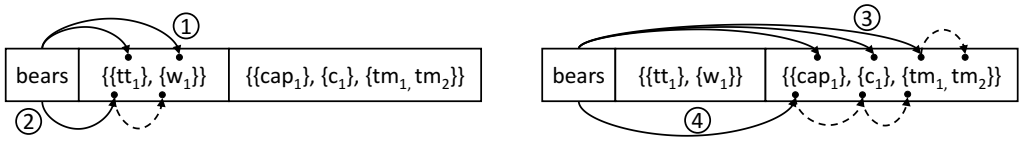


Figure 5.26: Endogenous Navigation Options. On the Left-hand Side for the Playing Dimension and the Right-hand Side for the Featuring Dimension

For instance, imagine the Dynamic Tuple included in an arbitrary RuN as depicted in Figure 5.26. Both sides represent the Dynamic Tuple *bears* as shown in Figure 2.4 with Roles in both, the playing and featuring dimension. The arrows represent all available endogenous navigation paths. The former for the playing dimension and the latter shows the options for the featuring dimension. There exist two kinds of arrows; the solid arrows mark direct navigation points and dashed ones iteration options.

Consequently, the left-hand side represents the options within the playing dimension and the right-hand side within the featuring dimension.

In detail, the encircled 1 on the left-hand side in Figure 5.26 points from the Dynamic Tuple to the Role w_1. This navigation can be performed by executing *plays*($bears$, WinnerTeam). There are no additional iteration options available, because only one Role of this Role Type is currently played. The encircled 2 navigates from the Dynamic Tuple to Role sets, in this scenario to the set of **Tournament-Team** Roles. Because there are two sets of Roles in this Dynamic Tuple's playing dimension, there exists an opportunity to iterate, especially to the set of **WinnerTeam** Roles. To access these Role sets, the *plays* functionality with the parameter $bears$ has to be called.

The right-hand illustration in Figure 5.26 shows all navigation opportunities within the featuring dimension. Precisely, the encircled 3 gives access to all **TeamMember** Roles, in particular to the Roles tm_1 and tm_2. Since there are two Roles available in this set, an iteration options exists on the created cursor. Assuming the initial cursor points to tm_1 a move forward would set it on the Role tm_2. To create this particular cursor, the functionality features has to be executed as features($bears$, TeamMember). Moreover, the encircled 4 describes the navigation to the sets of Roles, to manually extract the Roles, for instance. Initially the created cursor points to the set containing Roles of the Role Type **Captain**. An iteration would move this cursor toward the **Coach** and **TeamMember** sets.

Exogenous Navigation

In contrast to the endogenous navigation paths, the exogenous provide access to related Dynamic Tuple by using the overlapping or relationship information [51]. This enables the application to navigate from one Dynamic Tuple to another one while processing the result and, thus, access the information in which Compartment a certain Role is featured in or by which Dynamic Tuple a Role is played by, for example. Generally, there are three options available for the exogenous navigation, each based on the exogenous relations defined in the data model (see Section 4.3.2).

At first, the **played by** functionality provides navigation from a certain Role to the Dynamic Tuple this particular Role is played by. In particular, this functionality leverages the overlapping information to direct from a Role in the featuring dimension to a Dynamic Tuple that holds the same Role in the playing dimension. It consumes a certain Role r and returns a concrete Dynamic Tuple. There is no cursor, because a particular Role can have one player only, thus, the Dynamic Tuple is returned instead of a cursor.

Secondly, the **featured by** functionality is the inverse to the played by one. Consequently, it directs from a certain Role to the Compartment this Role is featured in. In detail, the overlapping information is used to navigate from a Role in a Dynamic Tuple's playing dimension the one that holds the same Role in the featuring dimension. It takes a certain Role as input and provides access to a Dynamic Tuple. As a Role is featured in only one Compartment, this functionality does not use a cursor, too.

Finally, the **related to** functionality takes the relationship information to navigate from a certain Role to the opposite Roles. A Role may have multiple partner Roles in a certain Relationship, hence, the related to functionality provides a cursor on all related roles. To return the correct partner Roles, this functionality consumes a certain Role and the Relationship Type this Role has to be connected by. The Compartment this relationship takes place is uniquely identifiable by the Role, thus, there will not be any ambiguity about this.

As navigation example, imagine a situation as depicted in Figure 5.27. There are three Dynamic Tuples, $John$, $Gert$, and TUD. All have Roles in certain dimensions. The directed arrows point

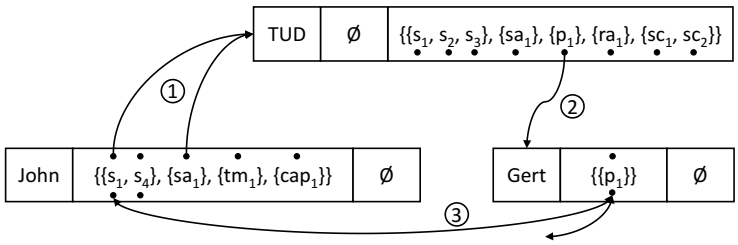


Figure 5.27: Exogenous Navigation Options Between Several Interconnected Dynamic Tuples

from a certain Role to the opponent, either a Dynamic Tuple or a Role. Generally, each black dot under or over a Role indicates a navigation option. For the sake of clarity only the discussed and explained arrows are drawn in the illustration.

Firstly, assume a cursor pointing to the Role s_1 in the Dynamic Tuple $John$ as indicated by the encircled 1. To get the information in which Compartment this Role is featured in, the **featured by** functionality is used. As it can be seen, the Role is located in the playing dimension of $John$ and in the featuring dimension of TUD. Hence, the **featured by** functionality called on $\sim_1$ and returns TUD. The same holds for the **StudentAssistant** Role sa_1 of $John$.

To demonstrate the **played by** functionality, imagine a cursor pointing to p_1 in the Dynamic Tuple TUD, as indicated by the encircled 2 in Figure 5.27. This is used in case player information about a Role featured in a certain Compartment are required. In general, this functionality is available for Roles situated in a featuring dimension of a particular Dynamic Tuple and in another's playing dimension. The **played by** functionality consumes a certain Role and returns the Dynamic Tuple this particular Role is located in the playing dimension. Thus, calling **played by** on p_1 in TUD returns $Gert$ as opponent.

Finally, the **related to** functionality can be used, especially in case relationship information has to be utilized for navigation. Imagine a cursor pointing at p_1 in the Dynamic Tuple $Gert$. Applying the **related to** functionality in conjunction with the supervises Relationship Type, as shown by the encircled 3, navigates to the s_1 Role played by $John$. However, a **related to** works for Roles in the playing dimension only, because in a featuring dimension listed Roles cannot be related to each other by definition. Moreover, the **related to** functionality is bi-directional, which means it works in both directions. However, this functionality returns a new cursor on a list of Roles, which can be iterated. These Roles will not be part of the same Dynamic Tuple, but additional information about the corresponding Dynamic Tuple can be collected by using the endogenous navigation paths. This characteristic is indicated by the second arrow connected to p_1 ind $Gert$.

5.6.3 Example Navigation

As complex example imagine the RuN returned by the following query and depicted in Figure 5.28.

```
1 SELECT * FROM Person p PLAYING (Student s OR StudentAssistant sa) AND TeamMember tm,
2   University u FEATURING s OR sa;
```

This query returns a RuN consisting of two result groups, one for each ⟨*config-expression*⟩. In this scenario, the first result group contains Dynamic Tuples with a Person core. Moreover, the initial cursor is set to $John$ as first Dynamic Tuple of the firstly specified ⟨*config-expression*⟩. The second result group comprises Dynamic Tuple with a University core. Each of these groups has a header,

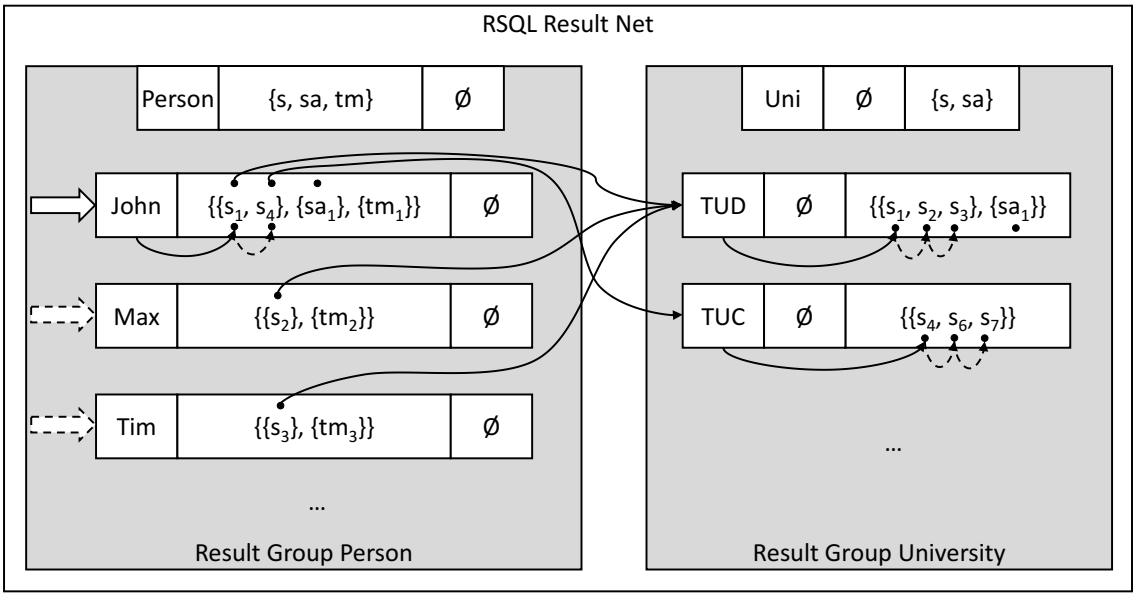


Figure 5.28: Navigation Example for a Small RSQL Result Net

specifying the Roles of which Role Type might be included. In detail, not all Role Types defined in the header must be filled with, especially in case of disjunctive Role Types in the ⟨*config-expression*⟩. A navigation to empty set of Roles returns an empty cursor. However, the header in the first result group defines the Role Types **Student**, **StudentAssistant**, and **TeamMember** in the filling dimension while the participating dimension is empty. The second result group header specifies an empty filling dimension and the Role Types **Student** and **StudentAssistant** in the participating dimension. These headers additionally define the inputs for the endogenous navigation paths. Role Type that are not included in the header cannot be used as navigation input.

Additionally, the arrows represent navigation paths, but not all navigation paths are illustrated in the figure. More precisely, the solid arrows are endogenous and exogenous navigation paths and the dashed ones represent iteration options. Moreover, the black dots indicate a possible navigation path. Roles that do not have a black dot, cannot be used to navigate to another Dynamic Tuple or Role, like the TeamMember Roles tm_1, tm_2, and tm_3. The big white arrow with the black borders on the left-hand side illustrates the main cursor and its iteration steps.

At first, the initial cursor points to $John$, as already mentioned. Executing a **plays**($John$, $Student$) results in new cursor within the set of **Student** Roles, initially pointing at s_1. By using the traditional *get* operations, known from a relational result sets, enable access to this Role's attributes, like $Student\_ID$ or $Studies$. Next, the featured by function is invoked on this Role resulting in navigating to the TUD Dynamic Tuple in the second result group. All endogenous and exogenous navigation paths are available for this Dynamic Tuple, too. Thus, an endogenous navigation by **features**(TUD, $Student$) provides access to all **Student** Roles within this university and creates a new cursor initially pointing to s_1. A **next** on this cursor moves it forward to the Role s_2 and provides access to all attributes it holds. Furthermore, a **close** on this Student Role pointer finishes the iteration and jumps back to the Dynamic Tuple TUD.

Now, lets assume the process on TUD is finished and the application handles the cursor on $John$'s Student Roles again. A next on this cursor moves it on the Role s_4. Once again, the university information is required to this Role, hence, a **featuredBy**($John$, s_4) is executed. This navigation directs to the Dynamic Tuple TUC. Again, all endogenous and exogenous navigation options are available.

After processing this Dynamic Tuple and collect the necessary information the cursors are closed, except of the main cursor, which still points at $John$.

Finally, a **next** is performed on the main cursor moving it to Max. A **plays** in conjunction with the **Student** Role Type on this Dynamic Tuple creates a new cursor on the Role s_2. From this Role the TUD is reachable by the featured by functionality. As this navigation options was already discussed, a detail discussion is skipped at this point. Finally, after processing Max the main cursor is moved to the Dynamic Tuple Tim, which is processed next. This iteration and navigation process can be continued until the last Dynamic Tuple of the result group, in which the main cursor is moved within, is finally processed.

5.7 SUMMARY

Each database system need a sophisticated external interface to communicate with database in general. Usually, such an interface provides a language to create schema objects, a language to create the database instance and manipulate the data, and a certain language to retrieve data. In case of RSQL a compound language consisting of data definition, data manipulation and data query language was introduced. All these language parts distinguish between RSQL's metatypes, thus, the separation of concerns is preserved within the query language.

Requirement	ConQuer	IQL	RSQL
Metatype distinction	☐	⊞	■
Data definition language	☐	⊞	■
Data manipulation language	☐	☐	■
Data query language	■	■	■
Syntax description	☐	■	■
Result representation	☐	☐	■

■: yes, ⊞: partial, ☐: no

Table 5.4: Evaluation of RSQL's Query Language and Processing in Contrast to Related Approaches

However, for each language part we presented a concrete syntax and demonstrated this syntax by utilizing several examples. In particular, the data definition language's syntax is introduced by creating a small database schema and associate the statements with the underlying database model and Dynamic Data Types. Moreover, the data manipulation language's syntax is demonstrated by creating a small database instance, featuring three Dynamic Tuples. Additionally, a step by step creation illustrated the evolution of Dynamic Tuples and their overlap. As last language part and formal syntax description, the actual data query language is presented and related to the formal database operators. As connection between the database model and the query language, a discussion on processing RSQL queries on the basis of Dynamic Tuples, including blocking factors and their solution, took place. Especially, the occurrence of invalid Dynamic Tuple states is explained. Finally, we illustrated the result representation of RSQL queries as RSQL Result Net by giving an architectural overview and presented iteration and navigation options. A detailed comparison of RSQL's query language, query processing, and result representation in contrast to the related approaches is presented in Table 5.4.

