The Database Normalization Theory and the Theory of Normalized Systems: Finding a Common Ground

Article · February 2016

CITATIONS
15

READS
31,774

1 author:

Erki Eessaar
Tallinn University of Technology
52 PUBLICATIONS **113** CITATIONS

SEE PROFILE

Baltic J. Modern Computing, Vol. 4 (2016), No. 1, 5-33

The Database Normalization Theory and the Theory of Normalized Systems: Finding a Common Ground

Erki EESSAAR

Department of Informatics, Tallinn University of Technology,
Akadeemia tee 15A, 12618 Tallinn, Estonia

Erki.Eessaar@ttu.ee

Abstract: Database normalization theory offers formalized guidelines how to reduce data redundancy and thus problems that it causes in databases. More lately, researchers have started to formalize ideas that the problems caused by unwanted dependencies and redundancy can be observed in case of any modular system like software, hardware, or organization. To tackle these problems, they have proposed the theory of normalized systems that complete application frees systems of combinatorial effects and thus the impact of a change in a system does not depend on the system size. At the time of writing this paper, the explanations of the theory of normalized systems do not say much about database normalization. We think that the theories are deeply related. In this paper, we search a common ground of the database normalization theory and the theory of normalized systems.

Keywords: database normalization, principle of orthogonal design, normalized system (NS), evolvability, separation of concerns, combinatorial effect (CE).

1. Introduction and Related Works

Normalization theory of relational databases dates back to the E.F. Codd's first seminal papers about the relational data model (Codd, 1970). Since then it has been extended a lot (see, for instance, Date (2007, Chap. 8)) and the work is ongoing. There are proposals how to apply similar principles in case of other data models like object-oriented data model (Merunka et al., 2009), (hierarchical) XML data model (Lv et al., 2004), or (hierarchical) document data model (Kanade et al., 2014). Database normalization process helps database developers to *reduce* (not to eliminate) data redundancy and thus avoid certain update anomalies that appear because there are combinatorial effects (CEs) between propositions that are recorded in a database. For instance, in case of SQL databases each row in a base table (table in short) represents a true proposition about some portion of the world. CEs mean in this context that inserting, updating, or deleting one proposition requires insertion, update, or deletion of additional propositions in the same table or other tables. The more there are recorded propositions, the more a data manager (human and/or software system) has to make this kind of operations in order to keep the data consistent. Thus, the amount of work needed depends on the data size and increases over time as the data size increases. Failing to make all the needed updates leads to inconsistencies. The update anomalies within a table appear because of certain dependencies between columns of the same table. Vincent (1998) shows how these

dependencies lead to data redundancy. Informally speaking, these anomalies appear if different sets of columns of the same table contain data about different types of real-world entities and their relationships. By rewording a part of third normal form definition of Merunka et al. (2009), we can say that these sets of columns have independent interpretation in the modeled system. According to the terminology in (Panchenko, 2012) these sets of columns have different themes. Thus, the table does not completely follow the *separation of concerns principle* because database designers have not separated sets of columns with independent interpretations into different software elements (tables in this case). The update anomalies across different tables within a database may occur because of careless structuring of the database so that one may have to record the same propositions in multiple tables. The update anomalies across different databases (that may or may not constitute a distributed database) may occur if one has designed the system architecture in a manner that forces duplication of data to different databases.

Conceptually similar update (change) anomalies could appear in the functionality of any system or its specification and these make it more difficult and costly to make changes in the system or its specification. Pizka and Deissenböck (2007) comment that redundancy is a main cost driver in software maintenance. The need to deal with the update anomalies in the systems that are not designed to prevent them is inevitable because the systems have to evolve due to changing requirements just like the value of a database variable changes over time. For instance, some of the changes are caused by the changes in the business, legal, or technical environment where the system has to operate, some by changing goals of the organization, and some by the improved understanding of the system domain and requirements by its stakeholders. The theory of normalized systems (NS) (Mannaert et al., 2012b) reflects understanding of the dangers of the update anomalies and offers four formalized design theorems that complete application helps developers to achieve systems that are free of CEs and are thus modular, highly evolvable, and extensible.

The NS theory speaks about modules and submodular tasks. The work with the NS theory started after the invention of the database normalization theory. Its proponents see it as a general theory that applies to all kinds of systems like software, hardware, information system, or organization or specifications of these systems. Like the database normalization theory, its goal is to improve the design of systems and facilitate their evolution. In our view, it would be useful to bring these two theories together to be able to understand their similarities and differences. Possibly, we can use the ideas that have been worked out for one theory in case of the other theory as well. Nowadays there is a lot of talk about object-relational impedance mismatch between highly normalized relational or SQL databases and object-oriented applications that use these. Thus, researchers and developers look these as quite distinct domains that require different skills and knowledge as well as have different associated problems, theories, methods, languages, and tools. Hence, in addition to technical impedance mismatch there is a mental one as well. We support the view that database design *is* programming and has the same challenges as the programming in the "traditional" sense like ensuring quality and high evolvability, separating concerns, managing redundancy and making redundancy controlled, testing the results, versioning, and creating tools that simplify all this.

Database and application developers sometimes have antagonistic views to the normalization topic. Merunka et al. (2009) mention a common myth in object-oriented development community that any normalization is not needed. Komlodi (2000)

compares object-oriented and relational view of data based on the example of storing a virtual car in a garage. He compares a design that offers a large set of highly normalized tables with an object-oriented design where there is class *Car* that describes complex internal structure and behavior of car objects. Readers may easily get an impression that normalization is something that one uses in case of databases but not in case of object-oriented software.

On the other hand, there are ideas of using the relational model, relational database normalization theory, and dependency theory, which is the basis of the normalization theory, to facilitate understanding of evolving systems. De Vos (2014) uses the relational model as a metalanguage and the relational database normalization theory as a theoretical tool to explain and predict language evolution in terms of gradual lexicon development as well as explain the levels of language ability of animals. We have found the work of Raymond and Tompa (1992), Lodhi and Mehdi (2003), and Pizka (2005) that apply the dependency theory to the software engineering. Raymond and Tompa (1992) analyze text editor and spreadsheet software. They describe functionality in terms of tables, investigate dependencies between the columns, and discuss implications of the dependencies to the design of data structures and software as well as end-user experience. They show how decomposing the tables along the dependencies, based on the rules of database normalization, reduces redundancy in software design and thus makes it easier to update the software. They suggest that it would be possible to teach object-oriented design in terms of multivalued dependencies. The authors note that users could tolerate certain amount of data redundancy but the goal to ensure data consistency leads to software that is more complex. Having different approaches for dealing with redundancy within the same software may reduce its usability. Lodhi and Mehdi (2003) describe and illustrate the process of applying normalization rules to the classes of object-oriented design. Pizka (2005) considers maintainability of software and discusses difficulties of maintaining code due to change anomalies, which are conceptually similar to the update anomalies in not fully normalized relational databases. He transfers the idea of normalization from data to code and defines two code normal forms in terms of semantic units and semantic dependencies. In principle, there is such dependency between program units (for instance, functions), if these units are equivalent or semantically equivalent. The latter could mean that the operations fulfill the same task but perform their task based on differently represented input data. He uses the defined normal forms for reasoning about, finding, and removing change anomalies in code to improve its maintainability. However, none of these ideas has achieved widespread attention. In October 2015, the paper (Raymond and Tompa, 1992) had six, the paper (Lodhi and Mehdi, 2003) had one, and the paper (Pizka, 2005) had two papers that referred to it according to the Google Scholar™. None of these references has the topic of the referenced papers as its main topic.

Software systems contain a layer that implements business logic, which is guided by the business rules. It is possible to represent these rules in decision tables. The works of Vanthienen and Snoeck (1993) as well as Halle and Goldberg (2010) are examples of research about normalizing decision tables to improve their understandability and maintainability. They derive the normalization process from the database normalization process and define different normal forms of business rules. Halle and Goldberg (2010) comment that the normalization leads to a decision model structure that causes the removal of duplicate atomic statements and delivers semantically correct, consistent, and complete rules.

In case of the database normalization theory there is a well-known technique of denormalization that requires reduction of normalization level of one or more tables (if we speak about SQL databases) to achieve pragmatic goals like better performance of some queries in a particular environment. Whether and how much to use it depends on context. For instance, Helland (2011) gives an impression that that in large-scale systems do not need database normalization because users add new facts instead of updating existing facts. Kolahi and Libkin (2010) provide formal justification that databases with good third normal form design (there are higher normal forms) offer the best balance between redundancy reduction and efficiency of query answering. Grillenberger and Romeike (2014) argue that computer science has to "rethink the idea of redundancy from something that generally should be avoided to a method that is applied in order to achieve certain specific goals" due to the emergence of NoSQL systems and Big Data. One starts to wonder, does it mean that these ideas also apply to the redundancy of functionality.

Perhaps understanding universalness of normalization principles, similarity of concepts, potential problems of redundancy, and situations when redundancy is tolerable or even desirable helps us also reduce the *mental gap* between the application development and database development domain. Fotache (2006) observes that database normalization theory has failed to become universal practical guide of designing relational databases and points to different reasons of that. Badia and Lemire (2011) mediate a report that in a typical Fortune 100 company database normalization theory is not used. There are even such provocative calls like "normalization is for sissies" (Helland, 2009). If one understands better the relationship between the NS theory and the database normalization theory, then one can learn from the problems of one theory how to make things better in case of the other. Badia and Lemire (2011) observe with regret that traditional database design (thus, also normalization) "is not a mainstream research topic any more" and is often considered "a solved problem." One could hope that the interest towards the NS theory will also increase the interest towards the database normalization. Grillenberger and Romeike (2014) suggest that at the age of Big Data the topic of strictly normalizing data schema is not any more a broadly influential fundamental database concept that deserves teaching in the general education. They also call for discussion as to whether (data) redundancy is such general concept any more. However, understanding fundamental similarities between the NS theory and the database normalization theory would strengthen the understanding that *redundancy* is an important concept of both *data* and *management*. It influences systems in general and thus certainly deserves teaching.

Thus, the *goal* of the paper is to bring together and search a common ground of the theory of NS and the theory of database normalization. We pointed to some observable similarities and differences between the theories in our earlier paper (Eessaar, 2014). However, the topic deserves a deeper analysis. To our knowledge, there has not been this kind of analysis yet in the literature. Since the NS theory is independent of any programming language, platform, or technology, we want to understand as to whether the theory of database normalization could be seen as a specialization of the NS theory in the domain of databases.