In sum, the external database interface, as introduced in this section, preserves the metatype distinction for the data definition, manipulation, querying, and result representation part. Moreover, the formal syntax is connected to the database operators and the underlying database model. Additionally, a result representation, built upon the basis of the database model and augmented with navigation paths, completes the database interface design. Hence, RSQL's query language fulfills all requirements posed to an external role-based database interface.

PROOF OF CONCEPT

RSQL is designed to directly represent role-based data in a database system to overcome the role-relational mismatch. This mismatch causes pains for the database systems themselves and the application developers. To evaluate RSQL's benefits, it is compared to a relational representation of role-based data. In detail, the evaluation focuses on the three RSQL parts, the database model, the query language, and the result representation. Moreover, we compare both database models by using the university scenario as presented in the Sections 2.4.1 and 2.4.2. First, we describe the evaluation setup by outlining RSQL's prototypical implementation and explaining the applied relational mapping schema. Moreover, we map the complete university scenario as relational baseline. On this mapping basis, the effort in representing the university schema in RSQL and SQL is compared to each other by contrasting the number of statements required to set up the schema. Next, the query languages are compared to each other by utilizing various example queries that represent different scenarios. As final evaluation, the result representation comparison takes place. This evaluation is based on the client-side effort to process both result representations. Finally, this chapter is summarized and concluded.

6.1 EVALUATION SETUP

First, we assume the database system to be used within an ecosystem as described in Section 3.1. In detail, the database system is the single point of truth with a diverse software system, consisting of multiple applications and possibly application servers. Moreover, the concrete example employed to compare a relational database with an RSQL database is based on the university scenario as presented in the Sections 2.4.1 and 2.4.2.

6.1.1 RSQL Prototypical Implementation

We implemented a prototypical version of the RSQL approach by adapting a relational database system. As this approach aims at an adapted data system and set-oriented interface, relying on an already existing system simplifies the implementation. In detail, this prototype is based on the H2 database engine[1] in version 1.3.176, which is the latest stable version (accessed January 2017). This engine is an open source, lightweight, and Java-based relational database management system. To describe our system adaptations, we firstly outline the implementation of the database model by showing adapted and additionally introduced classes. Secondly, the query processing and result generation is detailed. Finally, the client side adaptations with respect to the changed Java Database Connectivity[2] (JDBC) interface are outlined.

Database Model

As aforementioned, the prototype relies on a persistent storage engine of H2 and only the data system is changed. To achieve a metatype distinction, we introduce subtypes of the abstract class *Table* (h2.table.Table). These class store the metatype specific information, like the player type of a Role Type and its corresponding Compartment Type. Hence, RSQL's types are represented as specialized tables. This enables to rely on an existing storage system, such that an insert in a specialized table uses the same functionality as an insert in a regular table.

[1] `http://www.h2database.com/`

[2] `http://docs.oracle.com/javase/tutorial/jdbc/TOC.html`

However, we do not use these table structures as main data structure and do not access them directly, in fact they only help to add additional functionality atop the standard table functionality. Rather, the main organizational data structure is the Dynamic Data Type (h2.table.rsql.DynamicDataType) as shown in Figure 6.1. Each Dynamic Data Type holds a set of non-empty Configurations and each of these holds a set of Dynamic Tuples. Dynamic Tuples as instance representation of Dynamic Data Types hold a player consisting of a rigid type and the actual row data. Additionally, Roles (h2.result.rsql.RoleInstance) are grouped into two dimensions, the playing and featuring dimension. Moreover, each of these *RoleInstance* objects has a certain Role Type, the actual data represented as row, and additionally references to its player and featurer Dynamic Tuple. These references enable the exogenous navigation paths in the result representation. Moreover, the references on the actual types, which are the specialized tables, ensure to build the RSQL role-based semantics upon the underlying database system layers. In addition, these Dynamic Data Type and Dynamic Tuple representation ensures to encapsulate the role-based semantics in these data structures.

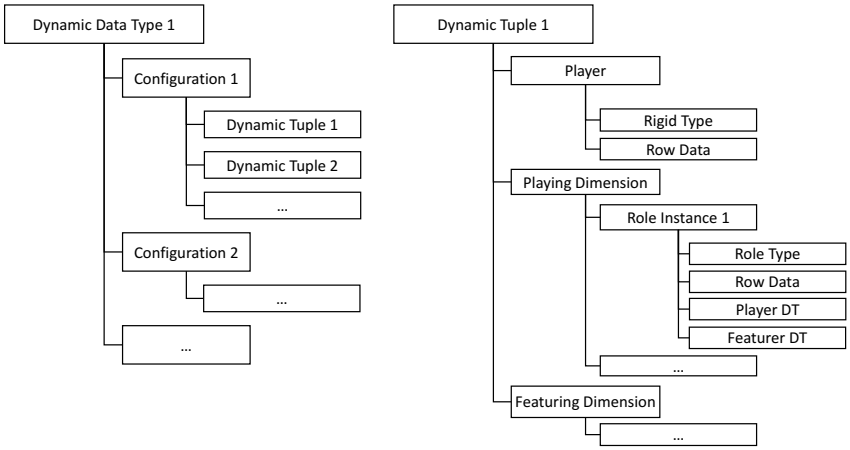


Figure 6.1: RSQL's Database Model Structure

Query Processing

The query processing prototypically implements the database model operators. Generally, the parser creates statements, like a select statement or an insert statement. Such a statement holds all information necessary to execute it. The main data structure for a statement on the basis of Dynamic Tuples is the class *DynamicDataTypeFilter* (h2.table.rsql.DynamicDataDypeFilter), which represents the Dynamic Data Type in queries. It is structured like the storage structure, but features additional functionality to filter Dynamic Tuples, for instance on the basis of the Configuration or a selection predicate. Thus, each statement features at least one of these Dynamic Data Type filters. For instance, an ⟨*insert-rt*⟩ statement on a Role Type will always feature two of these filters, one to specify the player Dynamic Tuple, and one to define the featuring Dynamic Tuple. Moreover, a ⟨*select*⟩ statement will feature as many of such filters as there are ⟨*config-expressions*⟩.

All the Dynamic Tuple query processing is encoded in these filters, as conceptually visualized in Figure 6.2. A Dynamic Data Type filter is set up a certain Dynamic Data Type and a certain Statement. It features cursors to iterate the Dynamic Tuples. For each Dynamic Tuple the manipulations and filters are applied on a copy of it, except for DML statements, these are executed on the actual base data. In case this copy passes the operators, it is added to the RSQL result (h2.result.RsqlResult). When the evaluation of all Dynamic Tuples is finished, the result is returned to the statement. There, they might be additionally manipulated by other operators or returned to the client as query result. However, this processing preserves the role-based semantics throughout the entire query processing as well as in the result.

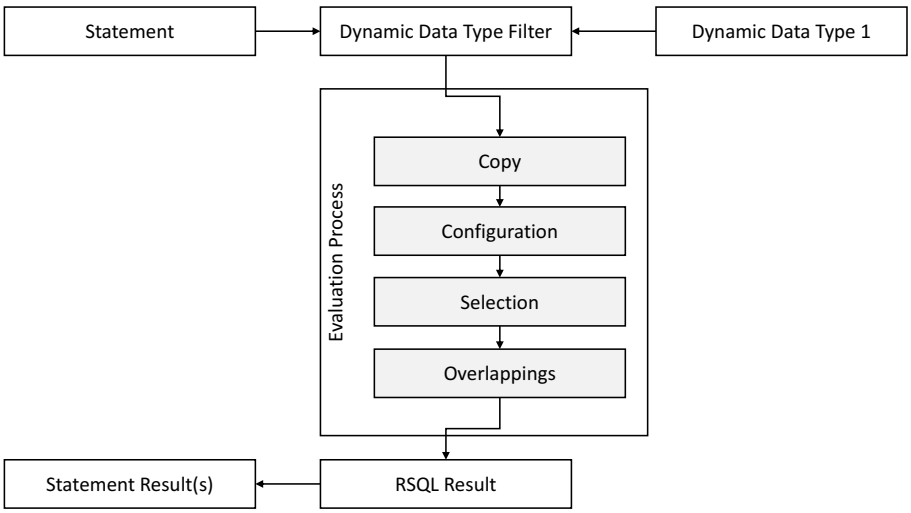


Figure 6.2: RSQL Query Processing on Dynamic Tuples by Dynamic Data Type Filters

Client-Side Support

As RSQL is based on a different data structure than the traditional relational data model, we adapted H2's Java Database Connectivity[3] (JDBC) implementation. To avoid a complete JDBC interface redefinition, we added RSQL client side functionalities in separate subclasses. Hence, an explicit type cast to these classes is necessary from the clients. A conceptual overview of the RuN implementation is presented in Figure 6.3. In detail, the class *JdbcRsqlResultSet* (h2.jdbc.rsql.JdbcRsqlResultSet) is a *JdbcResultSet* class (h2.jdbc.JdbcResultSet) extended by RSQL functionalities, like multiple results (h2.result.rsql.RsqlResult). A short JDBC introduction from a database application developer's perspective is given in [71, pp. 421–426]. This *RsqlResult* class represents a result group as defined in Section 5.6.1.

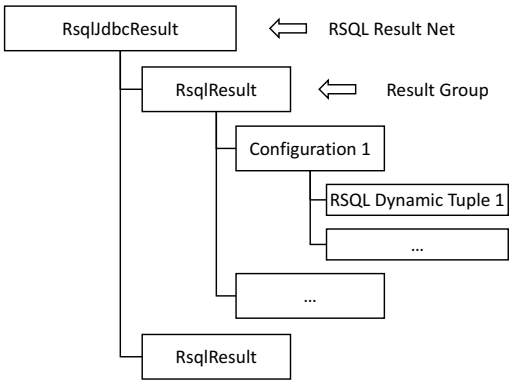


Figure 6.3: Conceptual Overview of RSQL's Result Representation Implementation

Moreover, such a result group consists of a client-side Dynamic Tuples representation (h2.result.rsql.RsqlDynamicTuple) grouped by their Configurations. It features a cursor to iterate the Dynamic Tuples and functionalities to use the endogenous navigation paths. In particular, this navigation paths are realized by the method *getRoleInstances(RsqlDimensionTuple dim, String roleType)*. A Role, represented by the class *RsqlRoleInstance* (h2.result.rsql.RoleInstance), holds the actual Role data, as well as references to it playing and featuring Dynamic Tuple. These references enable the

[3] `http://docs.oracle.com/javase/tutorial/jdbc/TOC.html`

exogenous navigation. In contrast to the base data representation of Dynamic Tuples, this result representation is not required to reference both, the playing and featuring Dynamic Tuple. In fact, only one has to be set. This is a direct effect of the operational data model, as described in Section 4.4.1.

However, the traditional *getter* and *setter* methods, to collect attribute values, of the JDBC interface are available on the Dynamic Tuple and the Roles. In case they are execute on the Dynamic Tuple, the core information is returned. In case of a Role, the Role data is accessed.

Additionally, we adapt the graphical user interface to be compliant to RSQL and visualize the metatype distinction. In detail, H2 features a console server that is accessed by the web browser[4]. To provide a visual distinction in combination with the interrelations, we added different symbols, as illustrated in Figure 6.4. A Natural Type is represented by a person icon, a Compartment Type by circles to support the collection perception of Compartment Types, a Role Type by two theatrical faces, and Relationship Type by three connected instances. Moreover, Role Types show information of the players and in which Compartment Type they are contained in. A Relationship Type directly visualizes its participants. The rest of the graphical user interface remains unchanged.

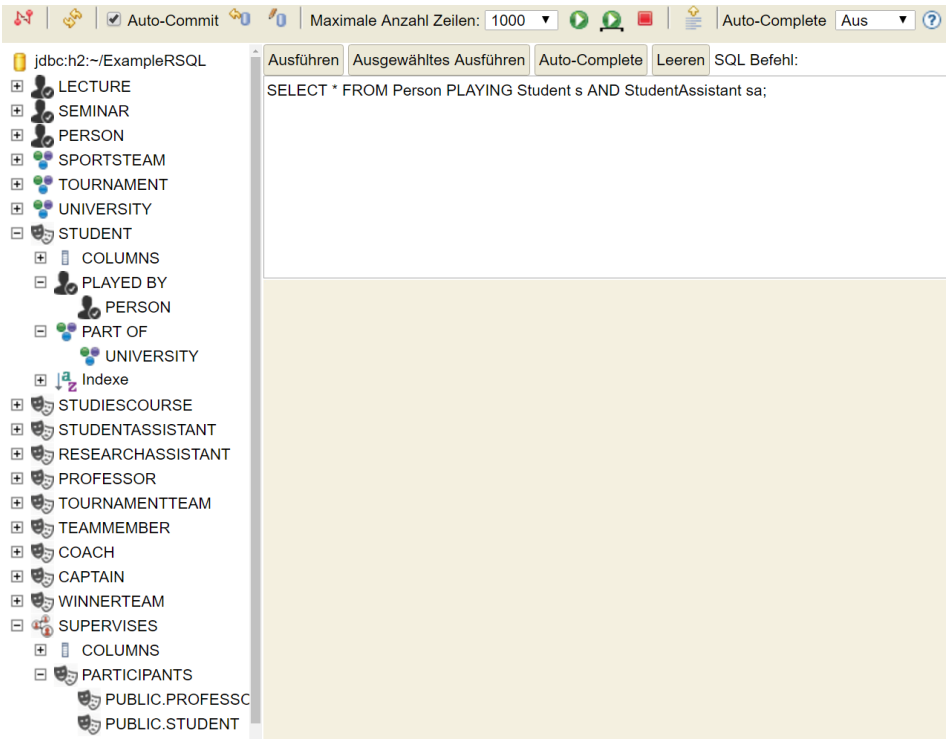


Figure 6.4: Graphical User Interface Adapted to the Metatype Distinction

6.1.2 Relational Mapping RSQL's Database Model

The RSQL database model can be represented in the relational data model. In detail, the structural mapping is explained first. Afterwards, the constraint mapping of the metamodel axioms is defined. These mappings build the foundations on which the RSQL approach is compared to its relational realization. Moreover, as prerequisite we assume that the applications provide the database system a standard relational mapping and the database system is only in charge of ensuring instance consistency with respect to the given schema. Thus, it is assumed, that the relational schema represents a valid instance of the metamodel.

[4] `http://www.h2database.com/html/quickstart.html`

Mapping RSQL Structures

At first, we discuss the structural transformation from RSQL's schema objects onto relations. To have a fair comparison the relational mapping is performed on the rules of good database design on the basis of mapping entities onto relations [54, p. 71–83]. Recap, the database model on the schema level is a tuple consisting of $\mathcal{S} = (NT, RT, CT, RST, \mathit{fills}, \mathit{parts}, \mathit{rel}, \mathit{card})$. It distinguishes four metatypes, Natural Types, Compartment Types, Role Types, and Relationship Types. Each of these type of these metatypes is represented in an individual table that is named as the type itself. In a pure automated mapping process, annotations that reference the tables belonging to a certain metatype support the mapping process, because these annotations only help users to understand the semantics of the tables. Thus, such annotations are omitted. Moreover, each type table receives an ID column as surrogate, which is used to reference the corresponding table's tuples by a single column, instead of using the composite primary key. This ID is not part of the primary key, but required to be unique. Next, the metatype interrelations are mapped.

fills The fills relation is surjective, but not bijective. Hence, the cardinalities of this relation are $N : M$. The surjectivity is ensured by the applications, thus, the database schema needs to represent the relationship only. Such cardinalities are usually mapped by relating the corresponding entities using an association table that references these entities as foreign non-nullable key constraints. Consequently, for each entry in *fills* an individual association table is created that connects the player type with the Role Type. In detail, these association tables have two columns only, one to reference the ID of the player, denoted as P_ID, and one for the referenced Role, named as R_ID.

parts This function stores information about the containment of Role Types in a certain Compartment Type. Each Role Type is part of exactly one Compartment Type, thus, the mapping adds the a Compartment Type reference to each Role Type table. This column is named C_ID and is defined as referential constraint.

rel This function stores the information on participating Role Types in a certain Relationship Type. Depending on the cardinalities of a relationship, there are separate mapping options. Usually, a $N : M$ relationship is mapped to an association table, a $1 : N$ to that part, which represents the *1* part, and a $1 : 1$ combines both entities and the relationship within one table.

card The *card* function stores the information on the concrete cardinalities for a certain Relationship Type. For each lower cardinality larger than 1 a trigger has to be implemented that checks the correct amount of occurrences. The same applies for limited upper cardinalities.

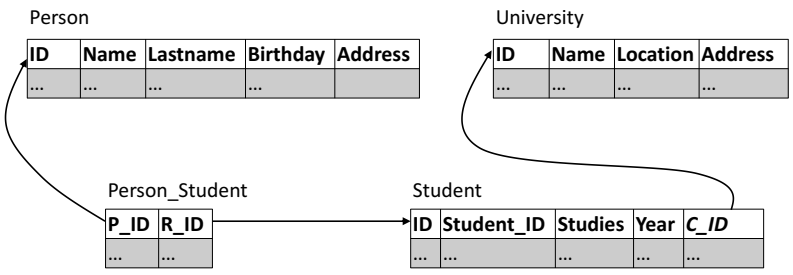


Figure 6.5: Example Mapping of Person, Student, and University

As example, imagine the mapping depicted in Figure 6.5. This illustrates the schematic mapping of the **Person** Natural Type, the **Student** Role Type, and **University** Compartment Type. Each of these types

is modeled in an individual table. Additional an association table **Person_Student** is introduced to model the relation between the Person and Student. Moreover, the Role Type table **Student** features an additional column to reference the **University**.

The instance structure definitions of RSQL's database model is mapped as well. Each instance of a type is represented as tuple in the corresponding table. The instance model relations and functions are mapped as following.

type The belonging of an instance to a certain type is implicitly represented by a tuple's belonging to a certain table.

plays This relation stores information which entity plays which Role in which Compartment. As the instances are stored as tuple in individual tables, this relation is distributed over several tables. In particular, this includes tuples in the Natural Type tables, the Role Type tables, the Compartment Type tables, and the association tables between the player types and Role Types.

links This function stores information which Roles participate in which Relationships. This represented as tuple in the corresponding Relationship Type table.

Mapping RSQL Constraints

These structure mapping rules ensure a correct transformation of an RSQL schema and the corresponding instances to a relational model. In addition, the instance level constraints have to be ensured by the database system. Note, the schema level constraints are omitted, because we assume the applications to provide a valid schema. In particular, the database system has to ensure the axioms as defined in Definition 4 on page 75.

Correct Role playing The first axiom (4.6) ensures Roles to be played by only valid cores and in the correct Compartment. This is automatically ensured by the associations tables between the player type and Role Type tables, as well as the foreign key constraint on the Compartment Type table within the Role Type representation.

Uniqueness of player, Compartment, and Role The second axiom (4.7) requires uniqueness of the tuple player, Compartment, and Role, which ensures that a certain Role Type can be instantiated within a particular Compartment only once per player. As the *plays* information is distributed over several tables, a trigger has to be created that checks for this uniqueness. Especially, that a certain Role is referenced only once in all association tables that are related to its Role Type table.

Player uniqueness of a Role A certain Role has exactly one player as defined in the third axiom (4.8). This guarantee is complex, because it possibly involves multiple tables in which the same Role Type may be referenced. As there is no uniqueness constraint over several tables available, this has to be ensured by triggers. In detail, this trigger has to ensure, that a certain Role is referenced exactly once over several tables.

Compartment uniqueness of a Role Each Role is required to be located within a certain Compartment (4.8). This is ensured by the *parts* function's mapping that induces the Compartment Type referential constraint in the Role Type table. Thus, only one Compartment can be referenced by a Role. Moreover, the `NOT NULL` constraint ensures the existence within a Compartment.

Relationships The axioms 4.9 – 4.11 ensure a correct representation of Relationships within the *links* function. This natively guaranteed by the Relationship Type table and do not require any additional effort in mapping.

Cardinality constraints To ensure the correct cardinalities, a trigger has to be implemented for each cardinality constraint. This is also guaranteed by mapping the schema level *card* function onto triggers. Consequently, the axiom 4.12 defines the actual limits implemented in these triggers.

In sum, the schema and instance mappings onto tables and tuples in combination with the setup of triggers to ensure complex consistency constraints is able to represent the RSQL database model, but without a metatype distinction.

Mapping the University Scenario

To illustrate these mapping rules, the example model shown in Figure 2.3 is transformed into a relational model. At first, the core types are directly transformed by specifying a relation for each core type and creating a surrogate ID. Additionally, the primary keys of these relations are denoted with underlines. The mapping process results in the following six core type relations.

$$
\begin{aligned}
\mathit{Person} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Name} : \mathit{String}, \mathit{LastName} : \mathit{String}},\\
&\underline{\mathit{Birthday} : \mathit{Date}}, \mathit{Address} : \mathit{String}]\}\\
\mathit{Seminar} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Name} : \mathit{String}, \mathit{Date} : \mathit{Date}}, \mathit{Creadits} : \mathit{Integer}]\}\\
\mathit{Lecture} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Name} : \mathit{String}, \mathit{Room} : \mathit{String}}, \mathit{Credits} : \mathit{Integer},\\
&\mathit{Time} : \mathit{String}]\}\\
\mathit{University} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Name} : \mathit{String}, \mathit{Location} : \mathit{String}}, \mathit{Address} : \mathit{String}]\}\\
\mathit{SportsTeam} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Name} : \mathit{String}, \mathit{Colors} : \mathit{String}}, \mathit{Cheerleader} : \mathit{String}]\}\\
\mathit{Tournament} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Name} : \mathit{String}}, \mathit{Date} : \mathit{Date}, \mathit{Location} : \mathit{String}]\}
\end{aligned}
$$

Next, the Role Types are mapped by creating a relation for each of these. Additionally, a surrogate ID is introduced to simplify referencing. Moreover, the reference to the corresponding Compartment Type is added to each of these relations. The mapping process results in ten relations.

$$
\begin{aligned}
\mathit{Student} :\{[&\mathit{ID}, \underline{\mathit{Student\_ID} : \mathit{Integer}}, \mathit{Studies} : \mathit{String}, \mathit{Year} : \mathit{Integer},\\
&\underline{\mathit{C\_ID} : \mathit{Integer}}]\}\\
\mathit{Professor} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Chair} : \mathit{String}, \mathit{Office} : \mathit{String}}, \mathit{ResearchBudget} : \mathit{Integer},\\
&\underline{\mathit{C\_ID} : \mathit{Integer}}]\}\\
\mathit{StudentAssistant} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Assistant\_ID} : \mathit{Integer}}, \mathit{HoursPerWeek} : \mathit{Byte}\\
&\underline{\mathit{C\_ID} : \mathit{Integer}}]\}\\
\mathit{ResearchAssistant} :\{[&\mathit{ID} : \mathit{Integer}, \underline{\mathit{Emp\_ID} : \mathit{Integer}}, \mathit{Office} : \mathit{String}, \mathit{Tel} : \mathit{String},\\
&\mathit{SalaryGroup} : \mathit{Byte}, \underline{\mathit{C\_ID} : \mathit{Integer}}]\}
\end{aligned}
$$

$$
\begin{aligned}
StudiesCourse :\{[&ID : Integer, \underline{Name : String}, \underline{Module : String}, Level : String \\
&\underline{C\_ID : Integer}]\} \\
Captain :\{[&ID : Integer, \underline{Games : Integer}, \underline{C\_ID : Integer}]\} \\
Coach :\{[&ID : Integer, \underline{Position : String}, Specials : String, \underline{C\_ID : Integer}]\} \\
TeamMember :\{[&ID : Integer, \underline{Memeber\_ID : Integer}, Registration : Date, \\
&Position : String, \underline{C\_ID : Integer}]\} \\
TournamentTeam :\{[&ID : Integer, \underline{TeamName : String}, Group : Byte, \underline{C\_ID : Integer}]\} \\
WinnerTeam :\{[&ID : Integer, \underline{FinalResult : String}, \underline{C\_ID : Integer}]\}
\end{aligned}
$$

The Relationship Types are mapped to relations as well. Each is modeled in a separate relation consisting of references to the respective Role Type tables only. A surrogate ID is not introduced, because these tables are not referenced by other ones. However, the primary key is composed of both foreign keys. Applying this mapping procedure results in a specification of three relations.

$$
\begin{aligned}
advises :\{[&\underline{R1\_ID : Integer}, \underline{R2\_ID : Integer}]\} \\
teaches :\{[&\underline{R1\_ID : Integer}, \underline{R2\_ID : Integer}]\} \\
takes :\{[&\underline{R1\_ID : Integer}, \underline{R2\_ID : Integer}]\}
\end{aligned}
$$

Finally, the *fills* relation is mapped to an individual table for each entry. The resulting tables consist of references to the player type table and the corresponding Role Type table only. Thus, the primary keys are composed of the corresponding two foreign keys. The resulting relations are specified as follows.

$$
\begin{aligned}
Person\_Student :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Person\_Professor :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Person\_StudentAssistant :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Person\_ResearchAssistant :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Person\_Captain :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Person\_Coach :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Person\_TeamMember :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Seminar\_StudiesCourse :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
Lecture\_StudiesCourse :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
SportsTeam\_TournamentTeam :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\} \\
SportsTeam\_WinnerTeam :\{[&\underline{P\_ID : Integer}, \underline{R\_ID : Integer}]\}
\end{aligned}
$$

In sum, these relations build the basis to evaluate the RSQL approach in comparison to a relational mapping of it.

6.2 EVALUATING THE DATABASE MODEL

The RSQL database model is designed to overcome the role-relational impedance mismatch (see 3.2). Moreover, we outlined that other database models cannot efficiently model the metatype discrimination and the interrelations between the metatypes on the schema as well as on the instance level. To support the claim of efficiently creating database schemata with RSQL, we compare its schema creation process to a relational SQL-based schema creation. This comprises the creation of types in RSQL and the definition of tables as well as triggers in SQL. Moreover, we show which actions, according to an evolution of the conceptual model, result in which effort to adapt the corresponding database schema.

6.2.1 Creating the RSQL Schema

At first, an RSQL database schema is created by using the data definition language statements as shown in Section 5.3.1. There are individual statements to create the corresponding metatypes. The *fills* relation and functions do not need to be created and populated manually. For instance, the Natural Type **Person** is created by the following statement.

```
1 CREATE NATURALTYPE Person (name varChar(128) PRIMARY KEY,
2   lastname varChar(128) PRIMARY KEY, birthday date PRIMARY KEY,
3   address varChar(256));
```

This primary key definition ensures uniqueness for the composition of the three stated columns. Consequently, there cannot be two **Person** Naturals in the system that have the same values for these three attributes.

A Role Type, for instance the **Student** Role Type, is created by a statement like the following.

```
1 CREATE ROLETYPE Student (Student_ID int PRIMARY KEY,
2   studies varChar(256), year int)
3   PLAYED BY (Person) PART OF University;
```

The primary key is applied on the attribute `Student_ID` and specifies the distinguishing feature. However, this primary key is not valid globally, in fact it is ensured for each Compartment. Because Role Types are embedded in exactly one Compartment Type, their corresponding primary key validity is limited by it. For instance, the student identity is assigned by each university manually and may has different patterns, depending on the university. Although, the same ID may appear several times, but at different universities, whereas at the same university a student identity is uniquely assigned.

Finally, the Relationship Types are created. For instance, the **supervises** Relationship Type as follows.

```
1 CREATE RELATIONSHIPTYPE supervises CONSISTING OF
2   (Professor BEING 0 .. *) AND (Student BEING 0 .. 1);
```

In RSQL Relationship Types do not feature any attributes, because attributed ones can be modeled as individual Compartment Types. Hence, only the participating Role Types including their cardinality constraints are specified.

In sum, for each type in the conceptual schema as type in the database is created. Trigger or additional statements are not required to ensure the database model constraints. Consequently, the sum of statements can be determined by the following equation.

$$|S_{RSQL}| = |NT| + |CT| + |RT| + |RST| \tag{6.1}$$

6.2.2 Creating the SQL Schema

Based on the present and assumed mapping various tables and triggers have to be created. The following statements are based on the syntax of the H2 database engine[5], because the RSQL implementation is based on this database engine [6]. The concrete syntax to specify, for instance primary keys or foreign keys, may differ from system to system, but has the same effects, the corresponding constraints are created.