We organize the rest of the paper as follows. Firstly, we present an additional explanation of the main principles and reasons behind database normalization and the NS theory. After that, we explore what existing NS literature says about databases and the database normalization theory. Thirdly, we explore a common ground of the normalization theories. We name some problems of SQL databases and database

management systems (DBMS) that are caused by the insufficient separation of concerns, which is a violation of the NS theory. We present a conceptual model that offers a unified view of the theories. Finally, we conclude and point to the further work with the current topic.

2. Normalization Theories

Here, we further explore the theories to explain the context and improve understanding.

2.1. Theory of Database Normalization

The normalization theory provides a formalized theoretical basis for the structuring of databases. The goal of its application is to *reduce* data redundancy in databases and hence avoid certain update anomalies. Although this is desirable in case of any data model, we refer to the relational data model in our discussions because the theory is probably the best known in this domain. Of course, one may choose to allow redundancy for the sake of improving some other aspects of the system but in this case it is a conscious design decision and one must take into account all of its good and bad results. Technical limitations of data management platforms often cause decisions to permit certain degree of data redundancy. Because of these, allowing the redundancy is the best possible way to speed up certain queries or, in case of data models that do not provide an operation that is similar to relational join, to make possible certain operations in the first place.

Our understanding of the relational data model, which is also the basis of the SQL database language, is based on The Third Manifesto (Date and Darwen, 2006). It differs from the underlying data model of SQL in many crucial details. However, in the discussions of the database normalization theory, we will use the terminology of SQL – table, column, and row. We will do it because SQL is well known and we hope that it will make the discussion more understandable. We will point to the differences of the underlying data model of SQL and the relational model where we need it.

Data redundancy in a database means that there is at least one proposition that has two or more distinct representations in the database (two or more separately registered propositions) (Date, 2006a). The update anomalies make data modification operations more time consuming and error prone. If one wants to insert, update, or delete a proposition, then the DBMS or its invoker have to make more work (how much more depends on the level of normalization and the number of already registered propositions) to complete the operation and to ensure consistency of propositions. If the system design determines that applications that use data have to be aware of the redundancy and ensure consistency, then it increases coupling between the applications and the database. In addition, data structures that feature data redundancy and update anomalies restrict propositions that one can record in the database because the system treats independent propositions at the database level as dependent propositions. Carver and Halpin (2008) note that NULLs in a fact (proposition) indicate that the fact is not atomic. Contrary to SQL, The Third Manifesto does not permit us to use NULLs to depict missing values. Thus, one cannot bundle together propositions to a row as a composite proposition if some parts of the composite proposition are missing.

Normalization is a multi-step process where each step takes the involved tables to a higher normal form that is defined in terms of certain well-formedness rules. In other

words, the process describes how to evolve the database schema. To take a table T to first normal form, one must be able to represent data in T so that in each row of T each field contains exactly one value that belongs to the type of the corresponding column (and not sets of such values known as *repeating groups*). Moreover, the table also has to have at least one (candidate) key to exclude duplicate rows because the table must satisfy all the requirements of the relational model to tables. It is said that each table that is in first normal form is normalized. In this paper, further database normalization means *projection-based decomposition* of tables. Thus, further normalization of T means searching certain dependencies between the columns of T and decomposition of T into smaller tables in a nonloss manner based on the dependencies by using projection operation. Further normalization process is often informally called normalization. Nonloss decomposition of a table T means in this case that a database designer replaces T with certain of its projections so that it is guaranteed that one can restore the original table T by joining all the projections and all the projections are needed to provide such guarantee. The process recursively applies also to all the newly created tables.

As a by-product, the result of the normalization process simplifies enforcement of certain integrity constraints because we can now enforce these by simply declaring keys to the tables. Enforcement of the integrity constraints means that the system has more information to check consistency of data and optimize operations. Another by-product is that the observer of the schema that contains tables will get gradually better understanding of the concepts and relationships of the real world (domain) that are reflected by the schema. Tables reflect more and more fine-grained concepts of the real world and declared keys and referential constraints explain the nature of relationships. De Vos (2014) draws parallel between normalization of tables and gradual expansion of lexicon to depict new concepts. Depending on the implementation of a particular DBMS, the normalization process could lead to the reduction of data storage costs due to the reduced redundancy.

The reverse process of normalization is denormalization. Database designers use it to achieve pragmatic goals like improved performance of certain queries or offering data to applications in an aggregate form so that it is easier for them to read the data. The latter can be achieved in databases by using viewed tables (views in short) (assuming that the DBMS supports them and supports operators needed to convert data to aggregate form) and thus avoiding the negative effects of data duplication (Burns, 2011).

In general, we can say that the normalization theory teaches us how to improve the design of databases in certain aspects. In addition, its formal and precise definition makes it possible to semi-automate the process and checking existing databases or database design models against its defined levels of normalization. On the other hand, the theory does not cover all the aspects of database design and hence its following does not guarantee a good database design in every aspect. Hence, the normalization theory is only one tool in the toolset of database designers. The application of the database normalization theory cannot even guarantee that a database is free of all kinds of data redundancies. We actually even do not need this because, for instance, duplication of data in a distributed database to several locations helps us to improve availability of the system as well as the speed of answering certain queries. On the other hand, this duplication introduces new level of complexity to the system because it must control the redundancy. Date (2006a) explains that in case of controlled redundancy the DBMS and not its users must take care of propagating (at least eventually) updates to avoid inconsistent or contradictory propositions in different parts of the database. Of course,

someone has to instruct the DBMS how to propagate the updates and thus there is more work for the developers.

The principle of orthogonal design addresses data duplication across multiple tables. It is not a part of the normalization theory. It requires that no two tables in a database should have overlapping meanings (Date, 2006a).

The definitions of normal forms, except first normal form, depend on the existence of keys in the tables, thus eliminating the problem of repeating rows in the tables. However, definition of first normal form does not require the existence of keys in tables according to the interpretation of SQL but does require the existence of at least one key in each table according to the interpretation of the relational model. Keyless tables lead to possible repeating rows in the tables that is another form of redundancy. As Date (2006b, Chap. 10) shows, it leads to, for instance, problems with interpreting the query results as well as optimizing the queries by the system.

Carver and Halpin (2008) argue that the previously described normalization process is inadequate because a table (as a variable) may have different values (sets of rows), some of which do not have multivalued or join dependencies but still have fact-redundancy. They do not question the need of normalization but the process of achieving fully normalized tables. They argue that if one creates a conceptual model that represents atomic fact types, then one can synthesize fully normalized tables from it. They comment that the process of deciding as to whether a fact type is atomic or not requires knowledge about the domain and is informal. Fotache (2006) also points out that alternatively one could use normalization to check the results of deriving database structure from a conceptual model.

2.2. The Theory of Normalized Systems (NS)

Databases are only one, albeit often very important, component of information systems. Intuitively, it is understandable that some design problems that appear in databases can appear in some form in any type of systems. These systems could be technical, sociotechnical, social, or natural. For instance, there could be multiple software modules in a software system that implement the same task, multiple forms in the user interface of the same actor providing access to the same task, multiple process steps, organizational units or organizations that fulfill the same task, or identical or semantically similar models that describe the same tasks. These examples potentially mean unnecessary wasting of resources and more complicated and time-consuming modification of tasks and their models. Being duplicates of each other, the parts have undeclared dependencies, meaning that changing one requires cascading modifications of its duplicates to keep consistency. The more there are such duplicates, the more changes we need to keep consistency.

If there are multiple unrelated or weakly related tasks put together to a module, then it is more difficult to understand, explain, and manage the module. Such modules have more dependencies with each other, meaning that changes in one require examination and possible modifications in a big amount of dependent modules. The less the general information hiding design principle is followed, the more cascading changes are needed. For instance, intuitively, one can understand how difficult it would be to understand places of waste and duplication in a big organization and after that reorganize it. In organizations, the more fine-grained are its tasks, the easier it is to distribute these between different parties and in this way achieve separation of duties and reduce the possibility of fraud.

Observers have noticed the same general problems in case of user interface design as well. Cooper et al. (2014, p. 274) write that navigation is "any action that takes the user to a new part of the interface or that requires him or her locate objects, tools, or data elsewhere in the system" and that the need of such actions should be minimized or eliminated. From the end users perspective, one could see it as a call to denormalize by bundling tasks together to one form or page. From the perspective of the developers, it is a warning of dangers because if there are duplicated tasks in different places, then they have to navigate to modify these. The latter demonstrates conflicting interests of different stakeholders and possibly the need to have different decomposition principles at the different system layers.

Thus, this knowledge is not new and the authors of the NS theory are not its discoverers. For instance, "Once and only once" software development best practice requires that there should not be duplication of behavior in a system (WEB, a). "Don't repeat yourself" best practice is a little bit more relaxed, meaning that each data element or action element (functionality) must have single authoritative representation in the system (Wilson et al., 2014). If there is duplication, then it must be controlled (automated) (WEB, b). Similarly, De Bruyn et al. (2012) refer to different code smells. Many of these indicate code duplication. Their analysis shows that avoidance of most of the smells (14 out of 22) contribute towards achieving NS. However, the NS theory tries to offer more formalized approach how to achieve the system that is free of such problems.

The Lehman's laws of software evolution state that E-type evolutionary software degrades over time unless it is rigorously maintained and adapted as well as its functional content is increased to maintain satisfaction of users (Godfrey and German, 2014). The NS theory assumes unlimited system evolution over unlimited time (Mannaert et al., 2012b). Of course, the stakeholders of the system want the evolution process to be as easy and problem-free as possible. However, this is not the case if the system grows larger and more complex over time. The bigger it gets, the more there are dependencies so that changing one part of the system requires changes in other unrelated parts as the ripple effect. Unfortunately Mannaert et al. (2012b) remain vague about what does "unrelated" mean here. The theory of NS calls such dependencies combinatorial effects (CEs) and calls for their complete elimination. Only if this is achieved, then the impact of change will not depend on the size of the system any more but only on the nature of the change itself. Thus, the system becomes stable with respect to a set of anticipated changes. Mannaert et al. (2012b) define a minimal set of such changes. The theory suggests four prescriptive design theorems to constrain the modular structure of systems and to guarantee that the system is free of CEs and thus highly evolvable. The Lehman's law of increasing complexity states that each E-type software system grows increasingly complex over time, unless stakeholders explicitly work to reduce its complexity (Godfrey and German, 2014). In this case, explicit work means work that is needed during the creation or modification of the system to achieve conformance to the theorems.