Base Tables

At first, the tables for the types are created. For instance the table that represents the **Person** Natural Type.

```
CREATE TABLE Person (ID int UNIQUE, name varChar(128) PRIMARY KEY,
  lastname varChar(128) PRIMARY KEY, birthday date PRIMARY KEY,
  address varChar(256));
```

As mentioned in the mapping, each type is added a surrogate ID column, to simplify referencing. This column is additionally set to unique. Next, Role Type tables are created, like the following **Student** table.

```
CREATE TABLE Student (ID int UNIQUE, Student_ID int PRIMARY KEY,
  studies varChar(128), year int, C_ID int,
  FOREIGN KEY (C_ID) REFERENCES University(ID));
```

In addition to the regular attributes, a surrogate ID and a column holding the foreign key value is added. Moreover, the foreign key column is set to reference the surrogate ID of the university. This models a Role Type's belonging to a certain Compartment Type. As final type creation, the Relationship Types are created as separate tables. As example, imagine the Relationship Type **supervises** that is created as follows.

```
CREATE TABLE supervises (R1_ID int PRIMARY KEY,
  R2_ID int PRIMARY KEY UNIQUE,
  FOREIGN KEY (R1_ID) REFERENCES Professor(ID),
  FOREIGN KEY (R2_ID) REFERENCES Student(ID));
```

The create statement for a Relationship Type table consists of two columns that reference the surrogate IDs of the corresponding Role Type tables. Additionally, these columns build the composed primary key. Moreover, a **Student** can be **supervised** by only one **Professor**, thus, the Student's referencing column is set to unique.

Finally, the tables that represent the entries of the *fills* relation have to be created. For each element in this relation, a separate table is added to the schema. For instance, the following association table connecting the Natural Type table Person with the Role Type table Student.

```
CREATE TABLE Person_Student (P_ID int PRIMARY KEY,
  R_ID int PRIMARY KEY,
  FOREIGN KEY (P_ID) REFERENCES Person(ID),
  FOREIGN KEY (R_ID) REFERENCES Student(ID));
```

This table specification is similar to a Relationship Type table, but connects a player type with a Role Type. The primary key and foreign key constraints are defined likewise. Moreover, the Role referencing column is set to unique, because a Role can be played only once. However, this constraint has to be ensured over several association tables, because a Role Type can be filled by possibly various player types.

[5] http://www.h2database.com/html/grammar.html
[6] http://www.h2database.com/html/main.html

Triggers

In case simple SQL consistency constraints cannot model complex RSQL consistency axioms, triggers are used. In the mapping applied to compare SQL and RSQL several complex consistency requirements have to be implemented. They have to be ensured on inserts, updates, and deletes. Generally, inserting a new Role is a complex and atomic process, that requires at least two inserts. One in the Role Type table and one in an association table. Thus, creating a Role has to be handled in a transaction, especially to avoid inconsistent states of unbound Roles, for example. Several metamodel violations may appear during the data manipulation process and most of them have to be caught by triggers, especially in case the information is distributed over several tables. An overview of metamodel violations that may appear during the data manipulation is given in Table 6.1.

Operation	*Table*	*Violation*
Insert	Role Type table	Role has no player
	association table	Role has multiple players
		Role is played several times per player in a certain Compartment
Update	Role Type table	Role is played several times per player in a certain Compartment
	association table	Role is played several times per player in a certain Compartment
		Role has multiple players
		Role has no player
Delete	Role Type table	–
	association table	Role has no player

Table 6.1: Inconsistent States of Role Bindings During Data Manipulation

Generally, two types of triggers are required. In the first place, triggers on a table that are fired for each row manipulation of a statement. This is the situation triggers are usually used. To this type of trigger we refer to as **immediate trigger**. Secondly, triggers that guarantee overall consistency, especially at the end of a transaction. As aforementioned, inserting a Role is a complex process encapsulated within a transaction. The consistency has to be checked at the end of a transaction, thus, triggers on the level of statements or rows cannot be use for this purpose. For instance, a trigger on a Role Type table cannot be executed right before or after an insert in the table, because the relations to its player may be inserted afterwards. Only few database management systems provide support for such triggers, especially the H2 database does not. For instance, PostgreSQL supports this feature by the *deferred* option in the trigger declaration[7]. To this trigger type we refer as **commit trigger**. However, in case commit trigger cannot be defined, a separate stored procedure has to be manually called that encapsulates the consistency checks for a certain axiom. In contrast to triggers, these procedures have to be explicitly called in the transaction and are not fired automatically.

Each of the mentioned anomaly has to be prevented by triggers in the database system. These anomalies and triggers are detailedly discussed in the following.

1. **No player for a Role after an insert in a Role Type table** This anomaly occurs in case an insert in a Role Type table is executed, but no mandatory reference in one of the association tables is created. This can be checked by an **insert commit trigger** on the corresponding Role Type table. For instance, a new **Student** Role is inserted, but no corresponding association tuple in the `Person_Student` table is created within the same transaction. Such triggers have to be created for each Role Type table separately.

[7] `https://www.postgresql.org/docs/9.1/static/sql-createtrigger.html`

2. **A Role has multiple players after an insert in an association table** This axiom violation may appear by inserting tuples in an association table. To check the uniqueness of exactly one player, a uniqueness constraint over several tables is required, because a Role Type may have several player types and this is represented in several tables. Nevertheless, each Role must be referenced exactly once. As there are no multiple table uniqueness constraints available, an **immediate insert trigger** on each row insertion has to be created that checks if the to be inserted Role is already referenced by a player. For instance, a **StudiesCourse** Role Type is related to a **Lecture** in the table `Lecture\_StudiesCourse` and the trigger has to check the uniqueness over all association tables of this Role Type. This comprises the check on `Lecture_StudiesCourse` as well as `Seminar_StudiesCourse`. The tuple will only be inserted, if the check returns that this Role is currently not referenced. The corresponding trigger is defined as follows by using the official SQL 2006 syntax [47, pp. 107, 659]. Creating a trigger in H2 syntax is more complex and requires to implement a separate Java class, which uses the *Trigger* interface of H2[8].

```
CREATE TRIGGER check_Lecture_StudiesCourse
  BEFORE INSERT ON Lecture_Student FOR EACH ROW
  BEGIN ATOMIC
  {
    IF (SELECT COUNT(*) FROM (
      (SELECT * FROM Lecture_Student WHERE R_ID = new.R_ID)
      UNION
      (SELECT * FROM Seminar_Student WHERE R_ID = new.R_ID))
    ) > 0
    THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'ROLE ALREADY PLAYED'
    END IF
  } END
```

This trigger is created on the association table between the Natural Type **Lecture** and Role Type **StudiesCourse**. In detail, an error is returned in case the Role ID is found in any of the association tables this Role Type table is referenced in. These tables are connected united to retrieve these tuples that also reference the Role ID to be inserted. Because the **StudiesCourse** can be filled by two Natural Types, only one `UNION` occurs in this trigger. In case there are more player types for a Role Type, the `UNION` in the corresponding trigger scale with the amount of player types.

3. **Compartment uniqueness for player and Role after an association table insert** Such metamodel violation is characterized by inserting a new tuple in an association table. Additionally, this insert violates the constraints that a Role Type can be instantiated only once per player and Compartment. To avoid such situation an **immediate insert trigger** on each association table has to be created. This constraint could be checked within the trigger created for the second situation, but for the sake of a clear separation of constraint checking, a separate one is created. As example, imagine the `Person_Student` table and a new entry relates a Student Role to a Person. This trigger checks, if the same **Person** plays another **Student** Role in the same Compartment by comparing all foreign keys to a **University** in the corresponding **Student** Roles.

4. **Compartment uniqueness for player and Role after a Role Type table update** A metamodel violation that a Role Type is instantiated several times for the same player within the same Compartment, can appear in case a Role Type table is updated on the Compartment referencing foreign key. For instance, an existing **Student** Role is assigned to another **University** and the **Person** tuple that is associated to this manipulated Role already plays a **Student** Role in this new **University** Compartment. An **update immediate trigger** solves this problem by checking the uniqueness for each manipulation on the foreign key constraint.

[8] `http://h2database.com/html/features.html#triggers`

5. **Compartment uniqueness for player and Role after an association table update** The same anomaly can occur in case the association tables are manipulated by updates. Two subsituations have to be considered. At first, the player referencing column is updated to another player such that a Role gets another player. Secondly, the Role referencing column is updated in a way the player gets another Role. In any case, the consistency is ensured by an **update immediate trigger** on each association table. For instance, assume a Student Role is bound to another player by setting another player reference for the tuple. Consequently, the new situation has to be checked for consistency.

6. **Role has multiple players after an association table update** This violation appears in case the Role referencing column of an association table is updated to an already existing Role reference. Because Role Types can have multiple filling player types, the existence of the Role reference over all corresponding association tables has to be ensured. This is done by an **update immediate trigger**. For instance, there exist two references to distinct **StudiesCourse** Roles and one is updated to be played by another player.

7. **Role has no player after an association table update** This metamodel violation can result in case a Role reference is manipulated, as in the previous violation case. This indicates, the corresponding player now starts playing another Role and the formerly referenced Role remains without a new player, thus, it is not deleted from the system. For instance, assume the following situation, a new **Student** Role is created and is designed to substitutes the old one. The update on the association table is performed by changing the Role association of the old to the new one, but the old Role is not discarded from the system. This situation is handled by an **update commit trigger** that checks for unbound Roles on the corresponding Role Type table after all statements have been executed. Such a check can be performed by checking all Roles in the corresponding Role Type table for an existing player or buffer manipulated Roles separately and only check these Roles. The association table cannot be utilized, because the Role reference is gone.

8. **Role has no player after a delete on an association table** The only metamodel violation that corresponds to a delete, is deleting from an association tables does. Deletes on the Role Type tables are secured by the referential constraints. However, such an anomaly is solved by a **delete commit trigger**. This has to be performed on the transaction level, because a trigger on the statement or row level will block all delete operation due to the interdependence between the Roles and their associations to a player. For example, the association between a Student Role and the corresponding Person tuple is deleted, but the Role stays in the system. This trigger will check, if the each affected Role has a new player or is discarded from the system.

As it can be seen, several critical situations during the Role manipulation exist and the RSQL database model axioms require several triggers. In sum, each Role Type table needs an insert and update trigger. Additionally, all association tables are required to have two insert triggers, three update triggers, and one delete trigger to enforce correct Role bindings and the related constraints. Moreover, for each operation at least one trigger has to be defined on the transaction level, to avoid mutual blocking situations and allow inconsistent states within a transaction. The number of statements to create a relational mapping of the RSQL database model is defined by the number of required tables and triggers, as shown in the following equation.

$$|S_{SQL}| = |Tables| + |Triggers| \tag{6.2}$$

The number of tables is defined by the overall number of types and their interrelations in the *fills* relation.

$$|Tables| = |NT| + |CT| + |RT| + |RST| + |fills| \tag{6.3}$$

As outline, each Role Type table requires a mandatory insert and update trigger. Moreover, each association table features two insert triggers, three update triggers and one delete trigger. In an optimal case, in which the different functionalities of the association table triggers are combined in a single one, at least three triggers are required. For comparison purposes, the optimal case is assumed resulting in the following rule to determine the number of triggers.

$$|Triggers| = 2 \cdot |RT| + 3 \cdot |fills| \tag{6.4}$$

6.2.3 Comparing RSQL and SQL

To compare RSQL and SQL with each other, we determine the number statements that are required to set up a proper schema and to guarantee the consistency constraints for the instances in the database. In general, the calculation rules presented in equation 6.1 and 6.2 are used to determine the number of statements. At first, schema creation of a database with respect to the conceptual model illustrated in Figure 2.3 is compared. An overview of the contained types is given in Table 6.2.

$\|NT\|$	$\|CT\|$	$\|RT\|$	$\|RST\|$	$\|fills\|$
3	3	10	3	11

Table 6.2: Elements Overview of the University Domain Conceptual Model

To create the corresponding database schema several `CREATE` statements in RSQL as well as in SQL have to be executed. Moreover, the mapping process and the respective rules as outlined in Section 6.2.2 are applied to determine the number of SQL statements. An overview of the number of statements is given in Table 6.3.

Statement	RSQL	SQL
`CREATE TABLE / TYPE`	19	30
`CREATE TRIGGER`	–	53
Sum	19	83

Table 6.3: Number of Statements to Create Corresponding Database Schema in RSQL and SQL

In sum, 19 RSQL statements are required to model the conceptual domain in RSQL. In contrast, the same domain in a relational DBS requires 83 SQL statements. Precisely, 30 statements to create the type and association tables, and 53 create statements to ensure the consistency constraints with triggers. Consequently, the number of statements and effort to create the same semantics in SQL is 4.3 times higher than in RSQL.

Sensitivity Analysis of the *fills* Relation

The actual number of SQL data definition language declarations is mainly influenced by the number of Role Types and the population of the *fills* relation. This relation has a minimum and maximum population. In the former case, all Role Types have exactly one possible player. In the latter one, each core type can fill all available Role Types. To demonstrate the influence of the relation's population on the number of statements, we vary the ratio of players per Role Type.

The sensitivity analysis shown in Figure 6.6 varies the population ratio of the *fills* relation, starting from a ration of 0.2 up to a ratio of 1 denoted as Max. In detail, the bottom line describes the number

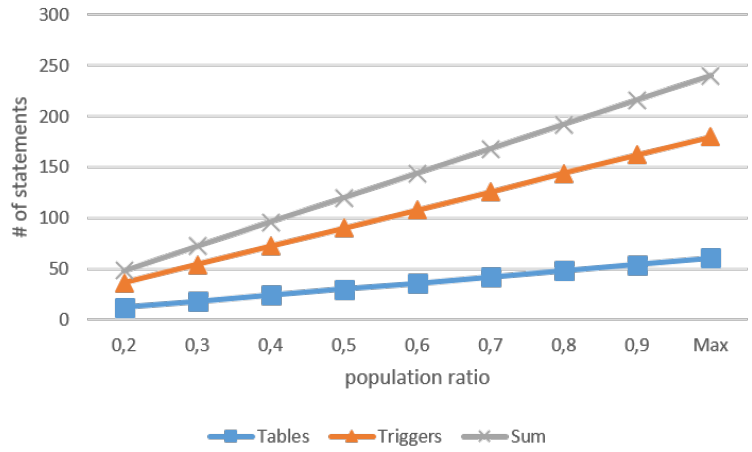


Figure 6.6: Sensitivity Analysis for the *fills* Relation in SQL

of statements required to create the corresponding association tables. The middle line shows the effort in creating trigger statements while the topmost line represents the sum of both lines. Moreover, this example assumes the same number of types as the university domain model, but connects more Role Types with player types. In RSQL, the number of statements is independent of population ratio of this relation, because it does not require separate types to represent it. In contrast, a relational mapping has to model this relation explicitly, which result in effort to create and maintain the tables and especially triggers.