The theory is generic in the sense that according to the authors one could apply it to any modular system. To achieve this, the theory is described in terms of very generic primitives like data elements and action elements that configuration forms a system. Anticipated changes of the system mean changes in the configuration of these elements. The design theorems that proofs Mannaert et al. (2012b) present in their paper have the following informal descriptions (Eessaar, 2014).

- *Separation of Concerns* means that each change driver (task, including the use of an external technology) of a system must be put into a separate module.
- *Data Version Transparency* means that there could be multiple versions of data elements without affecting action elements that produce or consume these.
- *Action Version Transparency* means that it must be possible to modify action elements without affecting action elements that call these.
- *Separation of States* means that system has to keep state after every action that belongs to a workflow to be able, for instance, to handle unexpected results.

For instance, there are examples of application of the theory to software (Mannaert et al., 2012b) and business architecture of information systems (Eessaar, 2014). It is possible to apply the theory to system development artefacts like requirements (Verelst et al., 2013) or code. If one uses model driven development to create code based on models by using transformations, then one must consider the theorems right from the first artefacts.

The systems that completely follow all the design theorems are free of CEs and are thus stable in the sense that small initial changes in the system will not lead to big cascading changes as the ripple effect. Thus, the effort to change the system will be similar or even constant over time. The theory calls such systems as *normalized systems*.

The separation of concerns theorem is nicely in line with the code normal forms proposed by Pizka (2005). The requirement of first code normal form that the basic building blocks of code must be indivisible atoms means that each block must have one task. Like the separation of concerns principle, it gives freedom in deciding the granularity of blocks and hence tasks. Akşit et al. (2001) and WEB (c) require that each separated concern must have canonical form property, meaning that it should not include irrelevant and/or redundant abstractions. Similarly, second code normal form requires that there must not be direct or transitive semantic dependencies between blocks, meaning essentially that there should not be blocks that are duplicates of each other.

In databases, a design decision could be to have some level of controlled data redundancy to improve, for instance, performance of some queries due to the technical limitations of the used platform (in this case a DBMS). Similarly, in systems in general the redundancy maybe needed and encouraged in order to achieve, for instance, some level of competition. For instance, in this reason it is useful to have multiple universities in a country. However, in order to achieve controlled redundancy they should ideally have some sort of agreement how to best share the common task to offer good education.

3. Databases According to the NS Literature

One of the questions that interests us is how much the database normalization theory has influenced the NS theory. The reverse process of normalization is denormalization. Therefore, it interests us how the NS theory regards the possibility of not completely enforcing all the theorems of NS and thus not achieving a NS. The database normalization theory stresses the concept of redundancy. Thus, it is interesting to know how the NS theory treats the concept. Finally, we want to know about the treatment of databases in general, DBMSs, and database design according to the NS theory.

To find answers to these questions, we conducted a small literature review. Firstly, we selected the key publication that is a journal paper (Mannaert et al., 2012b), which explains the NS theory and offers proofs of its theorems. Next, we selected the journal paper and all the papers that refer to this paper according to Google Scholar™ (at the

beginning of March 2015) into the initial set of papers. We also added to the initial set the earlier papers of Mannaert and Verelst that the selected journal paper references. For the review, we selected papers (from the initial set of papers) that's main topic is the NS theory or its application to some system and full text is available to us. In total, we reviewed 40 papers from 2006 to 2014 (all in English) that were available as pdf files. We used the following case insensitive search keywords to search parts of the papers that are relevant in terms of the research questions: "normal form", "relational", "sql", "denormalize", "denormalization", "database", "database management system", "update anomaly", "upadate anomalies", "orthogonal design ", "duplicate", "duplication", and "redundancy". In addition, we searched case sensitive word "Codd" as well as "NF" and "DBMS" that are the abbreviations of "normal form" and "database management system", respectively. Next, we summarize and discuss our findings. If there are multiple publications that present similar claims, then we will not present all the publications but make a selection.

Only Verelst et al. (2013) and our previous work (Eessaar, 2014) mention database normalization theory. Verelts et al. (2013) refer to only the paper of Codd (1970) in this regard. They say that it is a well-documented approach how to eliminate *many* CEs in case of databases. They correctly point out that the theory does not eliminate all the effects and thus does not eliminate all the redundancy. This is a crucial difference between the NS theory and the database normalization theory because the former requires us to remove all the CEs. None of the reviewed papers explicitly refers to the concept "denormalization". However, Verelst et al. (2013) speak about the need to eliminate CEs at the software level but relaxing this requirement at the higher levels. This relaxation is nothing else than denormalization in terms of NS. Similarly, in case of databases it is possible to denormalize views that constitute the external level of a database without denormalizing tables based on that the views have been defined (Burns, 2011). In case of software systems, the analogy is, for instance, user interface where each form/page could offer unrelated or weakly related functionality and thus violate the separation of concerns theorem that is one of the founding theorems of the NS theory. In case of documentation, an example are diagrams that could couple unrelated or weakly related model elements. Thus, denormalization is clearly a topic that the NS theory should consider.

Verelst et al. (2013) incorrectly claim that the CEs that the database normalization eliminates are caused by the "redundant definition of attributes." The definitions are not redundant but the attributes are grouped together so that there will be CEs between propositions that are represented by the recorded attribute values. Terminology here is also imprecise because in SQL columns and attributes are structural components of tables and structured types, respectively. Moreover, it is the principle of orthogonal design that addresses redundant definition of columns in different tables. The principle is related to but not a part of the database normalization theory. Only Eessaar (2014) mentions the principle. Only Eessaar (2014) mentions database normal forms and does so while giving an example of similarities of the normalization theories. Thus, we conclude that the database normalization theory has not been an important basis in working out the NS theory and there is a gap in the research, namely search of a common ground of these theories.

There are few mentions of relational databases (three papers), object-relational mapping (one paper), database management systems (two papers), and SQL (three papers) as examples of possible implementation technologies of systems. The NS theory is generic and these are just some possible implementation technologies. Data element is

one of the element types of the NS. According to Mannaert et al. (2012b), one can store instances of data elements in corresponding relational database tables. They look a DBMS as an external technology to applications and do not treat it and its provided data model in terms of the NS theory. The lack of references also shows that the research regarding database normalization has not been an important basis in working out the NS theory.

Seventeen papers refer to databases. Mannaert et al. (2012b) explain that if a transaction fails because, for instance, it violated integrity rules or if a system fails during active transactions, then the system must be aware of all actions that it has performed for the recovery purposes. Thus, the system needs state keeping and that is what the fourth NS theorem prescribes. Mannaert et al. (2012b) do not mention it but many DBMSs implement this by using, for instance, rollback/undo segments. Mannaert et al. (2012b) comment that over time programming languages have evolved to be more consistent with the NS theorems. The design of DBMSs is themselves a subject of the NS theory and the systems implement or facilitate independent implementation of many of the concerns, which separation the NS theory requires. We did not find analysis of DBMSs in terms of how much support they offer in building NS and how it has evolved over time. Thus, in our view, the systems deserve more attention in this regard (see Section 4.1).

At the higher level, applications store in databases the general observable states of real-world systems. Mannaert et al. (2012a) call these states macrostates. This is another manifestation of state keeping, required by the fourth NS theorem.

Some of the reviewed publications mention databases as a part of architecture of information systems. Their authors argue that the architecture would benefit from the application of the NS theory. Mannaert et al. (2012b) note that multi-tier architecture with a separate layer of database logic is a manifestation of the separation of concerns theorem.

In their examples, the papers concentrate to the application layer. Maes et al. (2014) stress that checking based on database as to whether a user has an authorization to use an IT application and its different functions is a separate concern. Developers must implement it in a separate module for the sake of evolvability. Ideally, all the applications will reuse it. Maels et al. (2014) call for reusing such software modules in case of developing new applications. De Bruyn et al. (2014) refer to many different cross-cutting concerns of systems like authorization policy, logging, integrity checking, external communication, and bookkeeping adapter that one should implement in separate modules. Integrity checking is a part of the underlying data model of a DBMS. Others are services that one can build on top of the model in a DBMS. Modern SQL DBMSs provide more or less separation of concerns between the model and the services by providing means to manage the services separately of managing elements determined by the data model.

A goal of the use of the NS theory is to reduce dependencies between modules. Coupling is a measure of such dependencies. Van der Linden et al. (2013) list seven different types of couplings. They name external coupling as the third tightest coupling. In this case, two or more modules communicate by using an external database. Access of this external resource is a concern that all the modules duplicate, meaning that these modules have multiple concerns, which violates the separation of concerns theorem. Fowler (a) calls this kind of approach integration database. The loosest types of couplings are stamp coupling and message coupling in which case modules communicate by passing data structures that they use only partially and messages,

respectively. These approaches are in line with the application database approach (Fowler, b) when each database is controlled and accessed by a single application. The applications share data by exchanging messages via services. In each system, one has to find a balance between different requirements. Although application databases reduce coupling they could lead to data duplication in multiple databases and thus problems in case of performance and ease of use (data needed for some decisions are in multiple places and it takes time to put it together). There could be problems of reusability (data structures in an application database have been created by taking into account requirements of a specific application and not multiple applications that all could need this data) and integrity (difficult to enforce constraints that checking requires reading data from multiple databases; duplication of data in different databases could lead to inconsistencies). In principle, this means CEs because changes in data as well as in the requirements to data mean changes in many parts of the system. Mannaert et al. (2012b) do not see a problem in that because according to them separation of concerns is only about action elements, not data elements.

Authors often use update anomalies to explain and justify database normalization. Similar anomalies exist in systems in general if updating one part of a system causes the need to update other parts as the ripple effect and the need increases with the increasing of the system size. However, only Eessaar (2014) briefly mentions update anomalies. Thus, the NS theory uses different terminology (like combinatorial effects and composition of concerns) and does not use update anomalies in the database world to explain similar problems in the general domain of systems.