Population	*Min*	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	*Max*
RSQL	19	19	19	19	19	19	19	19	19	19
SQL	79	87	111	135	159	183	207	231	255	279
Quotient	x4.16	x4.58	x5.84	x7.11	x8.37	x9.63	x10.89	x12.16	x13.42	x14.68

Table 6.4: Additional Effort for Creating a Relational Schema by Varying the *fills* Relation's Population Ratio

Depending on the population rate of the *fills* relation, the additional effort in SQL compared to RSQL varies as well. Table 6.4 comprises different population ratios of this relation and directly compares the additional effort in SQL on the basis of the applied mapping. The results represent the total number of statements, including types, tables, and triggers. As it can be seen, the number of RSQL statements is independent of the population ratio, in fact only the Role Type's create statement length increases with a higher ratio. This is caused by the enumeration of all player types within the corresponding create statements. However, by adding additional interrelations between the available Role Types and player types, the additional varies from 4.16 in the minimum case to 14.68 more statements in the fully connected case.

Varying the Number of Types

The university management domain as depicted in Figure 2.3 is a small example that features 19 different types only. To analyze the number of statements in more detail, the number of type is varied by keeping the ration in the *fills* static. As base data assume the three scenarios as shown in Table 6.5 with a population ratio of 0.2 of the *fills* relation.

Scenario	$\|NT\|$	$\|CT\|$	$\|RT\|$	$\|RST\|$	$\|fills\|$	$\|S_{RSQL}\|$	$\|S_{SQL}\|$	Quotient
1	1	1	1	0	1	3	9	x3
2	10	10	10	10	40	40	220	x5.5
3	10	20	50	10	300	90	1390	x15.44
4	2	10	100	10	240	122	1282	x10.51

Table 6.5: Comparison of the Number of Statements as a Function of the Number of Types

The first scenario comprises the most basic scenario, with one Natural Type, Compartment Type, and Role Type, but no Relationship Type. In RSQL this requires 3 statements and in SQL 9 by applying the mapping schema, which is three time more statements. Moreover, the *fills* relation has a fill rate of 0.5, because the example includes only one Role Type that is filled by one of two potential player types. Less it not possible.

The second scenario illustrates an equal amount of types for each metatype, in detail ten types each. As result of a 0.2 population ratio of the *fills* relation, it holds 40 entries. The schema in RSQL requires 40 statements contrary to 220 statements in SQL. In more detail, 80 statements are required for creating tables and 140 for the triggers to ensure instance consistency. In sum, the 5.5 times more SQL statements have to be built.

In the third example, the amount of Role Types is quintupled whereas the number of Compartment Types is doubled, compared to the previous example. All other amounts remain stable. This represents a scenario with few cores types and a lot of Role Types. As a consequence, the *fills* relation consists of 300 elements, which is the main reason for the 1390 SQL statements required to set up this scenario. In contrast, such an RSQL schema is created by only 90 statements. Thus, the SQL schema uses 15.44 more statements than the RSQL one.

The final example demonstrates a situation, in which only small number of core types and a large amount of Role Types is present. In detail, two Natural Types and ten Compartment Types on the one hand side and 100 Role Types on the other side. This results in 240 association tables in SQL with a total amount of 1282 statements to create such scenario. This is 10.51 time higher than in RSQL, which requires 122 statements only.

Conclusions

In sum, the database model comparison reveals a, partially huge, difference in the amount of statements needed to create the database schemata. This difference ranges from 3 times more statements in the most basic case (see Scenario 1), over 4.3 times more statements in the running example university domain, to a 15.44 times higher amount of statements in the discussed Scenario 3. Beside the pure number of statements, it can be concluded that the semantics of the RSQL database model are widely distribute over several tables. For instance, the information which player type can fill which Role Types is distributed over several association tables. Moreover, most of the instance constraints, like one Role per Role Type for a player in a certain Compartment, is ensured by triggers, which might become a problem for concurrent applications and their development. Precisely, when a database schema is investigated for existing information that can used in an additional application, the existing tables are of major interest. In the example mapping used to compare the RSQL database model to a relational one, most of the consistency constraints are hidden in triggers, thus, they are not visible on the first sight. This means, the exposed types are tables and triggers work in the background. In RSQL, the exposed types include the consistency constraints as inherent characteristic, in SQL they

are defined in separate types. This background processes can lead to unexpected output and behavior of the database system, from an external perspective. Especially, in case tuples cannot be inserted or deleted from the database, because the triggers prevent that, or the insert and delete statements are not embedded into proper transactions. In sum, the RSQL database model is cut into pieces and spread over several tables and triggers which massively complicates the semantics reconstruction.

Beside this problem of the relational mapping, the bare number of triggers created to ensure consistency produces a very fragile relational schema according to schema evolution. This results in a higher maintenance effort for database developers and administrators.

To sum up, the RSQL database is superior to the relational in terms of modeling role-based systems efficiently. The RSQL database model provides an out of the box role notion, which simplifies the modeling compared to a relational mapping dramatically, because the semantics is directly included in the database model and does not need to be constructed implicitly. Finally, the constraints for schema as well as for the instance level are ensured without explicitly model them, which has to be performed by a relational mapping, especially by using triggers.

6.3 EVALUATING THE QUERY LANGUAGE

To compare the role-based RSQL and relation SQL approach, we directly compare both languages to each other. The use of a language is very subjective and strongly depends on the experience and feeling of the language user. Moreover, there is no standard metric or best practice guide on measuring a language's expressiveness or complexity. Thus, we use various metrics as indicators to achieve a comparison and derive a conclusion. In detail, we compare the amount of used words, the number of character with spaces and the number without them. To have a fair comparison, the standard conceptual model names are used as illustrated in Figure 2.3. Moreover, the abbreviations are as short as possible to avoid unnecessary characters. In detail, the query language evaluation uses two example queries. The first is a simple query that features only a single ⟨*config-expression*⟩. The second one is a more complex query that uses overlapping ⟨*config-expressions*⟩ in combination with a condition.

6.3.1 Writing Single Config-Expressions

The first scenario comprises a simple query on a single Dynamic Data Type by using one ⟨*config-expression*⟩. Precisely, this query asks for Person Dynamic Tuples that play a Role of the Role Types Student and StudentAssistant. In RSQL this query is formulated as follows.

```
1 SELECT * FROM Person p PLAYING Student s AND StudentAssistant sa;
```

Assuming the introduced mapping, the same query rephrased in SQL looks like the following.

```
1 SELECT * FROM Person p
2   INNER JOIN Person_Student ps ON p.ID = ps.P_ID
3   INNER JOIN Student s ON ps.R_ID = s.ID
4   INNER JOIN Person_StudentAssistant psa ON psa.P_ID = p.ID
5   INNER JOIN StudentAassistant sa ON psa.R_ID = sa.ID;
```

Measurement	*Words*	*Chars*	*Chars w/o spaces*	Types / Tables
RSQL	10	65	55	1 (3)
SQL	36	219	183	5
Relative	x3.6	x3.37	x3.33	x5 (x1.67)
Absolute	26	154	128	4 (2)

Table 6.6: Comparison of RSQL and SQL for a Single Config-Expression

As it can be easily seen, the SQL query is much more complex in terms of joining various tables. Moreover, the semantics which player has to be connected to which Roles are vanished by several joins. This is caused by the mapping itself, which implements association tables between the player and Role Types. The original semantics have to be reconstructed by these joins. In Table 6.6 a detailed comparison of RSQL and SQL, according to the introduced queries, is presented.

In particular, the number of words used in the query differs by the factor 3.6 between both query languages while RSQL is shorter. The RSQL query uses only 10 word and in contrast to that, SQL requires 36 words. Note, the $*$ is not counted as separate word. As words vary in their length, the number of characters with and without spaces is compared. The corresponding RSQL query requires 65 characters including the spaces and 55 excluding them. The SQL queries uses more than 3 times more characters. In detail, the number of characters used is 219 in case the spaces are included and 183 when they are excluded. Finally, we compare the amount of computed tables and types, respectively. The RSQL query touches one Dynamic Data Type only and 3 types if the base elements are counted. In contrast to this, the SQL example consumes 5 different tables.

6.3.2 Writing Overlapping Config-Expressions

As second scenario, a more complex and overlapping query is employed. In detail, this scenario asks if good sportsmen are good students as well. To query this information, an attribute GPA is imaginary added to the Role Type **Student**. In detail, a **Person** Dynamic Tuple has to play Role of the Role Type **Student** and **TeamMember**. This **TeamMember** has to be in a certain **SportsTeam** that attended a **Tournament** and won it. The win has to take place prior 2015. Moreover, the Student's GPA must be greater than 4. In RSQL such a query is specified as follows.

```
SELECT * FROM Person p PLAYING Student s AND TeamMember tm,
  Uni u FEATURING s,
  SportsTeam st FEATURING tm PLAYING WinnerTeam wt,
  Tournament t FEATURING wt,
  WHERE t WITH t.date < 01.01.2015, s WITH s.gpa > 4;
```

This RSQL query comprises four different ⟨*config-expressions*⟩, one for each queried Dynamic Data Type. Moreover, the Dynamic Data Types overlap at certain points. In particular, the **Person** overlaps with the **University** by the Role Type **Student**, and with the **SportsTeam** in the Role Type **TeamMember**. Additionally, the **SportsTeam** shares the Role Type **WinnerTeam** with the **Tournament** Dynamic Data Type. Furthermore, the **Tournament** as entire Dynamic Tuple is filtered by the *date* attribute. Finally, the **Student** Roles are filtered by the *gpa* attribute, which has to be greater than 4.

To retrieve the same information from a relation database system, the following query is executed.

```
SELECT * FROM Person p
  INNER JOIN Person_Student ps ON p.ID = ps.P_ID
  INNER JOIN Student s ON ps.R_ID = s.ID
  INNER JOIN Person_TeamMember ptm ON ptm.P_ID = p.ID
  INNER JOIN TeamMember tm ON ptm.R_ID = tm.ID
  INNER JOIN SportsTeam st ON tm.C_ID = st.ID
  INNER JOIN University u ON s.C_ID = u.ID
  INNER JOIN SportsTeam_WinnerTeam stwt ON stwt.P_ID = st.ID
  INNER JOIN WinnerTeam wt ON stwt.R_ID = wt.ID
  INNER JOIN Tournament t ON wt.C_ID = t.ID
  WHERE t.date < 01.01.2015 AND s.gpa > 4;
```

This query uses nine joins to combine the various tables, which gives the query a very homogeneous appearance, but in a negative sense. Basically, each line looks alike with only slight differences. Moreover, the query mixes instance information and mapping information such that the query itself is interspersed with mapping details, like association tables that are joined on entity tables. Thus, the original semantics of this query is hard to identify. This may complicate query debugging in case of wrong or unexpected results. In contrast, the RSQL query only includes entity and overlapping information without the need to include for mapping information. The evaluation on our metrics is presented in Table 6.7.

Measurement	| *Words* |	| *Chars* |	| *Chars w/o spaces* |	| Types / Tables |
RSQL	36	207	171	4 (7)
SQL	84	489	394	10
Relative	x2.33	x2.36	x2.30	x2.5 (1.43)
Absolute	48	282	223	6 (3)

Table 6.7: Comparison of RSQL and SQL for Multiple Config-Expressions

The comparison on the word count reveals a 2.33 higher amount of words in the regular and shortened version of SQL. In particular, the RSQL query uses 36 words whereas the SQL versions require 84 words. Moreover, the number of characters including the spaces in the RSQL is 207. In contrast, the SQL version has a character count of 489 characters with counting the spaces. Compared to RSQL, these number are 2.36 times higher. The relative difference in the word and character counts may shrink, but the absolute difference grows as the character count clearly shows. In the first scenario the absolute difference is 154 and in this scenario 282. In case the space is excluded from the character set, the number go down 171 in the RSQL query and 394 the SQL query. This is 2.30 times higher than the RSQL character count. Finally, the number of used types respectively tables is compared. The RSQL query uses 4 Dynamic Data Types and touches 7 base types, whereas the SQL versions requires 10 tables. Consequently, the number of handled types is 2.5 time higher in the SQL cases by assuming the Dynamic Data Types as reference and 1.43 times higher in case the RSQL base elements are counted.

Conclusions

In sum, the comparison of RSQL's query language and SQL reveals longer query statements in SQL. This is mainly caused by the distribution of role-based entity information over several tables and the introduction of association tables. These tables have to be manually joined in the query statement to reconstruct the original information. Moreover, an RSQL statement is structured by Dynamic Data Types while a SQL statements buries the playing and featuring information in joins over foreign keys, which complicates query debugging. Furthermore, SQL statements look very homogeneous in a negative sense, because they mix entity and mapping information in the same query. However,

the relative difference between RSQL and SQL in count words as well as in the characters shrinks. In contrast, the absolute difference grows. Thus, there is no clear dependency between the amount of words used in RSQL and SQL. To conclude, the RSQL query language provides a sophisticated external database interface to retrieve role-based data from the database.

6.4 EVALUATING THE RESULT REPRESENTATION

RSQL's result representation RuN uses a hybrid approach, consisting of navigation and iteration, to browse the results and collect the required information. In fact, an entity's information, including the Roles, is concentrated in a single Dynamic Tuple, which is processable at once. In contrast, relational results cannot represent the dimensionality of RuN. Thus, the entity information is split into pieces and distributed over several tuples.

There are two extremes of representing a result set of complex and role-based entities in a standard relational result set. At first, all entity information is included in a single query and thus, in a single result set. This introduces redundancy in the result set, because the complex entity information is distributed over several tuples. As a consequence of the redundancy, a more complex result processing is required to deal with this redundant data. The second option distributes the information over several result sets. Thus, multiple queries have to be executed, especially to retrieve the information that is not present in the other result sets. For instance, at first the core table is queried and processed. Next, for each core the association tables are queried in a separate statement to get the information which core is playing which Role. Finally, a query on the Role Type table is executed to collect the concrete Role information. This basically simulates an application-implemented join. Thus, the entity information is distributed over several queries and tuples.

To evaluate RuN's hybrid approach, to navigate and iterate through the results, the query processing on the client side as well as the result's dimensionality respectively size are evaluated. As initial scenario to demonstrate the processing models, the client queries for **Persons**, who play Roles of the Role Type **Student** and **StudentAssistant**. Additionally, the corresponding attributes of an entity have to be stringed together, such that each entity and its Roles are printed in a single line. This simulates the restoring of an application's runtime object. The following code examples are written in Java[9] and use a JDBC interface to communicate with the H2 database engine[10].