Interestingly, only three papers, two of which discuss requirements engineering and use cases, say something about redundancy in the context of the NS theory. Verelst et al. (2013) comment that use cases that describe the same functionality or terminology violate the separation of concerns principle. In this case there is redundancy and thus also a CE. The redundancy could exist only at the model level but it could also reflect redundancy in the real world. In total, we found four papers that mentioned that duplication (another word that refers to the concept "redundancy") (for instance, code or processes) causes violation of the separation of concerns principle. De Bruyn et al. (2012) analyze different code smells. These smells often indicate duplicate code that is a sign of CEs according to De Bruyn et al. (2012). They give an example of two modules that share the same code to implement a duplicate functionality but have also additional functionality. They comment that in this case there is a violation of the separation of concerns theorem because the modules have two change drivers. It is unclear from the comment as to whether two identical modules that both have one change driver would violate the theorem. In our view, there is still a CE because modification of one copy requires modifications of other copies or conscious decisions not to modify and the more there are copies, the more work one has to do.

4. Exploring a Common Ground of the Normalization Theories

Data version transparency, action version transparency, and separation of states theorems are all about encapsulating action elements, which call each other and could have multiple versions. The theorems require that each action element should continue functioning if new fields are added to its consumed data elements, newer versions of its called action elements are created, or called action elements return unexpected results (including do not return a result). The database normalization theory does not directly

deal with conceptually similar questions but as we later point out, the use of highly normalized tables makes it easier to implement data version transparency. Each subtype inherits all the properties of its supertype. The database normalization theory does not deal with problems that are conceptually similar to the theorems about encapsulation and versioning. Thus, it is incorrect to say that the NS theory is its generalization.

On the other hand, the separation of concerns theorem deals broadly with the same questions as the database normalization theory and therefore one can say that these theories have an overlap. The NS theory describes systems in terms different primitives, including action element and data element. WEB (c) defines concern as "A canonical solution abstraction that is relevant for a given problem." However, the separation of concerns, according to the NS theory, applies specifically to the action elements. "Essentially, this theorem describes the required transition of submodular tasks—as identified by the designer—into actions at the modular level" (Mannaert et al., 2012b). On the other hand, users of the database normalization theory apply it to the data elements. We argue that database normalization is a domain-specific application of separation of concerns theorem to the data elements in the domain of databases.

We wanted to validate our impression that literature *does not explain* the database normalization theory in terms of separation of concerns (as of October 2015). Firstly, we looked all the materials (books and papers) about normalization topic that the current paper mentions and did not find any references to the separation of concerns principle. Secondly, we searched Google Scholar™ with the search phrases "separation of concerns in databases" (one result) and combination of "separation of concerns" and "database normalization" (41 results). We also looked the 357 papers (as of October 2015) that cite the work of Hürsch and Lopes (1995) and searched with phrases "normalization" (two results) and "normal forms" (four results) within this set of papers by using Google Scholar™. Similar search with the phrase "separation of concerns" from the 216 papers citing Fagin (1979) returned four results. We found only one previous source (Adamus, 2005) that comments database normalization in terms of the separation of concerns. He mentions database normalization only once, claiming that it causes tangling, which is not good. He applies the principles of aspect-oriented programming to object-oriented databases and defines aspects so broadly that every software feature that "can be isolated, named, described and documented at a high abstraction level" is an aspect (concern).

Both Adamus (2005) and WEB (c) see concerns as conceptual abstractions that implementation involves the creation of one or more elements in the implementation environment. Adamus (2005) gives an example that an aspect (concern) *Person*, which one could represent as an entity type in a conceptual data model, could be implemented by using multiple tables in a relational or SQL database. Adamus (2005) thinks that because of the creation of multiple tables the concern is scattered to multiple places of the database. He characterizes such concern as *tangling*, meaning that due to the restrictions of an implementation environment (in this case a SQL DBMS) implementers have no other choice than to scatter the concern. Of course, this does not have to be the case if the DBMS properly supports definition of new types and using these as column types as the relational model requires. Each concern is a conceptual abstraction that according to WEB (c) depends on problem at hand. The definitions leave it fuzzy as to what is an appropriate abstraction level to consider something as a concern. Hence, in this case, one could change the level of abstraction. Instead of looking *Person* as a concern that one must implement with the help of tables *Person_detail* and *Person_e_mail*, one could consider these tables as data elements that bijectively map to

and represent more fine-grained concerns. Based on these two tables, one could create a denormalized view (also a data element) that corresponds to the more coarse-grained concern *Person*. Database normalization helps us to achieve more fine-grained tables that correspond to more fine-grained concepts (see the work of De Vos (2014)). Thus, we can use the database normalization to find conceptual structure of the system and define its conceptual (data) model. Usually the order of creation is the opposite. Firstly, modelers create a conceptual data model. Based on that they create the database design models by using model transformation.

Very informally speaking, in case of tables the separation of concerns means that each table must address the main general concern of a database that is to record data corresponding to some entity type or relationship type. In case of tables that are only in first normal form, each table is a structure that contains data about multiple types of entities and relationships. For instance, it makes it more complex to perform data modification operations and enforce certain integrity constraints. The higher is the normalization level of tables, the more fine-grained the concerns will become and thus the separation of concerns gradually increases. Hürsch and Lopes (1995) note that there must be a gluing mechanism that holds the decoupled concerns together. In case of relational or SQL databases and "traditional" projection-based normalization these are candidate keys and foreign keys of tables. One can use the values of the keys to join the decomposed tables back together in the nonloss manner (recouple concerns).

What about constraints to data, which help us to enforce business rules? In our view, one should look these rules as separate concerns as well. Ideally, one could implement each such rule by using one declarative database language statement. However, if due to the restrictions of a DBMS the implementation of a constraint requires the creation of multiple trigger procedures, which have to react to different events and are perhaps attached to different tables, then this is an example of tangled database concern. Declarative statements and triggers are examples of action elements of the NS theory.

Implementation platforms of concerns determine what concerns one can and cannot separate. According to the terminology of Tarr et al. (1999), data and functionality would be different dimensions of concerns along of that to decompose the system. For instance, object-oriented systems couple functionality and data because objects contain both methods, which implement behavior and attributes, which hold data. The same is true in case of user-defined structured types in SQL that couple attributes (data) and methods to access the attributes. These methods also have to enforce whatever constraints there are to the values of the attributes in addition to the type of the attribute. On the other hand, the relational data model takes the approach that it is possible to specify separately operators for performing operations with data, constraints to restrict data, data structures, and data types. This distinction follows the spirit of the separation of concerns principle.

Hürsch and Lopes (1995) write about the benefits of separating concerns and observe that we must separate the concerns at both the conceptual and the implementation level to achieve these. Section 4.1 explains that unfortunately current SQL DBMSs have many shortcomings in this regard.

Hürsch and Lopes (1995) note that separating concerns makes it possible to reason about and understand concerns in isolation. One could say the same about tables that one creates as the result of decomposition during the normalization process. Each table heading represents a generalized claim (external predicate) about some portion of the world. The lower is the normalization level of a table, the more its predicate contains weakly connected sub-predicates as conjuncts. During the normalization process, these

predicates are separated to distinct tables. For instance, a table in first normal form might have the following external predicate that explains the meaning of the table to its users: *Client with identifier CLIENT_NO, first name CFNAME, and last name CLNAME rents property, which has the identifier PROPERTY_NO, address ADDRESS by paying RENT Euros in a month.* If one further normalizes the table, then one of the resulting tables would contain data about clients and has the following external predicate: *Client has identifier CLIENT_NO, first name CFNAME, and last name CLNAME.* All that the current paper writes about separate concerns is true and relevant in case of this table as well.

Hürsch and Lopes (1995) write that separation of concerns makes concerns implementable one by one and substantially reduces the complexity of implementing individual concerns. In case of database normalization, we can implement the resulting tables, which correspond to separate and more cohesive concerns, one by one. If we proceed with the normalization process, then it is gradually easier to implement these tables because due to the decomposition the tables have gradually less and less columns. Moreover, it is easier to update data in these tables due to the reduction of update anomalies. If we want to enforce constraints to data, then as the result of normalization process more and more constraints must refer to multiple tables. It is complicated to enforce these constraints in modern SQL DBMSs because they do not support general declarative constraints (assertions) that can refer to multiple tables and multiple rows and are not directly connected to any table (separation of concerns at the implementation level). However, this is a restriction of implementation environments not a principal flaw of the relational data model and the normalization process.

Dependencies between columns based on that the tables are decomposed are actually a type of constraints that should be known and enforced by the system. These constraints are also separate concerns. Complexity of their implementation depends on the choices during the decomposition process. Sometimes it is possible to decompose a table in multiple different nonloss ways. Date (2003) explains that one could classify the resulting tables (projections) of each decomposition as independent or dependent. If the resulting tables are independent, then it is possible to enforce constraints corresponding to functional dependencies by declaring proper keys to the new tables. For instance, in case of table *Client*, one has to declare {client_no} as the key to enforce the rule that each client must have exactly one first name and exactly one last name. However, if the resulting tables are dependent, then enforcing these constraints requires the creation of database constraints that span multiple tables, making the implementation more complex.

Hürsch and Lopes (1995) comment that separating concerns results in a weak coupling of the concerns, meaning, "changes to one concern have a limited or no effect on other concerns." Date (2003) comments that in case of two tables that are the result of an independent decomposition, one can update data in either of these without updating the other (expect the updates that violate referential constraints between the tables). In case of two tables that are the result of a dependent decomposition, one must monitor data updates of both tables to ensure that they do not violate dependencies that span the tables. Understandably, Date (2003) suggests us to prefer independent decompositions.

Systems usually have a layered architecture that is themselves a manifestation of the separation of concerns principle (Mannaert et al., 2012b). Changes in a concern at one layer must be propagated to the depending concerns at the upper layers as well. At least partially, these changes can be hidden behind interfaces, which elements recouple concerns of the interfaced layer but at the same time allow us to avoid tight coupling

between the layers. Because each layer (an implementation of a coarse-grained concern) has its own responsibilities, avoiding their tight coupling is a direct application of the separation of concerns principle. An example of interface is virtual data layer, which can contain views that join (recouple) data from different tables (Burns, 2011). There could be duplication of tasks at different layers because at different layers they help us to achieve different goals. For instance, data validation in user interface gives quick feedback and reduces network load whereas database constraints among other things express the meaning of data and help the system to optimize operations. However, there is a dependency between the tasks and changes in one must be propagated to another. Preferably, it must happen automatically to make the redundancy controlled.

Hürsch and Lopes (1995) comment that weak coupling of concerns increases their reusability. Burns (2011) notes that reusability of data in case of different applications and business uses is one of the basic principles of data management. If a table contains data about different general business areas (in other words about separate concerns), then it could discourage data reuse. For instance, if a program has to register personal data about clients but the table *Client* contains also information about the contracts with clients and products or services that the clients consume, then it *could* increase temptation to create a separate table just for this program to register some personal details of clients.

Database normalization deals with separating concerns at the conceptual database level according to the ANSI/SPARC layered architecture of DBMSs. The layered architecture is themselves an example of separation of concerns. DBMSs separate concerns like persistence, checking privileges, speeding up performance of operations, failure recovery, and replication in the sense that the DBMS together with its human operators deals with these questions in the background and the users of data *ideally* do not have to refer to these in statements of data manipulation language.

If one wants to achieve the highest separation of concerns in case of designing tables, then one must create tables that are in sixth normal form. If a table has the key (a set of columns) and in addition at most one column, then it is in sixth normal form. We cannot decompose such tables in a nonloss manner to tables that have fewer columns than the original. Date (2006a) calls it the ultimate normal form with respect to normalization. The use of this kind of tables offers advantages like better handling of temporal data, better handling of missing information, and simplification of schema evolution. Regarding the last property, Panchenko (2012) notes that modifiability of a database schema is one of the most important quality criteria of applications that use the database. Conceptually, this argument is also the driving force behind the NS theory, which offers guidance how to create highly evolvable systems, which retain this characteristic over time. The sixth normal form tables have been popularized by the anchor modeling (Rönnbäck et al., 2010), which is a model-driven development approach for creating highly evolvable databases that are well suited for storing temporal data. It results with tables that are mostly in sixth normal form. However, the approach also requires the creation of views and function that present more denormalized view of data. Thus, there are multiple interfaces for working with the data. One provides direct access to tables that correspond to anchors and attributes to make it possible to register new true propositions about the world. Another contains elements that implement less granular concerns (for instance, the latest view in case of anchors that presents the latest values of historized attributes). Rönnbäck et al. (2010) do not write about separation of concerns in the context of their approach.

In relational databases, first normal form requires that each field of a row must contain exactly one value that belongs to the type of the corresponding column. Although this is not a part of the database normalization theory, one must be able to use any type (including types that have complex internal structure; only excluding the type *pointer*) as a column type. Selection of column types determines the granularity of values that database users and DBMS can process (read from and write back to the system) as one value. Thus, one could decide that in case of the entity type *Person* there is a table *Person* with the columns *first_name* and *last_name* with type VARCHAR or perhaps user-defined type *Name_T*. It means that one wants to treat values in these columns as the main units of processing (reading and writing) data of persons. On the other hand, one can decide to create the user-defined type *Person_T* with components of its possible representation *first_name* and *last_name* and to create a column with this type to be able to record values with this type. In this case, one treats values with the type *Person_T* as the main units of processing. The relational normalization theory does not look inside the recorded values of the column types and does not deal with possible data redundancy within these values.

Mananert et al. (2012b) state that the identification of tasks that one should treat as separate concerns and should place to different modules is to some extent arbitrary. In relational databases, the use of tables in sixth normal form together with the possibility to use simple or complex types as column types offers a flexible model in determining the granularity of concerns. One could design tables so that dealing with first names and last names are separate concerns, and the corresponding data is in separate tables. On the other hand, one may decide to consider dealing with data about persons as one concern and register data of persons in a table that has exactly one column, which has the type *Person_T*. This column is also the key column of the table. The latter design is less flexible, just like we expect from less-separated concerns. It is more difficult to implement recording historic attribute values in case of some attributes (but not all attributes) of *Person*. Difficulties in database evolution like starting to register data corresponding to new attributes or making constraints to attributes stronger or weaker depend on the inheritance model of types that the DBMS offers. For instance, a new requirement is to start registering national identification numbers of persons. It could be that the system does not support type inheritance or its inheritance model does not permit definition of subtypes that values the system cannot represent in terms of the possible representation of its supertype. In other words, we cannot add to the possible representations of the subtype a new component *national identifier*. In this case, we have to create a new type without inheritance and have to create a new table that has the column with the new type. Now there are two places (tables) in the database where the names of persons are registered.

Developers could externally couple modules (action elements) by using an integration database (Van der Linden et al., 2013). In this case, they can implement data elements as relational database tables. NSs must exhibit data version transparency, meaning that action elements must function even if there are multiple versions of data elements. Mannaert et al. (2012b) describe anticipated changes in systems in terms of very generic primitive elements. Two of these changes are addition of a data attribute/field and addition of a data element. If we use the anchor database approach, then both these modifications are non-invasive to existing tables, meaning creating new tables, not altering the existing tables. Thus, every old database conceptual schema version is a proper subset of the latest schema. We can hide such changes in the schema behind the interface of views and functions that encapsulate the database. Anchor

database offers an interface that denormalizes data and presents it in terms of different temporal perspectives. Adding tables means that we have to recreate the functions and views but the action elements that use these can continue function normally. Until the creation of new version of the action elements, they just do not use the new data presented through the database interface.

Publications about database normalization theory state explicitly that its objective is to reduce data *redundancy*. The papers about the NS theory (Mannaert et al., 2012b) as well as separation of concerns that is a founding principle of the NS theory (Hürsch and Lopes, 1995), (Tarr et al., 1999), (Akşit et al., 2001) pay less attention to this objective. Still, Mannaert et al. (2012b) offer a proof of the separation of concerns theorem in terms of redundant implementations of a task (let's call it A) in different action elements (see part a1 of Fig. 1), showing how it leads to unbounded amount of coding changes during system evolution. If the task A needs modification, then one has to make changes in multiple modules. The more modules that contain A there are, the more difficult and time consuming the work will become. Thus, there is a CE, which we must eliminate according to the NS theory. In essence, it is an example of update anomaly. By the way, conceptually similar situation could appear within one module as well (see part a2 of Fig. 1).

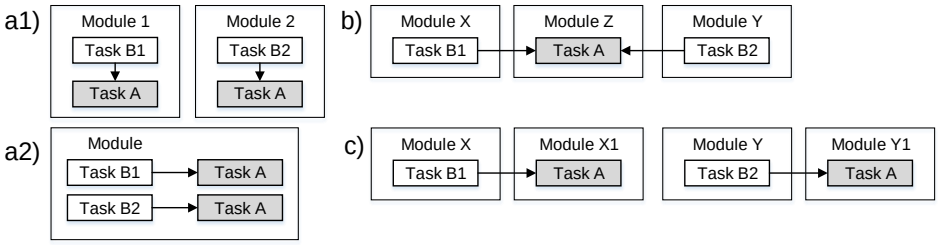

Fig. 1. Different design options to combine and separate concerns

The solution to the problem depicted in the proof of separation of concerns theorem in Mannaert et al. (2012b) is separation of the tasks A to a separate module (see part b of Fig. 1). However, one could do this in a way that actually does not reduce redundancy and hence does not avoid a CE (see part c of Fig. 1). The principle of nonloss decomposition in case of the database normalization theory (Date, 2006a) requires among other things that one needs all the resulting components to restore the original table. This would correspond to the design b on Fig. 1. The canonical form property of concerns also means that one must avoid such redundancy. However, what should happen if there is already a separate module that implements the same version of Task A as in case of Fig.1 parts a1 or a2? Clearly, there is no need to create a new module (Module Z in Fig.1 part b) but instead the redesigned modules should refer to this existing module. Unfortunately, both normalization theories look one element at a time during the decomposition and do not consider other already existing elements in the system, possibly leading to the creation of redundant elements. At least in case of relational or SQL databases, there is the principle of orthogonal design to search and reduce redundancy across tables whereas in case of the NS theory there is no explanation about what to do with CEs that appear because of the duplication of action elements, each of which separately conforms to the separation of concerns theorem. Just like different names of action elements make it more difficult to discover redundancy, different column names make it more difficult across tables.

Merunka et al. (2009) address the same problem in the definition of second and fourth normal form for the object-oriented data model and require elimination of such CEs.

Aspect-oriented software development describes cross-cutting concerns as concerns that are scattered to multiple other concerns leading to duplication or significant dependencies. They could exist because of the restrictions of the implementation environments that do not allow implementation of such concerns in any other way. Adamus (2005) characterizes such concerns as *tangled aspects*. However, implementation in a manner that increases dependencies and duplication could also be a choice of designers. This is the case in case of violations of the orthogonal design principle in databases. It means that two or more tables do not have mutually independent meaning in the sense that the same row can satisfy the predicates of multiple tables and thus appear in multiple tables. The choice to violate the principle is not from absolute technical necessity. The reason could be an expectation of better performance of some read operations.

In case of the NS systems, denormalization would mean knowing violation of one or more design theorems in at least one part of the system to improve the overall satisfaction with the system. The database normalization theory defines intermediate levels of normalization (normal forms) whereas the NS theory only states the end goal that the system must satisfy all the design theorems. It means that if one wants to reverse the normalization process after its completion or perhaps not to complete it in the first place, then the database normalization theory offers possible levels where to stop but the NS theory does not. The proponents of the NS theory promote full normalization at the software level but allow relaxation of the rules at the higher levels (Verelst et al., 2013) without exact guidelines where to stop. Both theories offer normalization as a tool that one should use according to his/her best understanding and the needs of a particular context.

Different levels of database normalization make it possible to do database normalization iteratively in a manner that different iterations take tables to different normal forms. The NS theory also suggest possibility of iterative normalization (Mannaert et al., 2012b) in the same way. Because of the lack of different levels, the authors have to use vague descriptions like “making the elements ever more fine-grained over time.”

One could say that denormalization in terms of separation of concerns appears in layered architectures. For instance, Pizka (2005) notes that normalizing (and thus reducing update/change anomalies) on one level of abstraction does not guarantee that higher levels of abstraction are free from these anomalies. For instance, a page or a form in a user interface is a user connector element in terms of the NS theory. This element may combine different tasks (functionality) and thus recouple concerns of the lower system layers. Moreover, it may present data about different entity types and relationship types that one considers separate concerns at the database level and thus recouple these concerns. Another example is that in the database one can implement the virtual data layer (Burns, 2011), which consists of functions and procedures that are both action elements as well as views that are data elements. Again, these elements could recouple concerns of the lower system layers. Yet another example are macros in many applications that recouple lower-level actions and considerably improve the usability of the system. Thus, we see that denormalization at the higher system layers in terms of lower layers is even desirable to ensure usability of the system. In this context, we cannot speak about total separation of concerns as required by the NS theory but only

about a goal to increase the separation within and between layers but not necessarily in the interfaces of these layers.