6.4.1 Processing a RSQL Result Net

At first, the RuN result processing is elaborated and discussed. Figure 6.7 illustrates the result representation of a RuN. In detail, the database driver is loaded and the connection to the database system is established. On this connection a new statement is created (line 3). Next, the statement is used to execute the query and retrieve the RuN (line 4). In fact, the RSQL query starts with a `SELECT_RSQL`, which is caused by the prototypical implementation. Moreover, this helps in debugging scenarios to follow the parsing paths in the RSQL parser. The actual result has to be casted to a *JdcbRsqlResultSet* in order to get the RSQL specific behavior, like accessing the core attributes or navigate into the dimensions. The result processing is performed between the lines 6 and 18. At first, the main cursor is iterated in a *while* loop, which moves the cursor from one Dynamic Tuple to the next one. For each Dynamic

[9] `https://www.oracle.com/de/java/index.html`
[10] `http://www.h2database.com/html/tutorial.html#connecting_using_jdbc`

```
1  Class.forName("org.h2.Driver");
2  Connection conn = DriverManager.getConnection("jdbc:h2:~/RSQLExample", user, pw);
3  Statement stmt = conn.createStatement();
4  JdbcRsqlResultSet rsqlResult = (JdbcRsqlResultSet) stmt.executeQuery("SELECT * FROM Person p
       ↪ PLAYING Student s AND StudentAssistant sa");
5  String out = "";
6  while (rsqlResult.next()) { //iterate Person Dynamic Tuples
7    out = "Person: " + rsqlResult.getString("p.name");
8    LinkedList<RsqlRoleInstance> students =
       ↪ rsqlResult.getRoleInstances(RsqlDimensionTuple.PLAYING_DIMENSION, "Student"); //get
       ↪ Student Roles
9    for (int i = 0; i < students.size(); i++){
10     out = out + " Student Role: " + students.get(i).getInt("s.Student_ID");
11   }
12   LinkedList<RsqlRoleInstance> studentAssistants =
       ↪ rsqlResult.getRoleInstances(RsqlDimensionTuple.PLAYING_DIMENSION, "StudentAssistant");
       ↪ //get StudentAssistant Roles
13   for (int i = 0; i < studentAssistants.size(); i++){
14     out = out + " StudentAssistant Role: " + studentAssistants.get(i).getString("sa.SA_ID");
15   }
16   System.out.println(out); //print out
17 }
18 rsqlResult.close();
19 conn.close();
```

Figure 6.7: Processing an RSQL RuN

Tuple the core attributes are available right away and can be accessed by using the well-known getter methods of JDBC.

Next, the endogenous navigation is performed in line 9. The RuN returns all Roles of the particular Dynamic Tuple in a linked list, that is capable to be iterated. This iteration process takes place in the lines 10 to 12. Basically, for each Role the Student_ID attribute is attached to the output string. The same procedure is applied for the StudentAssistant Roles in the lines 14 to 16. Finally, the Dynamic Tuple is completely processed, in terms of collecting the corresponding IDs, and the output string is printed (line 17). To complete the result processing, the result as well as the connection are closed.

6.4.2 Processing an All In One Relational Result

The first relational result processing example reflects the all in one extreme, in which a single query execution retrieves all data, but introduces data redundancy. The corresponding code is illustrated in Figure 6.8. In the first place, the database driver is loaded and the database connection is established. This connection object is utilized to create a statement object on which query is executed. The query itself is the same as presented in the relational query language example, illustrated in the single ⟨*config-expression*⟩ example in Section 6.3, but with an additional ordering. In detail, the result is ascending ordered by the `Person.name` and `Person.lastName`, the `StudentAssistant.ID` and `Students.Student_ID`. This helps to implement the result processing, because it clusters the rows by the entity core. As soon as a new `Person.ID` appears during the iteration process, the previous **Person** entity is completely processed. To remove the redundant data from the result, we define three sets to track already seen types, one for already seen **Persons**, one for the **Students**, and one for the **StudentAssistants**. These sets are used to check if the corresponding instance ID has already been seen during result processing and to avoid a redundant representation in the output string.

However, the actual result processing step is specified between the lines 11 and 38 in a *while* loop. This loop moves the row cursor and for each row it is checked if the current contains a new **Person**,

```
1  Class.forName("org.h2.Driver");
2  Connection conn = DriverManager.getConnection("jdbc:h2:~/SQLExample1", user, pw);
3  Statement stmt = conn.createStatement();
4  ResultSet result = (ResultSet) stmt.executeQuery("SELECT * FROM Person p INNER JOIN
       ↪ Person_Student ps ON p.ID=ps.P_ID"
5      + " INNER JOIN Student s ON s.ID=ps.R_ID INNER JOIN Person_StudentAssistant psa ON
       ↪ psa.P_ID=p.ID"
6      + " INNER JOIN StudentAssistant sa ON sa.ID=psa.R_ID ORDER BY p.name, p.lastName, sa.SA_ID,
       ↪ s.Student_ID ASC");
7  String out = "";
8  int seenPerson = 0
9  HashSet<Integer> seenStudents = new HashSet<Integer>();
10 HashSet<Integer> seenStudentAssistants = new HashSet<Integer>();
11 while (result.next()) { //iterate tuples
12   if (result.getInt("p.P_ID") != seenPerson) { //whole new entity
13     if (!out.equals("")) { //reset for new entity
14       System.out.println(out); // print out previous entity
15       out = "";
16       seenStudents.clear();
17       seenStudentAssistants.clear();
18     }
19     seenPerson = result.getInt("p.P_ID");
20     seenStudents.add(result.getInt("s.S_ID"));
21     seenStudentAssistants.add(result.getInt("sa.SA_ID"));
22     out = out + "Person: " + result.getString("p.name");
23     out = out + "Student Role: " + result.getInt("s.Student_ID");
24     out = out + "StudentAssistant Role: " + result.getInt("sa.SA_ID");
25   } else { //existing entity
26     if (!seenStudents.contains(result.getInt("s.S_ID"))) {
27       seenStudents.add(result.getInt("s.S_ID"));
28       out = out + "Student Role: " + result.getInt("s.Student_ID");
29     }
30     if (!seenStudentAssistants.contains(result.getInt("sa.SA_ID"))) {
31       seenStudentAssistants.add(result.getInt("sa.SA_ID"));
32       out = out + "StudentAssistant Role: " + result.getInt("sa.SA_ID");
33     }
34   }
35 }
36 System.out.println(out); // print out final entity
37 result.close();
38 conn.close();
```

Figure 6.8: Collecting Entity Information in a Single Relational Result

a new **Student** Role, or a new **StudentAssistant** Role. In case it is a new **Person**, the output string is printed and reset. Moreover, the corresponding sets to keep track of seen information are cleared, as shown in the lines 14 to 18 in Figure 6.8. Next, the new **Person**, the first **Student** Role as well as the first **StudentAssistant** Role are added to the tracking sets and the output string (lines 20 to 25). In case the **Person** information has been seen before, it is checked if a new **Student** or **StudentAssistant** Role is included in this row. Depending on the check result, the information is appended to the output string and the type is added to the particular tracking set. This loop continues until all rows have been processed. Finally, the last entity is printed and the result as well as the connection are closed. In total, this processing approach requires additional tracking effort to filter the redundant information and detect which tuples contain Role information of which Role Type. Moreover, the redundant data representation results in higher transmission costs between the database system and the applications.

6.4.3 Processing a Multi-Query Relational Result

The second relation result processing example avoids the data redundancy in the result representation, but has to load data on demand by executing a query each time. The client side code for this

example is depicted in Figure 6.9. After loading the H2 database driver and establishing a database connection, the statement, on which the initial query is executed, is created (lines 1 – 3). Moreover, four prepared statements are created and initialized, one for each table that is queried during the main tuple processing. The use of prepared statements is beneficial and more efficiently for frequently performed queries.

```
1  Class.forName("org.h2.Driver");
2  Connection conn = DriverManager.getConnection("jdbc:h2:~/SQLExample2", user, pw);
3  Statement stmt = conn.createStatement();
4  PreparedStatement pstmtPS = conn.prepareStatement("SELECT * FROM Person_Student ps WHERE
       ↪ ps.P_ID=?");
5  PreparedStatement pstmtS = conn.prepareStatement("SELECT * FROM Student s WHERE s.ID =?");
6  PreparedStatement pstmtPSA = conn.prepareStatement("SELECT * FROM Person_StudentAssistant psa
       ↪ WHERE psa.P_ID=?");
7  PreparedStatement pstmtSA = conn.prepareStatement("SELECT * FROM StudentAssistant sa WHERE
       ↪ sa.SA_ID =?");
8  ResultSet result = (ResultSet) stmt.executeQuery("SELECT * FROM Person p");
9  String out = "";
10 while (result.next()) { //iterate Persons
11   pstmtPS.setInt(1, result.getInt("p.ID"));
12   ResultSet resultPS = pstmtPS.executeQuery();
13   out = "Person: " + result.getString("p.name");
14   while (resultPS.next()) { //iterate Person_Student
15     pstmtS.setInt(1, resultPS.getInt("ps.R_ID"));
16     ResultSet resultS = pstmtS.executeQuery();
17     while (resultS.next()) { //iterate Student
18       out = out + " Role Student: " + resultS.getInt("s.Student_ID");
19     }
20     resultS.close();
21   }
22   resultPS.close();
23   pstmtPSA.setInt(1, result.getInt("p.ID"));
24   ResultSet resultPSA = pstmtPSA.executeQuery();
25   while (resultPSA.next()) { //iterate Person_SAssistant
26     pstmtSA.setInt(1, resultPSA.getInt("psa.R_ID"));
27     ResultSet resultSA = pstmtSA.executeQuery();
28     while (resultSA.next()) { //iterate StudentAssistant
29       out = out + " Role StudentAssistant: " + resultSA.getInt("sa.SA_ID");
30     }
31     resultSA.close();
32   }
33   resultPSA.close();
34   System.out.println(out); //print out entity
35 }
36 result.close();
37 conn.close();
```

Figure 6.9: Collecting Entity Information in Multiple Relational Results

Firstly, the main cursor is moved to iterator over all Person tuples that are retrieved by the main query execution (line 8). Secondly, the first prepared statement is parametrized with the current **Person** tuple's ID and executed afterwards (lines 9 and 10). This query's result describes the linkage between the current **Person** tuple and its related **Student** Roles. In case, it is related, all relations will be iterate in a nested *while* loop. For each of the result entries, the corresponding Student Role tuple is loaded by parameterizing and executing the second prepared statement (lines 15 and 16). This result sets represents the actual Student Role information, thus, for each element in this result, the output string is extended by the corresponding Role information (lines 17 – 19). If this iteration process is finished, the corresponding result set objects are closed (lines 20 – 21). The same information collection process is applied on the association table `Person_StudentAssistant` in combination with the Role Type table `RT\_SAssistant`. Therefor, the prepared statements *pstmtPSA* and *pstmtSA* are parametrized and repeatedly executed. Finally, the results are closed and the output string is printed (lines 31 – 37). In sum, this process requires additional effort to query the required information. Moreover, the joins are performed on the application level and not in the database system.

6.4.4 Comparing the Result Representations

To compare the three result processing models, several indicators are contrasted to each other in four scenarios. Generally, the efficiency of the result representation can be measured in effort required to process it on the client side and its dimensions in terms of elements contained or data transferred between the database system and the applications. To assess the processing effort, two scenarios and five metrics are applied on the three processing models. The scenarios vary by their number of queried Role Types. Furthermore, the result dimensions are investigated by varying the number of Roles played per Role Type and by using three metrics.

Varying the Number of Role Types

To illustrate the complexity of the client side code and its expansion with an increasing number of queried Role Types, five different measurements are detailed for each of the scenarios. Precisely, (i) the number of queries executed ($|Q|$), (ii) the size of the initial result ($|RS|$), (iii) the amount of lines of code ($|Loc|$), (iv) the number of conditional statements ($|If|$), and finally (v) the number of loops ($|Loop|$). The last three metrics are influenced by a programmer's individual coding style and may vary from person to person. For a fair comparison, all examples use the same style and apply the same patterns, for instance, to iterate lists. To determine the expansion and metrics several rules have to be applied.

$|Q|$ The RSQL as well as the SQL all in one scenario require only one statement to retrieve all information. In contrast, the separate query SQL scenario executes queries for each Natural, Role Type, and Role combination plus 1. The additional query represents the initial one.

$|RS|$ The RSQL as well as separate SQL query result are populated with the number of Naturals whereas the all in one SQL scenario result holds as many rows as the combination of Naturals, Role Types and Roles.

$|Loc|$ The lines of code in RSQL are determined by the basic summand of 6 lines and four additional lines for each Role Type. This is also shown in Figure 6.7. In case of SQL all in one, the skeleton consists of 17 lines and is extended by 8 lines per Role Type, as shown in Figure 6.8. Finally, the SQL scenario implementing separate queries has an initial effort of 6 lines of code, but is expanded by 13 lines for each Role Type. These 13 lines are also shown in Figure 6.9.

$|If|$ As the code examples show, RSQL as well as the separate query SQL scenario do not utilize conditional statements at all. In contrast, the all in one SQL case uses two basic statements and requires one more for each queried Role Type.

$|Loop|$ An RSQL result processing demands for at least one basic loop and an additional one for each Role Type. The first SQL example uses only one loop, especially to move the main cursor. In contrast, the separated SQL setting needs the main loop and two more for each Role Type.

Moreover, it is assumed that all Dynamic Tuples are equally populated, which means each core plays the same amount of Role per Role Type. This simplifies especially the computation of the result size and number of queries. An overview of the rules is given in Table 6.8 and the two scenario evaluations are presented in Table 6.9.