"Concern" concept is very flexible. Thus, an interpretation of the previous section is that in case of different layers the completely separated (atomic) concerns have different granularity. It is the general property of the layered architecture that elements at the higher layers "abstract away" and hide elements at the lower layers. Thus, they recouple concerns that are separate at the lower layers. For instance, if we create tables in a SQL database, then it abstracts away from the users of the tables things like internal data storage, indexing to improve query performance, algorithms for checking integrity constraints, and logging data modifications to be able to roll them back. At the internal database level (layer), one would consider these as separate concerns but at the conceptual level of databases one recouples these concerns and works with a higher-level concept that is a *table*.

Moreover, denormalization at one layer may lead to denormalization at the upper layers. Raymond and Tompa (1992) write that data redundancy leads to redundant processing in applications. Because such redundancy violates the canonical form property of concerns it means that CEs in data lead to the CEs in applications. One can mitigate the effect with the help of views, which can give impression of data redundancy to readers while reducing redundant processing because there is no need to update data in multiple places. Tarr et al. (1999) explain the concept of one single, dominant dimension of separating concerns at a time in typical software environments. Data models (like the relational data model) that support creating views and thus implementing virtual data layer break the "tyranny of the dominant decomposition" *within* the data dimension of concerns.

Akşit et al. (2001) describe six "c" properties that each concern should have. The database tables (as implementations of concerns) have all the six properties. Tables correspond to *solution domain concerns* that describe parts of systems that one creates to solve problems of clients. Nonloss decomposition of tables ensures that the resulting tables have *canonical form* property, meaning that in the result of decomposition there are no redundant tables, which one does not need to restore the original table. Tables are *composable* by using join operator. Moreover, the closure property of relational algebra ensures that one could further compose the composed tables by again using join operation because output from one such operation could be an input to another relational algebra operation. Tables are *computable*, meaning that they are first class abstractions in the implementation language (for instance, SQL) and thus one could create them in the implementation environment (DBMS). Tables have *closure* property, meaning that both separated and composed tables have all the same general properties (no duplicate rows, each column has a type and unique name within the table, etc.). Concerns must also have *certifiability* property, meaning that it must be possible to evaluate and control their quality. There are certainly methods for evaluating and improving table design (one of them is the database normalization theory) but it would be a topic of another paper.

Hürsch and Lopes (1995) note that redundant system elements help us to achieve fault-tolerance in computing. The database normalization theory does not deal with the data redundancy caused by the need to protect data assets by making distinct copies of them (by using replication or by making backups). Similarly, we want to protect source code or documents by making copies of them, thus increasing redundancy and CEs. This is outside the scope of the NS theory as well. In both cases, we need appropriate tools and processes for version control. Please note that this example also shows that there are types of redundancies that are outside the scope of the theories.

To summarize, the look to the separation of concerns in the NS theory is too narrow. We propose to treat database data structures (for instance, tables) as data elements of the NS theory and argue that normalization process separates concerns in case of these elements. A table as a data element might couple multiple more fine-grained concerns that we separate as the result of the process. One can recouple these concerns with the help of views that are also data elements and are a part of database interface. Constraints to data are also concerns. Declarative statements for enforcing constraints and database triggers are action elements of the NS theory. If, despite technical restrictions of a DBMS, one manages to enforce a constraint by using a declarative statement, then the concern maps to exactly one action element. One may need multiple triggers to enforce a constraint and it is an example of tangled concern. If a constraint is a logical conjunction of more fine-grained constraints, then it couples more fine-grained concerns. Designers have to decide, based on the context, what is the best level of separation of these database-related concerns. The NS theory dictates that the concerns must be as separated as possible.

Following of the separation of concerns principle clearly offers advantages. If one chooses to follow it only in case of functionality of the system, then one can compare the resulting system with a factory that has machinery (functionality) that has to be easy to maintain and extend and needs a lot of inexpensive raw material (data) to produce products that quality is not so important. Perhaps the only requirement is that the factory must fulfill orders as quickly as possible. If the system follows separation of concerns principle in case of functionality *and* data, then, in addition to machinery, the factory and its owners also value more the quality of material that it consumes and products that it produces.

4.1. Some Violations of Separation of Concerns in the SQL Database World

1970-ties saw separation of database management functionality from applications to separate database management systems (Van der Aalst, 1996). This is a good example of the application of the separation of concerns principles.

It is unfortunate that in the domain of databases, researchers and developers often overlook the separation of concerns principle and do not describe problems that its violations cause in terms of the principle. For instance, Badia and Lemire (2011) only recently raised a question as to whether it is time that database design science should start to look relations as purely conceptual entities that in other words means completely separating concerns between the conceptual and internal levels of databases. However, this is something that already Codd's 12 Rules (Voorhis, 2015) desired in terms of physical data independence. Hürsch and Lopes (1995) also note that the benefits of separating concerns (like higher level of abstraction, better understanding, weak coupling, and also more creative freedom because of less dependencies) appear if it is applied at both conceptual and implementation level.

Here, we present a non-exhaustive list of such problems in SQL DBMSs. Many of these support the observation of Hürsch and Lopes (1995) that separation of concerns is often practiced at the conceptual level, but not at the implementation level. Unfortunately, instead of demanding to fix the problems, there are calls to scrap the relational model and start to use technologies that have even more such problems. For instance, the need to enforce integrity constraints and access rights to data elements are cross-cutting concerns. Database languages should support separation of these concerns by providing dedicated sub-languages that one can use to implement these concerns in

one place (in a DBMS). Coupling the concerns with application code or procedural database interface leads to scattering of the functionality across the code base and significant dependencies. For instance, there could be a requirement to restrict access of certain users to data about certain entities or their specific attributes or relationships. If the applications have a task to save data with security labels and a task to ask data by explicitly referring to the labels, then now this mechanism is coupled with the database application code and is scattered to multiple places in one or more applications. The same thing happens if one uses a DBMS that does not require explicit definition of database schema at the database level. In this case, the developers of database applications have to define the schema implicitly within algorithms (procedures) of applications across the code base of applications.

Does it mean that application developers themselves do not believe the principle? We do not think that it is the case. Instead, a problem seems to be that application developers have not thought about these problems in terms of the separation of concerns principle.

We think that understanding the root causes of the problems and similarities with the problems of their own domain (for instance, application development) could ideally lead to the increased understanding of the logical difference between a model and its implementations. In the longer run, it could ideally lead to technology improvement because customers start to demand it from vendors. In case of solving the separation of concerns problems, SQL DBMSs and their offered model of data management could be a good example and success story of applying the separation of concerns principle.

Next problems occur because the DBMSs do not provide sufficient separation of different database levels (layers, tiers) that the ANSI/SPARC DBMS model (Date, 2003) defines and thus these have multiple change drivers.

- One can denormalize tables, which are the elements at the conceptual level of a database, to improve the performance of certain queries. With the changes at the conceptual level, one hopes to influence the organization of data at the internal level of the database. One could also denormalize to offer data to applications in an aggregate form so that it is easier for them to read the data, thus fulfilling a task of the external level. The reason could be that the DBMS does not support building virtual data layer (Burns, 2011) or because it does not provide suitable operators (or means to create these) to aggregate data on the fly. Moreover, one could denormalize to make possible or simplify enforcement of some constraints in a database because the DBMS does not support assertions or subqueries in table constraints. Thus, depending on circumstances, the conceptual database level couples the following concerns: reflecting the concepts, relationships, and rules of the domain in the scope of the functional requirements; read operations of applications; data integrity; performance; data storage.
- Horizontal partitioning of tables at the conceptual database level to improve performance of queries makes the conceptual database level to couple the following concerns: reflecting the concepts, relationships, and rules of the domain in the scope of the functional requirements; performance; data storage. Moreover, if a statement of data manipulation language refers directly to partitions at the conceptual or internal level, then the statement couples the following concerns: functional requirements; performance; data storage.
- One can use views to implement the external level of databases. If the possibility of updating data in the database through a view or performance of queries based on a view depends on how one writes its subquery, as it

unfortunately does in modern SQL DBMSs, then the external level couples the following concerns: fulfilling functional requirements; the use of language constructs; performance.

Next problems occur because the SQL Data Manipulation Language is not sufficiently declarative. Therefore, one has to take into account low-level implementation details of the DBMS while writing the statements and the statements have multiple change drivers.

- In case of declarative database language statements, the system has to produce the execution plan that specifies the best low-level algorithm for achieving the desired results of the statement in case of current data. If the proposed plan and thus, also, the efficiency of executing the statement depends on how one writes the statement, then the system couples the following concerns: functional requirements; the use of language constructs; performance.
- If a SQL dialect provides an option to use hints in the SQL Data Manipulation Language statements that guide the selection of its execution plan, then the statements that use the hints couple the concerns: functional requirements; performance. As a result, the code to improve performance is scattered across triggers, database routines, and applications, making its maintenance difficult and causing potential performance problems if the hints do not take into account the current amount and distribution of data in the database.
- If a DBMS implements the Multiversion Concurrency Control method and provides snapshot isolation instead of serializable snapshot isolation, then the database users have to lock explicitly data elements to avoid read phenomena of concurrent transactions and the resulting data inconsistencies. The locking policy depends on the internal implementation of DBMS. Thus, the system couples the following concerns: functional requirements; concurrency control; data integrity.

Next problems occur because DBMSs provide inadequate support for creating declarative constraints that leads to the scattering of constraint checking to multiple parts of the system. The elements that implement the constraints have multiple change drivers. In our view, functional requirements and data integrity are separate concerns because data integrity comes from the business rules that are determined by the domain, are universal for applications, and thus do not depend on a particular application that uses data.

- The creation of trigger procedures instead of a declarative integrity constraint may mean scattering integrity enforcement code to multiple parts of a database because the triggers have to react to all the possible events that could lead to the violation of the constraint. The same trigger procedure could couple enforcement of multiple integrity constraints in addition to other tasks. The triggers couple the following concerns: functional requirements; data integrity; concurrency control.
- In SQL user-defined structured types, one cannot enforce declarative constraints to the values of attributes, except the type of attribute. Thus, one has to implement these constraints in the methods that modify the values of the attributes. The same method could couple enforcement of multiple constraints as well as business logic for retrieval and manipulation of data. The methods couple the following concerns: functional requirements; data integrity. If there are multiple methods that change the value of the same attribute, then these duplicate constraint-checking code.

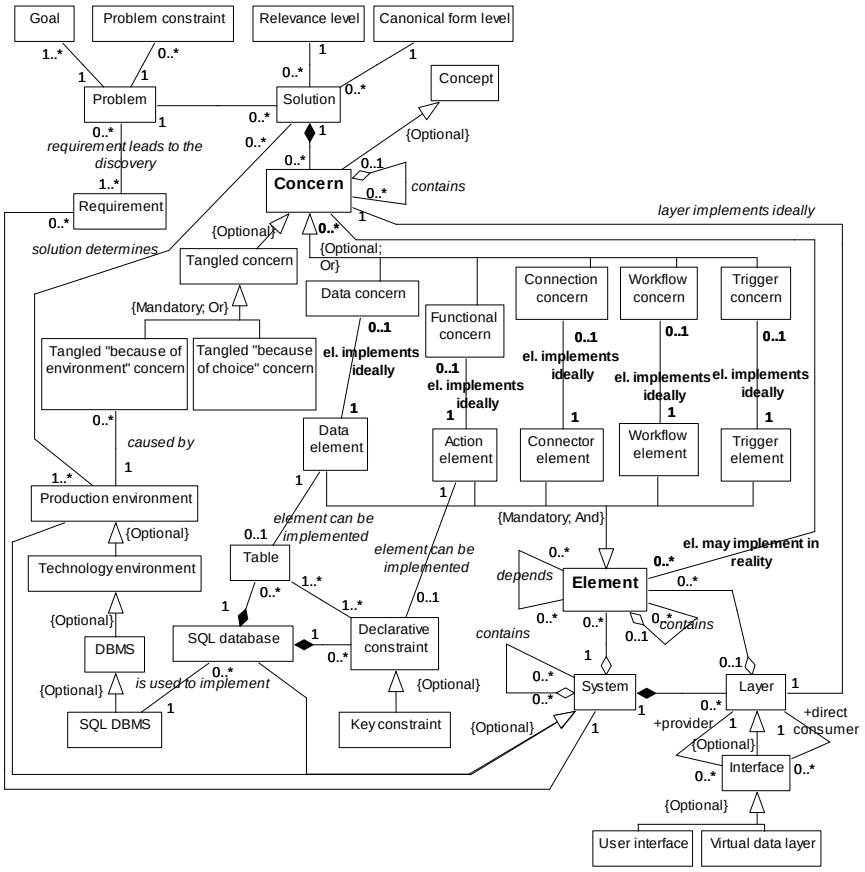

Fig. 2. A conceptual model of a holistic view to the separation of concerns principle

- The enforcement of constraints in procedures or functions (routines) that are a part of virtual data layer (Burns, 2011) may be inevitable due to the technical restrictions of a DBMS. It makes the routines to couple following concerns: functional requirements; data integrity; concurrency control. The same routine could couple enforcement of multiple integrity constraints as well as business logic for retrieval and manipulation of data. If there are multiple routines that modify data in the same table, then these duplicate constraint-checking code.
- Explicitly created unique index (an internal level element) to enforce uniqueness (a constraint at the conceptual level) couples the following concerns: data integrity; data storage; performance. Unfortunately, SQL does not provide means to *declare* uniqueness to a subset of table rows but some DBMSs permit the creation of partial unique index to a subset of table rows.

Next problems do not belong to the previous categories.

- In SQL, one can create user-defined structured types that group attributes (structure) and methods (behavior) to access and modify their values. These types couple concerns of structure and behavior whereas The Third Manifesto

(Date and Darwen, 2006) offer a model that separates types (with possible complex internal structure), operators that one can use to perform operations with the values of these types, and integrity constraints.

- In some SQL dialects, if one creates a table (an element at the conceptual database level), then one has various options how to guide the internal storage of the table data by using the same statement. Thus, the statement couples concerns of representing data at the conceptual and internal level.
- If a SQL dialect permits inline functions in data manipulation statements to improve performance, then the statements couple declarative and procedural processing as well as performance concern. The statements duplicate functions.
- SQL terminology speaks about tables. The Third Manifesto (Date and Darwen, 2006) points that table as a variable and table as a possible value of this variable are two distinct concepts. If one uses one concept instead of two distinct concepts, then it is a violation of the separation of concerns principle at the semantic level. Another example is the use of the word "database" to refer to databases as well as DBMSs. One can notice similar problem in case of the NS theory where it is unclear when and where data element means value or variable.
- The SQL data model violates orthogonality principle in language design (Eessaar, 2006). A complex web of dependencies between model elements means that a localized change in a database design can lead to other cascading design changes. This is a violation of the separation of concerns at the language level.
- If the specification of the underlying data model of a database language couples the description of its concrete syntax like in case of SQL, then one firstly has to separate the description of the data model (Eessaar, 2006) to be able to reason about, analyze, and compare the data model with other data models.

4.2. Towards a Holistic View of Separation of Concerns

Based on the previous discussion, we propose a conceptual model of a holistic view of separation of concerns principle where action elements are not the only focus (see Fig. 2). It takes into account the ideas of Mannaert et al. (2012b), Adamus (2005), WEB(c) as well as our understanding of the principle. The NS theory describes different types of elements, and all these could correspond to different types of concerns.

5. Conclusions and Future Work

Data and functionality are two fundamental aspects of systems. Unfortunately, there is a mental gap between these aspects. Therefore, nowadays many look the corresponding research and development fields as quite distinct with different terminology, tools, problems, processes, and best practices. We think that it should not be the case and that the fields have many similar problems and solutions. One of these is the principle of separation of concerns that WEB (c) calls "ubiquitous software engineering principle." In reality, researchers and developers rarely discuss it in the context of data. Although the ideas about concerns in databases and applying the principles of database normalization to software engineering are not new, there is very little literature about this.

Recently the theory of normalized systems (NS) has started to gain attention. By using four design theorems, it declares the conditions that systems must fulfill in order to be free of combinatorial effects and thus be highly evolvable. The theory is general and should be applicable to all kinds of systems. Database normalization is older and more mature theory. In this paper, we wanted to gain understanding what the relationship between the theories is. We conducted a literature review of 40 papers about the NS theory and found that the papers had little to say about databases and almost nothing to say about the database normalization. We found a comment that database normalization resolves many combinatorial effects in databases. Papers of the NS theory treat databases as external services used by applications and concentrate attention to the application design.

The lack of references between the theories is not surprising because the NS theory defines "separation of concerns", which is one of its central pillars, in terms of the action elements but not data elements. We analyzed the theories and concluded that the database normalization theory actually helps us to achieve separation of concerns in case of data elements, meaning that data about different entity types and relationships and in extreme cases data corresponding to different attributes of entity types is in different data structures (for instance, tables). On the other hand, the NS theory is not a generalization of the database normalization theory, because the database normalization theory does not deal with the questions of encapsulation and versioning.

We observed that methods of coupling that allow us to achieve higher separation of concerns in action elements could reduce separation of concerns in case of data. If we treat data and functionality as equal partners, then we have to find a balance. The advantages of database normalization are conceptually very similar to the advantages that the following of the separation of concerns principle helps us to achieve in case of action elements (for instance, modules or use cases). The respective processes are also quite similar. Both can be done iteratively, look system one element at a time, increase the number of elements in the system, make elements more cohesive and understandable, make it easier to evolve the resulting system, and are not silver bullets in terms of removing redundancy. A difference is that the database normalization offers different levels of normalization (normal forms) whereas the NS theory only declares the end goal that all the concerns must be separated. However, literature does not explain database normalization theory in terms of separation of concerns. We pointed to the problems of lack of separation of concerns in SQL databases and SQL DBMSs. We proposed a conceptual model of a holistic view to the separation of concerns principle that considers both action and data elements as well as other elements proposed by the NS theory. For instance, in the education process it is important to facilitate understanding that redundancy and coupling weakly related elements is a fundamentally similar challenge to both data and functionality. Thus, if one teaches database normalization to people with business analysis, system analysis, or application development background, or, for instance, application design to database developers, then one can and should point to the conceptually similar problems and solutions in different domains.

To conclude, the paper is a step towards understanding that problems and solutions of data and functionality management are not so different after all. One should not declare that database normalization is old fashioned, nowadays almost unimportant, and a product of a legacy technology but instead learn and apply its lessons. We agree with Raymond and Tompa (1992), who write that "the update implications of systems are important, and that they can be profitably studied in a formal setting taken from dependency theory."

Future work could include the use of the database normalization theory for the normalization of action elements (for instance, use cases). To do this, one firstly has to find the most suitable way to represent the action elements in terms of data elements (tables). To allow bigger separation of concerns in case of both action and data elements there is a need to improve technology. Otherwise, there is a continuous need to find a proper balance between normalization of action and data elements. It would be interesting to analyze modern NoSQL systems as well as systems that implement the relational model according to the principles of The Third Manifesto to find how they support achieving NS, including separation of concerns. It would also be interesting to analyze as to whether the use of tables in sixth normal form together with the type inheritance model proposed by Date and Darwen (2006) would help us to implement NS and if not, then what changes in the model are needed. Perhaps the most important is to create a teaching program that emphasizes fundamental similarities as well as differences in the application and database design.

References

Aalst, W.M.P. van der (1996). Three good reasons for using a Petri-net-based workflow management system, In: Wakayama, T., Kannapan, S., Chan Meng Khoong, Navathe, S., Yates, J. (Eds.), *Proceedings of IPIC*, International Working Conference on Information and Process Integration in Enterprises (14-15 Nov. 1996, Cambridge, Massachusetts, USA), 179–201.

Adamus, R. (2005). *Programming in aspect-oriented databases*. PhD thesis, Institute of Computer Science, Polish Academy of Science, Warsaw, Poland.

Akşit, M., Tekinerdogan, B., Bergmans, L. (2001). The six concerns for separation of concerns, In: *Proceedings of ECOOP*, Workshop on Advanced Separation of Concerns (18-22 June 2001, Budapest, Hungary).