In detail, the first scenario assumes two Natural playing ten Roles each. These Roles are equally distributed over five Role Types. By definition, RuN as well as the all in one SQL scenario have a

Setting	*Metric*	*RuN*	*SQL all in one*	*SQL separate*
Rules	$\|Q\|$	1	1	$1 + \|N\| \cdot \|RT\| \cdot \|R\|$
	$\|RS\|$	$\|N\|$	$\|N\| \cdot \|RT\| \cdot \|R\|$	$\|N\|$
	$\|Loc\|$	$6 + 4 \cdot \|RT\|$	$17 + 8 \cdot \|RT\|$	$6 + 13 \cdot \|RT\|$
	$\|If\|$	0	$2 + \|RT\|$	0
	$\|Loop\|$	$1 + \|RT\|$	1	$1 + 2 \cdot \|RT\|$

Table 6.8: Rules to Determine the Five Metrics For Each Result Processing Model

Setting	*Metric*	*RuN*	*SQL all in one*	*SQL separate*	*Q1*	*Q2*
2 N, 5 RT, 2 R per RT	$\|Q\|$	1	1	21	x1	x21
	$\|RS\|$	2	20	2	x10	x1
	$\|Loc\|$	26	57	71	x2.19	x2.73
	$\|If\|$	0	7	0	–	–
	$\|Loop\|$	6	1	11	x0.17	x1
2 N, 50 RT, 2 R per RT	$\|Q\|$	1	1	201	x1	x201
	$\|RS\|$	2	200	2	x100	x1
	$\|Loc\|$	206	417	656	x2.02	x3.18
	$\|If\|$	0	52	0	–	–
	$\|Loop\|$	51	1	101	x0.02	x1.98

Table 6.9: Comparison of the Three Result Processing Models by Two Role Type Varying Scenarios

constant number of queries, exactly 1. For this scenario the separate SQL case requires 21 executed queries to retrieve all entity information. In contrast, the number of elements in the initial result is 2 for the RSQL and separated SQL cases, whereas the other SQL scenario contains 20 elements. As it can be seen, the two relational processing models feature one positive aspect, fewer queries or less initial elements in the result, but RuN combines both aspects. However, the code expansion is measured by the lines of code, the number of conditional statements, and the amount of loops. While RSQL only uses 26 lines and 6 loops, the all in one SQL scenario requires 57 lines, consisting of 7 conditional statements and the main *while* loop. Furthermore, the split SQL setting requires 71 lines of code with 11 embedded loops. Especially, the number of lines of code is more than two times higher in both SQL processing models.

The second scenario implements ten times more Role Types while the number of Roles per Role Type and the amount of entities remains stable. Thus, each entity plays 100 Roles distributed over 50 Role Types. This increases the number of queries in the separated SQL setting to 201, whereas the SQL and RSQL number stays at 1. In contrast, the number of elements in the initial result is stable for the RuN and separated SQL cases, but grows for the other SQL scenario from 20 to 200. The lines of code expand from 26 to 206 in the RSQL scenario, from 57 to 417 in the first SQL case, and from 71 to 656 in the separated SQL setting, which is more than two times more code. The number of conditional statements grows linearly to the amount of Role Types in the all in one SQL example, thus, 52 of these statements are required. Finally, the amount of loops increases from 6 to 51 for RSQL and from 11 to 101 in the second processing model of SQL.

Varying the Number of Roles per Role Type

To demonstrate the result dimensions and their growth, three metrics applied in two scenarios are evaluated. Both, the number of queries and the amount of elements in the initial result, are used in

previous evaluations. Thus, an explanation is skipped at this point. As additional metric, the amount of data transferred between the database system and the application is introduced. In case of RSQL and the separate query SQL scenario, this number is defined by the number of Natural in the result multiplied with the entities Natural and Roles payloads. In detail, we assume a **payload of 1KB** per Natural and per Role. Based on this, a 1 is added to the overall Role payload for each Natural. In fact, this 1 represents the Natural's payload. In the all in one SQL setting, this metric is determined by the number of Role Types plus 1, multiplied with the number of elements in the result. The former factor represents the header dimension, which consists of as many Role Types as queried plus the Natural Type. The latter one describes the total amount of elements in the result as multiplication of Naturals, the amount of Role Types, and the Roles per Role Type. Moreover, the amount of data caused by the association tables is neglected in the examples. The rules to determine the metrics are listed in Table 6.10. Moreover, Table 6.11 presents the evaluation for two scenarios.

Setting	*Metric*	*RuN*	*SQL all in one*	*SQL separate*
Rules	$\|Q\|$	1	1	$1 + \|N\| \cdot \|RT\| \cdot \|R\|$
	$\|RS\|$	$\|N\|$	$\|N\| \cdot \|RT\| \cdot \|R\|$	$\|N\|$
	$\|Data\|$	$\|RS\| \cdot (1 + \|RT\| \cdot \|R\|)$	$\|RS\| \cdot (1 + \|RT\|)$	$\|RS\| \cdot (1 + \|RT\| \cdot \|R\|)$

Table 6.10: Rules to Determine the Three Metrics For Each Result Processing Model

Setting	*Metric*	*RuN*	*SQL all in one*	*SQL separate*	*Q1*	*Q2*
2 N, 5 RT, 10 R per RT	$\|Q\|$	1	1	101	x1	x101
	$\|RS\|$	2	100	2	x100	x1
	$\|Data\|$	102	5100	102	x50	x1
2 N, 5 RT, 100 R per RT	$\|Q\|$	1	1	1001	x1	x1001
	$\|RS\|$	2	1000	2	x500	x1
	$\|Data\|$	1002	6000	1002	x5.99	x1

Table 6.11: Comparison of the Three Result Processing Models by Two Role Varying Scenarios

The first scenario, comprises moderately populated Role Types for each Natural. In detail, we assume 5 Role Types and each is filled with 10 Roles, which are 50 Roles per Natural. The number of queries remains stable for RSQL and the all in one SQL example. In the second relational result processing model, 101 queries are required, which is 101 times more queries than in RSQL. The number of elements in the main result is 2 for RSQL and the separated scenario, but 100 for the first SQL example. The data transferred from the database to the application is 50 time higher in the all in one scenario than in the RSQL and separated query setting.

To investigate the result dimensions in case more Roles per Role Type are played, the second example scenario is employed. In particular, this example assumes 100 Roles per Role Type while the amount of overall Role Types and Natural remains steady. Consequently, each Natural plays 500 Roles in this example. The evaluation reveals, that the number of queries in the separate queries case dramatically raises to 1001, whereas the number is constant for RSQL and the other relational result processing approach. In contrast, the number of elements in the result is much higher in the all in one example, in fact 500 times higher than in RSQL. Finally, the quotient of transferred data between RSQL and the all in one setting shrinks from 50 in the first evaluation scenario to 5.99 times more transferred data in the second one.

Conclusions

The evaluation on RSQL's RuN in comparison to two different relational result processing models shows that RuN combines the positive aspects of the relational processing models in a single one. As the evaluation on the varying Role Type amount clearly shows, RuN requires only one statement, fewer lines of code, and avoids a combinatorial explosion of the result. In contrast to the all in one approach in a relational processing, which is mainly based on conditional statements, RSQL iterates over the corresponding Roles to collect their information. Moreover, the second comparison is focused on varying the number of Roles played by a certain Natural. This rating illustrates that RuN transfers less data from the database system to the applications than the SQL processing models. In sum, we prove that RuN is an efficient result representation for Dynamic Tuples and combines the positive aspects of two extreme relational processing models.

6.5 SUMMARY

In the beginning of this chapter we described a prototypical implementation of RSQL on the basis of the H2 database engine. Moreover, we presented a relational mapping of our role-based metamodel and applied it on the university scenario. This mapping built the basis for the evaluation of RSQL in contrast to SQL. In detail, we evaluated the database models, the query languages as well as the result representations. Precisely, we showed that RSQL requires fewer statements to set up a role-based database schema than SQL. In the investigated scenarios the relative difference ranged from 3 times more SQL than RSQL statements in the most basic scenario, up to 15 times more SQL statements. Most important, the consistency constraints are natively ensured by RSQL, whereas they have to be implemented manually in the relational setting. Especially, the amount of triggers required to ensure instance consistency increases rapidly, which lowers the schema maintainability. In particular, the relational mapping required up to 15 times more statements than RSQL in the investigated scenarios. The comparison of the query language against the corresponding schemata revealed that RSQL demands approximately 2.5 times less words and characters to express the query. Moreover, the SQL statements look very homogeneous, in a negative sense, in their visual representation, which complicates debugging. Additionally, SQL queries mix mapping details with entity information. The evaluation on RuN against two relational result processing models demonstrated its adequacy as role-based result representation. Moreover, its functionality combines the positive aspects of both relational processing models. In total, this evaluation clearly demonstrated the benefits of fully implementing the role notion and its accompanied metatype distinction in a database system against a relational mapping of it. This holds for the investigated scenarios in which the software system manages role-based dynamically evolving entities.

CONCLUSIONS

Today, software has become ubiquitous. In particular, it is ubiquitous in physical space on wearables and smartphones, ubiquitous in logical space by Internet-based applications, and finally ubiquitous in time according to longevity. As a consequence of this ubiquity, software acts in frequently changing contexts. Traditional modeling and programming languages, especially the object-oriented ones, do not feature an explicit notion of context, hence, the context-awareness is simulated and manually implemented. This simulation mixes regular entity behavior with context-adaptation procedures in the same code.

To cope with challenges posed from ubiquitous software systems, research proposed several approaches, including the concept of roles. The idea of roles is to extract the context-dependent behavior from the entity and model it in a separate type. This enables a separation of concerns within entities. In particular, the core behavior and structure is defined in the entity type, and all context-dependent and fluent parts are specified in role types. Moreover, entities are able to start and stop playing roles to adapt their behavior and structure dynamically during runtime, without the need for reinstantiation. This concept is established in modeling and programming languages, but not in database systems.

A database system is an integral component of today's software system and provides standard functionalities for the persistent data management to applications. Moreover, it guarantees global data consistency with respect to a given schema. These guarantees have to be provided in a ubiquitous, especially role-based, software environment as well. Unfortunately, there is neither a database system nor a role-based database model available that implements roles as first class citizen. Caused by this lack of an explicit role notion in the database system, the role-relational impedance mismatch arises. This mismatch describes the problems of applications and their developers, the database system itself, and the software system in general in a role-based software system setup with a relational database system as central data storage. First, this mismatch describes the inability of a database system to ensure global consistency, because it cannot directly reflect the consistency constraints. Thus, some data management tasks move towards the applications, causing additional effort in programming and maintaining the applications. Moreover, the software system does not feature a clearly layered structure, because data management activities are performed on several layers.

7.1 THESIS CONCLUSIONS

To overcome the role-relational mismatch, an integration of the entity's separation of concerns in the database system is required. In detail, the metatype distinction between the entity core, its playable role types and the contexts these role types are situated in.

Firstly, we put this thesis in the context of roles, by explaining the concept of roles and its ability to enable a separation of concerns on the level of an entity, in Chapter 2. This builds the foundation for the discussion on the need for a role-based database system as described in Chapter 3. Moreover, we detailed the **role-relational impedance mismatch** and explained its consequences on the software developers, the database system itself, and the software system in general. Based on this knowledge, we specified several requirements to overcome it and evaluated several architecture approaches with respect to these requirements. As it turned out, only a full database system integration of the role-based semantics helps to overcome this mismatch. To introduce such role-based semantics in a database system, we proposed adaptations on set-oriented interface as well as on data system. These adaptations are concentrated under the RSQL approach, which consists of the following key contributions.

RSQL Database Model We introduced and defined the **RSQL database model** as logical foundation for the data system adaptation in Chapter 4. This encapsulates the role-based semantics in **Dynamic Data Types** on the scheme level and **Dynamic Tuples** on the instance level. Precisely, Dynamic Data Types encompass the notion of an entity core type and Role Types in two dimensions, the filling dimension that includes Role Types that can be filled by a core, and the participating dimension that describes these Role Types that are contained in a Compartment Type. This specifies which core may acquire which additional structure during runtime. In contrast, Dynamic Tuples are the instance representation and consist of an entity core and Roles in two dimensions. This explicitly states, which entity currently plays and features which Roles. These main data structures were augmented by an explicit notion of Relationship Types and Relationships, respectively. In addition, we defined several **formal operators** to give the database model a processing model.

RSQL Query Language and Processing On the basis of the proposed database model, we defined a formal syntax to create Dynamic Data Types as well as inserting and extending Dynamic Tuples in Chapter 5. This represents the adaptation of the set-oriented interface with respect to the novel database model, from an input perspective. As described, this language has three parts, the **data definition language** to create the role-based database schema, the **data manipulation language** to populate the database, and a **data query language** to retrieve data from the database. Moreover, we connected the query language's syntax elements to the formal operators defined in the database model. On these foundations, we discussed **options for a query processing** of Dynamic Tuples.

Result Representation To adapt the set-oriented interface from an output perspective, we introduced the **RSQL Result Net**. It encompasses the query output, consisting of various Dynamic Tuples, in result groups, one group for each queried Dynamic Data Type. To access the Dynamic Tuple internal information, for instance a role of a certain Role Type, RuN described several **endogenous navigation paths**. In contrast, to navigate from a Role to the Dynamic it is played or featured in, we discussed the **exogenous navigation paths**.

Finally, Chapter 6 provided a proof of concept. In detail, RSQL's database model, the query language, and result representation were evaluated against a relational representation of the assumed role notion. It turned out, RSQL is able to **ensure global consistency constraints** for a role-based database schema without the need to implement consistency constraints in triggers. Moreover, we showed that the **RSQL query language uses fewer words** and characters to retrieve equivalent data from the corresponding database. As final evaluation, we compared a RuN-based result processing with two possible relation processing models and showed that **RuN combines the positive aspects** of both relational models.

In total, the RSQL approach, as it has been defined and discussed in this thesis, overcomes the role-relational impedance mismatch by introducing the notion of roles and its accompanied metatype distinction within a database system. As result, the role-based semantics and constraints are directly represented in the database model, the query language, and result representation, which brings the database system in the position to ensure global consistency for role-based software systems. This brings it back in its rightful position as single point of truth within such a software system. Moreover, the software system is structured more clearly, because the persistent data management tasks are performed by the database system only, which also takes load and implementation tasks from the application developers.

7.2 FUTURE WORK

Software systems become more complex and so the entities do. To keep control over the data management of entities that frequently change their structure and behavior according to context switches, it is necessary to reflect the accompanied semantics in the database system. The RSQL approach is a first step that introduces the metatype distinction, on the basis of roles, as separation of concerns in the database system. RSQL's foundation is its logical database model consisting of Dynamic Data Types and Dynamic Tuple. As every novel data model, RSQL opens a wide space for future work, especially for its implementation and its extension.

Optimizations The proposed database model neither considers logical nor physical optimizations. On the side of the **logical optimization**, it is possible to optimize the operator plans be rearrange the operators to minimize the intermediate results between several operators. As the outlined query processing requires multiple executions of the operator plan, as long as the results differ, it would be an option to minimize these runs by a smart operate arrangement or by giving the operators additional information.

On the physical side, the RSQL's logical database model does not state anything about its physical implementation. Hence, it is possible to optimize the physical storage for several workload scenarios, for instance, read intensive workloads on Roles or frequently traversed navigation paths between Dynamic Tuples. Moreover, various physical implementations of an operator are possible, like it is in relational database systems the feature several physical join-operator implementations. Depending on the query, a different physical operator could be chosen. In addition, an indexing structure adapted to Dynamic Tuples is required to speed up the data access.

Operators RSQL features a set of various operators that enable processing, manipulation and filtering of Dynamic Tuples. This set could be extended to provide functionalities like **aggregation functions**. This requires to consider the effects of these function on the database model. For instance, assume a count function that is applied on Roles. It is not defined whether the counting result is represented as new Natural in a separate output stream, or as Role within a given Dynamic Tuple, or even as attribute of the Natural or Role. Moreover, such functionality requires a novel construction infrastructure for Dynamic Tuples, because so far the base Dynamic Tuples are filtered and manipulated, but newly constructed ones are not considered.

Metatypes The database model uses Roles Types as only metatype to express the fluent parts of an entity. In contrast, the real world provides several examples for **other metatypes**. This could extend the expressiveness of the database model in regard to a more exact real world modeling. For instance, being a father is not essential to be a person, hence, it is not part of the core type, but it is also not a Role Type because Roles may be abandoned. In contrast, once the fatherhood starts, it never ends. This could be modeled in a separate metatype *status*, such that an instance of it becomes part of the entity's core, once it is acquired.

Additionally, several **role-specific constraints** are possible. For instance, mutual exclusions (between Professor and Student), implications (when StudentAssistant then also Student), or an order of Role acquisitions. By using ordered role acquisitions, a process modeling within the database would be enabled by using roles. This would also add a temporal dimension to roles as modeling concept.

Client Side Support The traditional interaction schema between a database system and applications is query to result, and this to the result processing. This model introduces an imaginary line between both and the communication is triggered by the application at that point it executes a query. This line could be redesigned by introducing a **novel interaction schema**. For instance, the current interaction requires an application to explicitly trigger the data retrieval process to the database by a query as well as the persistence process by updating or inserting data. In case of Roles this could be redesigned by loading and storing roles as soon as the entity or software changes its context. In detail, the runtime environment or any other kind of infrastructure the application runs on, recognizes a context change and provides this context information to the database, which provides the required Roles to the infrastructure. Additionally, this infrastructure initializes the Roles and merge them into the runtime object. This process does not require an explicit action on the application side, in fact the information is automatically provided in the background. Moreover, such an interaction design would remove explicit control over the data persistence and load process from the applications, which is also performed by traditional object-relational mapping engines.

BIBLIOGRAPHY

[1] Gregory Abowd, Anind Dey, Peter Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a Better Understanding of Context and Context-awareness. In *International Symposium on Handheld and Ubiquitous Computing*, pages 304–307. Springer, 1999.

[2] Antonio Albano, Roberto Bergamini, Giorgio Ghelli, and Renzo Orsini. An Object Data Model With Roles. In *VLDB*, volume 93, pages 39–51, 1993.

[3] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems (TODS)*, 10(2):230–260, 1985.

[4] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. Fibonacci: A Programming Language for Object Databases. *The VLDB Journal*, 4(3):403–444, 1995.

[5] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computer Surveys*, 40(1):1:1–1:39, February 2008.

[6] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented Programming with Java. *Information and Media Technologies*, 6(2):399–419, 2011.

[7] Charles Bachman. The Programmer as Navigator. *Communications of the ACM*, 16(11):653–658, 1973.

[8] Charles Bachman and Manilal Daya. The Role Concept in Data Models. In *VLDB*, pages 464–476. VLDB Endowment, 1977.

[9] Stephanie Balzer. *Rumer: A Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH Zürich, 2011.

[10] Stephanie Balzer, Thomas R Gross, and Patrick Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *European Conference on Object-Oriented Programming*, pages 323–346. Springer, 2007.

[11] Jörg Baumgart. *Analyse, Entwurf und Generierung von Rollen-und Variantenmodellen*. PhD thesis, TU Darmstadt, 2003.

[12] Margit Becher. *XML: DTD, XML-Schema, XPath, XQuery, XSLT, XSL-FO, SAX, DOM*. W3L-Verlag, 2009.

[13] Anthony Bloesch and Terry Halpin. ConQuer: A Conceptual Query Language. In *International Conference on Conceptual Modeling*, pages 121–133. Springer, 1996.

[14] Anthony Bloesch and Terry Halpin. Conceptual Queries using ConQuer-II. In *Proceedings of the International Conference on Conceptual Modeling*, pages 113–126. Springer, 1997.

[15] Oliver Böhm. *Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP*. dpunkt-Verlag, 2006.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4, 2008.

[17] Liu Chen and Ting Yu. A Semantic DBMS Prototype. In *Advances in Conceptual Modeling: ER 2013 Workshops*, pages 257–266. Springer International, 2014.

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

[19] Edgar Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[20] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. A generic role model for dynamic objects. In *International Conference on Advanced Information Systems Engineering*, pages 643–658. Springer, 2002.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[22] Suzanne Dietrich and Susan Urban. *An Advanced Course in Database Systems: Beyond Relational Databases*. An Alan R. Apt book. Pearson/Prentice Hall, 2005.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[24] Valerio Genovese. A Meta-Model for Roles: Introducing Sessions. In *Proceedings of the 2nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies*, pages 27–38, 2007.

[25] Giorgio Ghelli. Foundations for Extensible Objects With Roles. *Information and Computation*, 175(1):50–75, 2002.

[26] Sebastian Götz. Dampf - Dresden Auto-Managed Persistence Framework. Diploma thesis, Technische Universität Dresden, 2010.

[27] Sebastian Götz and Thomas Kühn. Models@run.time for Object-Relational Mapping Supporting Schema Evolution. In *10th International Workshop on Models@run.time (MRT15)*, 2015.

[28] Kasper Graversen. Explaining the Implementation of Chameleon – A Short Overview. Technical report, Unversity of Copenhagen, 2003. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.594&rep=rep1&type=pdf` ; accessed 07-January-2017.

[29] Kasper Graversen and Kasper Østerbye. Implementation of a Role Language for Object-specific Dynamic Separation of Concerns. In *Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[30] RoSI Research Training Group. Role-based software Infratructures for continious-context-sensitive Systems. http://www.rosi-project.org, july 2016. [Online; accessed 25-July-2016].

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

[32] Nicola Guarino, Massimiliano Carrara, and Pierdaniele Giaretta. An Ontology of Meta-Level Categories. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 270–280. Morgan Kaufmann, 1994.

[33] Terry Halpin. Conceptual Queries. *Database Newsletter*, 26(2), 1998.

[34] Terry Halpin. Object-Role Modeling (ORM/NIAM). In *Handbook on Architectures of Information Systems*, pages 81–103. Springer, 1998.

[35] Terry Halpin. ORM 2. In *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, pages 676–687. Springer, 2005.

[36] Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. Rava: Designing a Java Extension With Dynamic Object Roles. In *International Symposium and Workshop on Engineering of Computer-Based Systems*, pages 7–pp. IEEE, 2006.

[37] Rolf Hennicker and Annabelle Klarl. Foundations for Ensemble Modeling–The Helena Approach. In *Specification, Algebra, and Software*, pages 359–381. Springer, 2014.

[38] Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. CoDEL–A Relationally Complete Language for Database Evolution. In *East European Conference on Advances in Databases and Information Systems*, pages 63–76. Springer International Publishing, 2015.

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

[40] Stephan 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.

[41] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3), 2008.

[42] 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.

[43] Jie Hu, Qingchuan Fu, and Mengchi Liu. Query Processing in INM Database System. In *Web-Age Information Management*, pages 525–536. Springer, 2010.

[44] Jie Hu and Mengchi Liu. Modeling Context-dependent Information. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 1669–1672, New York, NY, USA, 2009. ACM.

[45] International Organization for Standardization. Information processing systems – Database language – SQL. Standard, International Organization for Standardization, 1987. http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=16661 ; accessed 01-September-2016.

[46] International Organization for Standardization. Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation). Standard, International Organization for Standardization, 2011. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53682` ; accessed 01-September-2016.

[47] ISO/IEC. ISO/IEC 9075-x:200x (Information technology – Database languages – SQL), 2006.

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

[49] Tobias Jäkel, Thomas Kühn, Stefan Hinkel, Hannes Voigt, and Wolfgang Lehner. Relationships for Dynamic Data Types in RSQL. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2015.

[50] 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, 2014.

[51] Tobias Jäkel, Thomas Kühn, Hannes Voigt, and Wolfgang Lehner. Towards a Role-Based Contextual Database. In *20th East European Conference on Advances in Databases and Information Systems*, pages 89–103. Springer International Publishing, 2016.

[52] Donald A. Jardine, editor. *The ANSI/SPARC DBMS model : proceedings of the second SHARE Working Conference on Data Base Management Systems, Montreal, Canada, April 26-30, 1976*, 2nd, Montréal, Québec, 1977. North-Holland Pub. Co. New York.

[53] Tetsuo Kamina and Tetsuo Tamai. Towards Safe and Flexible Object Adaptation. In *International Workshop on Context-Oriented Programming*, page 4. ACM, 2009.

[54] Alfons Kemper and Andre Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg, 2006.

[55] Alfons Kemper and Andre Eickler. *Datenbanksysteme: Eine Einführung*. De Gruyter Studium. Gruyter, Walter de GmbH, 2015.

[56] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. A Combined Formal Model for Relational Context-Dependent Roles (Extended). Technical report, Technische Universität Dresden, 2015.

[57] Thomas Kühn, Stephan Böhme, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Combined Formal Model for Relational Context-Dependent Roles. In *International Conference on Software Language Engineering*, pages 113–124. ACM, 2015.

[58] 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*, pages 141–160. Springer, 2014.

[59] Max Leuthäuser. SCROLL - A Scala-based library for Roles at Runtime. In *Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015)*, pages 7–8, 2015.

[60] Max Leuthäuser and Uwe Aßmann. Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-based Programming. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, MORSE/VAO '15, pages 25–33. ACM, 2015.

[61] Mengchi Liu and Jie Hu. Information Networking Model. In *International Conference on Conceptual Modeling*, pages 131–144. Springer, 2009.

[62] 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, 2009.

[63] Mengchi Liu, Jie Hu, Liu Chen, and Xuhui Li. Representing Hierarchical Relationships in INM. In *International Conference on Conceptual Modeling*, pages 297–304. Springer, 2014.

[64] Riichiro Mizoguchi, Kouji Kozaki, and Yoshinobu Kitamura. Ontological Analyses of Roles. In *Federated Conference on Computer Science and Information Systems*, pages 489–496. IEEE, 2012.

[65] Eila Niemelä and Juhani Latvakoski. Survey of Requirements and Solutions for Ubiquitous Software. In *Proceedings of the 3rd international Conference on Mobile and Ubiquitous Multimedia*, pages 71–78. ACM, 2004.

[66] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[67] Olaf Otto. Entwicklung einer Persistenzlösung für Object Teams auf Basis der Java Persistence API. Diploma thesis, Technische Universität Berlin, 2009. `http://objectteams.org/publications/Diplom_Olaf_Otto.pdf` ; accessed 01-September-2016.

[68] Michael Pradel and Martin Odersky. Scala Roles: Reusable Object Collaborations in a Library. In *Software and Data Technologies*, pages 23–36. Springer, 2009.

[69] Benjamin Rosenzweig and Elena Rakhimov. *Oracle PL/SQL by Example*. Prentice Hall Professional Oracle Series. Pearson Education, 2008.

[70] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. The Addison-Wesley object technology series. Addison Wesley, 2010.

[71] Gunter Saake, Andres Heuer, and Kai-Uwe Sattler. *Datenbanken: Konzepte und Sprachen*. mitp, 2008.

[72] Gunter Saake, Kai-Uwe Sattler, and Andres Heuer. *Datenbanken: Konzepte und Sprachen*. mitp, 2005.

[73] Ravi Sandhu, Edward Coynek, Hal Feinsteink, and Charles Youmank. Role-based Access Control Models. *IEEE computer*, 29(2):38–47, 1996.

[74] Chandra Sekharaiah and Janaki Ram. Object Schizophrenia Problem in Object Role System Design. In *International Conference on Object-Oriented Information Systems*, pages 494–506. Springer, 2002.

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

[76] Ian Sommerville. *Software Engineering*. International computer science series. Addison-Wesley, 2007.

[77] Xiaopu Song. *Information Network Model Query Processing*. PhD thesis, Carleton University Ottawa, 2010.

[78] Friedrich Steimann. *Formale Modellierung mit Rollen*. Habilitation thesis, Universität Hannover, 2000.

[79] Friedrich Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data & Knowledge Engineering*, 35(1):83–106, October 2000.

[80] Friedrich Steimann. The Role Data Model Revisited. *Applied Ontology*, 2:89–103, 2007.

[81] Robert Taylor and Randall Frank. CODASYL Data-Base Management Systems. *ACM Computer Surveys*, 8(1):67–103, March 1976.

[82] Rik Van Bruggen. *Learning Neo4j*. Community Experience Distilled. Packt Publishing, 2014.

[83] Roel Wieringa, Wiebren de Jonge, and Paul Spruit. *Roles and Dynamic Subclasses: A Modal Logic Approach*, pages 32–59. Springer, 1994.

[84] Raymond Wong, Lewis Chau, and Frederick Lochovsky. DOOR: A Dynamic Object-oriented Data Model With Roles. Technical report, Hong Kong University of Science and Technology, 1996.

[85] Raymond Wong, Lewis Chau, and Frederick Lochovsky. The Roles and Views of Multimedia Objects. Technical report, Hong Kong University of Science and Technology, 1996.

[86] Raymond Wong, Lewis Chau, and Frederick Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. In *International Conference on Data Engineering*, pages 402–411. IEEE, Apr 1997.

[87] Raymond Wong, Lewis Chau, and Frederick Lochovsky. Dynamic Knowledge Representation in DOOR. In *Proceedings of Knowledge and Data Engineering Exchange Workshop*, pages 89–96, Nov 1997.

List of Tables

List of Figures