Badia, A., Lemire, D. (2011). A call to arms: revisiting database design. *ACM SIGMOD Rec.* 40, 61–69.

Burns, L. (2011). *Building the Agile Database. How to Build a Successful Application Using Agile Without Sacrificing Data Management.* Technics Publications, LLC, New Jersey.

Carver, A., Halpin, T. (2008). Atomicity and normalization, In: Halpin, T., Proper, E., Krogstie, J., Hunt, E., Coletta, R. (Eds.), *Proceedings of EMMSAD*, International Workshop on Exploring Modeling Methods for Systems Analysis and Design (16-17 June, Montpellier, France), CEUR-WS.org, 40–54.

Codd, E.F. (1970). A relational model of large shared data banks. *Comm. ACM* 13, 377–387.

Cooper. A, Reimann, R., Cronin, D., Noessel, C., Csizmadi, J., LeMoine, D. (2014). *About Face. The Essentials of Interaction Design.* 4th ed. Wiley, Indianapolis, Indiana.

Date, C.J. (2003). *An Introduction to Database Systems,* 8th ed., Pearson, Addison Wesley.

Date, C.J. (2006a). *The Relational Database Dictionary. A comprehensive glossary of relational terms and concepts, with illustrative examples.* O'Reilly.

Date, C.J. (2006b). *Date on Database. Writings 2000-2006.* Apress.

Date, C.J. (2007). *Logic and Databases. The Roots of Relational Theory*. Trafford Publishing.

Date, C.J., Darwen, H. (2006). *Databases, Types and the Relational Model,* 3rd ed., Addison Wesley.

De Bruyn, P., Dierckx, G., Mannaert, H. (2012). Aligning the normalized systems theorems with existing heuristic software engineering knowledge, In: Mannaert, H., Lavazza, L., Oberhauser, R., Troubitsyna, E., Gebhart, M., Takaki, O. (Eds.), *Proceedings of ICSEA*, International Conference on Software Engineering Advances (18-23 Nov. 2012, Lisbon, Portugal), 84–89.

De Bruyn, P., Mannaert, H., Verelst. J. (2014). Towards organizational modules and patterns based on normalized systems theory, In: Jäntti, M., Weckman, G. (Eds.), *Proceedings of ICONS*, International Conference on Systems (23-27 Feb. 2014, Nice, France), 106–115.

De Vos, M. (2014). The evolutionary origins of syntax: optimization of the mental lexicon yields syntax for free. *Lingua* 150, 25–44.

Eessaar, E. (2006). *Relational and object-relational database management systems as platforms for managing software engineering artifacts*. PhD thesis, Department of Informatics, Tallinn University of Technology, Tallinn, Estonia.

Eessaar, E. (2014). On applying normalized systems theory to the business architectures of information systems. *Baltic J. Modern Computing* 2, 132–149.

Fagin, R. (1979). Normal forms and relational database operators, In: Bernstein, P.A. (Ed.), *Proceedings of ACM SIGMOD ICMD*, International Conference on Management of Data (30 May - 1 June 1979, Boston, Massachusetts), ACM, New York, NY, USA, 153–160.

Fotache, M. (2006). *Why normalization failed to become the ultimate guide for database designers?*, available at http://papers.ssrn.com/

Fowler, M. (a). *IntegrationDatabase*, available at http://martinfowler.com/bliki/IntegrationDatabase.html

Fowler, M. (b). *ApplicationDatabase*, available at http://martinfowler.com/bliki/ApplicationDatabase.html

Godfrey, M.W., German, D.M. (2014). On the evolution of Lehman's Laws. *J. Softw. Ev. Process.* 26, 613–619.

Grillenberger, A., Romeike, R. (2014). Big data–challenges for computer science education, In: Gülbahar, Y., Karataş, E. (Eds.), *Proceedings of ISSEP*, Conference on Informatics in Schools: Situation, Evolution, and Perspectives (22-25 Sept. 2014, Istanbul, Turkey), Springer International Publishing, 29–40.

Halle, B. von, Goldberg, L. (2010). *The Decision Model: A Business Logic Framework Linking Business and Technology*. CRC Press.

Helland, P. (2009). *Normalization is for sissies.* Conference on Innovative Data Systems Research, available at http://www-db.cs.wisc.edu/cidr/cidr2009/gong /20Helland.ppt

Helland, P. (2011). If you have too much data, then "good enough" is good enough. *C. ACM* 54, 40–47.

Hürsch, W.L., Lopes, C.V. (1995). Separation of concerns. Technical report by the College of Computer Science, Northeastern University.

Kanade, A., Gopal, A., Kanade, S. (2014). A study of normalization and embedding in MongoDB, In: Batra, U., Sujata, Arpita (Eds.), *Proceedings of IACC*, IEEE International Advance Computing Conference (21-22 Feb. 2014, Gurgaon, India), IEEE, 416–421.

Kolahi, S., Libkin, L. (2010). An information-theoretic analysis of worst-case redundancy in database design. *ACM T. Database Syst.* 35, 5.

Komlodi, J.T. (2000). Technical and market viability of object database technology, In: Gibson, R.G. (Ed.), *Object Oriented Technologies: Opportunities and Challenges*, Idea Group Publishing, 58–76.

Linden, D. van der, De Bruyn, P., Mannaert, H., Verelst, J. (2013). An explorative study of module coupling and hidden dependencies based on the normalized systems framework. *Intern. J. on Adv. Syst. and Meas.* 6, 40–56.

Lodhi, F., Mehdi, H. (2003). Normalization of object-oriented design, In: *Proceedings of INMIC*, International Multi Topic Conference (8-9 Dec. 2003, Islamabad, Pakistan), IEEE, 446–450.

Lv, T., Gua, N., Yanb, P. (2004). Normal forms for XML documents. *Inform. Software Tech.* 46, 839–846.

Maes, K., Bruyn, P.D., Oorts, G., Huysmans, P. (2014). On the need for evolvability assessment in value management, In: Sprague, R.H. Jr. (Ed.), *Proceedings of HICSS*, Hawaii International Conference on System Sciences (6-9 Jan. 2014, Hawaii, USA), IEEE, 4406–4415.

Mannaert, H., De Bruyn, P., Verelst. J. (2012a). Exploring entropy in software systems: Towards a precise definition and design rules, In: Kaindl, H., Koszalka, L., Mannaert, H., Jäntti, M.

(Eds.) *Proceedings of ICONS*, International Conference on Systems (29 Feb. - 5. Mar. 2012, Saint Gilles, Reunion), 93–99.

Mannaert, H., Verelst, J., Ven, K. (2012b). Towards evolvable software architectures based on systems theoretic stability. *Software Pract. Exper.* 42, 89–116.

Merunka, V., Brožek, J., Šebek, M., Molhanec, M. (2009). Normalization rules of the object-oriented data model, In: Barjis, J., Kinghorn, J., Ramaswamy, S. (Eds.), *Proceedings of EOMAS*, International Workshop on Enterprises & Organizational Modeling and Simulation (8-9 June 2009, Amsterdam, The Netherlands), ACM New York, NY, USA.

Panchenko, B.E. (2012). Framework design of a domain-key schema of a relational database. *Cybern. Syst. Anal.* 48, 469–478.

Pizka, M. (2005). Code normal forms, In: *Proceedings of SEW*, Annual IEEE/NASA Software Engineering Workshop (6-7 Apr. 2005, Greenbelt, Maryland), IEEE, 97–108.

Pizka, M., Deissenböck, F. (2007). How to effectively define and measure maintainability, In: *Proceedings of SMEF*, Software Measurement European Forum (9-11 May 2007, Rome, Italy).

Raymond, D.R., Tompa, F.W. (1992). *Applying database dependency theory to software engineering*. University of Waterloo, Dept. of Computer Science & Faculty of Mathematics.

Rönnbäck, L., Regardt, O., Bergholtz, M., Johannesson, P., Wohed, P. (2010). Anchor modeling —agile information modeling in evolving data environments. *Data & Knowl. Eng.* 69, 1229–1253.

Tarr, P., Ossher, H., Harrison, W., Sutton Jr, S.M. (1999). N degrees of separation: multi-dimensional separation of concerns, In: Boehm, B.W, Garlan, D., Kramer, J. (Eds.), *Proceedings of ICSE*, International Conference on Software Engineering (16-22 May 1999, Los Angeles, CA, USA), ACM New York, NY, USA, 107–119.

Vanthienen, J., Snoeck, M. (1993). Knowledge factoring using normalization theory, In: *Proceedings of ISMICK*, International Symposium on the Management of Industrial and Corporate Knowledge (27-28 Oct. 1993, Compiègne, France), EC2, Paris.

Verelst, J., Silva, A.R., Mannaert, H., Ferreira, D.A., Huysmans, P. (2013). Identifying combinatorial effects in requirements engineering, In: Proper, H., Aveiro, D., Gaaloul, K. (Eds.), *Proceedings of EEWC*, Enterprise Engineering Working Conference (13-14 May 2013, Luxembourg), Springer Berlin Heidelberg, 88–102.

Vincent, M.W. (1998). Redundancy elimination and a new normal form for relational database design. In: Thalheim, B., Libkin, L. (Eds.), *Semantics in Databases*. Springer Berlin Heidelberg, 247–264.

Voorhis, D. (2015). *Codd's twelve rules*, available at http://computing.derby.ac.uk/wordpress/codds-twelve-rules/

Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P., Davis, M., et al. (2014). Best practices for scientific computing. *PLoS Biol* 12.

WEB (a). *Once and only once*. http://c2.com/cgi/wiki?OnceAndOnlyOnce

WEB (b). *Dont repeat yourself*. http://c2.com/cgi/wiki?DontRepeatYourself

WEB (c). *Separation of concerns*. http://trese.cs.utwente.nl/taosad/separation_of_concerns.htm

Authors' Information

Erki Eessaar, dr., is a full-time Associate Professor at the Department of Informatics in Tallinn University of Technology. He teaches courses about database design and database development. He is the author or a co-author of about 40 research papers and the author of one book in the field of databases and information systems development. Research interests: data models, automation of the evaluation of database design, model- and pattern-driven development and evolution of information systems (including databases) with the help of domain-specific languages, metamodeling, metadata, and software measures.

Received November 18, 2015, revised February 7, 2016, accepted February 8, 2016