ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 6 | Issue: 8 | August 2020 || Journal DOI: 10.36713/epra2013 || SJIF Impact Factor: 7.032 ||ISI Value: 1.188

FRAGMENTATION OF RELATIONAL SCHEMA BASED ON CLASSIFYING TUPLES OF RELATION AND CONTROLLING OVER REDUNDANCY OF DATA IN A RELATIONAL DATABASE LEADS TO VEXING IN DATABASE INTERFACES

Praveen Kumar Murigeppa Jigajinni
Sainik School Amaravathinagar
Post: Amaravathinagar,
Udumalpet Taluka,
Tiruppur Dt,
Tamilnadu State

ABSTRACT

Database system architecture is evolving for the past five to six decades; this gave magnanimous dimension to the field of Database system, not only in terms of architecture, but also in terms of usage of the system across the world, further this has led to revenue growth.

The research on controlling data redundancy in a relation introduced normalization process. There are 6 normal forms which enhance representation of the data, The Normal Forms such as 1NF, 2 NF, 3NF, BCNF, 4NF, 5 NF and 6NF are domain specific meaning the decomposition of relation focuses on the attribute of relations.

This manuscript is specifically examines the vexing in database programming when database is normalized. The decomposition of tables causes an effect on code and increases the complexity in the programming.

KEYWORDS: ***Database Management Systems (DBMS), Structured Query Language (SQL), Relational Database Management system (RDBMS). Normal Forms (NF)***

INTRODUCTION

The concept of a Relational Database Management system (RDBMS) came to the fore in the 1970s. This concept was first advanced by Edgar F. Codd in his paper on database construction theory, "A Relational Model of Data for Large Shared Data Banks". The concept of a database table was formed where records of a fixed length would be stored and relations between the tables maintained [2].The mathematics at the heart of the concept is now known as tuple calculus [3]. The variation on relational algebra served as the basis for a declarative database query language, which in turn formed the basis for the Structured Query Language (SQL). SQL remains as the standard database query language some 30 years later.

Database technology, specifically Relational Database Technology (RDBMS), has seen incremental advancements over recent decades but the competition has narrowed to a few remaining larger entities. The pursuit for improvement has largely left technology practitioners, especially the database administrators, focused on the benefits of performance tuning of the database technology. The infrastructure teams have relied on benefits of the potential compression factors from various RDBMS offerings to help quell the ever expanding footprint of structured and unstructured data that fills the capacity of the typical data center. As a result, infrastructure teams continually seek hardware refreshes with promises of faster disk performance and improved memory caching to gain new database performance tools. Ultimately, a database administrator is left with only a few tools to improve database performance such as adding and tuning database indexes, which only add to the amount of space required for the database. In the end, the data of concern becomes secondary too and can even become smaller than the indexes themselves, leaving the technology practitioners faced with a diminishing rate of return from their efforts. Technology can only go so far and the physics of spinning disks is reached eventually with the associated costs of competing methods to store, retrieve and query data Today, IT professionals are challenged with the task of on going improvements to achieve goals of businesses. Unfortunately, IT budgets do not dynamically grow as fast as business needs. That sequence of events creates majors obstacles for DB infrastructure, deployment,[1]

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 6 | Issue: 8 | August 2020 || Journal DOI: 10.36713/epra2013 || SJIF Impact Factor: 7.032 ||ISI Value: 1.188

HISTORICAL VIEW

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

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 6 | Issue: 8 | August 2020 || Journal DOI: 10.36713/epra2013 || SJIF Impact Factor: 7.032 ||ISI Value: 1.188

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.[4]

THE PROCESS OF NORMALIZATION

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.[4]

NORMAL FORMS

Database Normalization is a method of organizing the data in the database. Normalization is a systematic approach of fragmenting tables to eliminate data redundancy (repetition) It is a multi-step process that creates data into tabular form, removing duplicated data from the relation tables.

Normalization process is divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form
7. Sixth Normal Form

DATABASE PROGRAMMING TECHNIQUES AND ISSUES

Programming is the process of designing and developing a executable code to meet computing results.

When software development is under process the coder has to have clear cut idea of database design process. We now turn our attention to the techniques that have been developed for accessing databases from programs and, in particular, to the issue of how to access SQL data-bases

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 6 | Issue: 8 | August 2020 || Journal DOI: 10.36713/epra2013 || SJIF Impact Factor: 7.032 ||ISI Value: 1.188

from application programs. Our presentation of SQL in Chapters 4 and 5 focused on the language constructs for various database operations—from schema definition and constraint specification to querying, updating, and specifying views. Most database systems have an interactive interface where these SQL commands can be typed directly into a monitor for execution by the database system. For example, in a computer system where the Oracle RDBMS is installed, the command SQLPLUS starts the interactive interface. The user can type SQL commands or queries directly over several lines, ended by a semicolon and the Enter key (that is, "; <cr>"). Alternatively, a file of commands can be created and executed through the interactive interface by typing @<filename>. The system will execute the commands written in the file and display the results, if any.

The interactive interface is quite convenient for schema and constraint creation or for occasional ad hoc queries. However, in practice, the majority of database inter-actions are executed through programs that have been carefully designed and tested. These programs are generally known as application programs or database applications, and are used as canned transactions by the end users, as discussed in Section 1.4.3. Another common use of database programming is to access a database through an application program that implements a Web interface, for example, when making airline reservations or online purchases. In fact, the vast majority of Web electronic commerce applications include some database access commands. Chapter 14 gives an overview of Web database programming using PHP, a scripting language that has recently become widely used.

In this section, first we give an overview of the main approaches to database programming. Then we discuss some of the problems that occur when trying to access a database from a general-purpose programming language, and the typical sequence of commands for interacting with a database from a software program.

1. Approaches to Database Programming

Several techniques exist for including database interactions in application pro-grams. The main approaches for database programming are the following:

Embedding database commands in a general-purpose programming language. In this approach, database statements are embedded into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string EXEC SQL, which precedes all SQL commands in a host language program. A precompiler or preproccessor scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code. This technique is generally referred to as embedded SQL.

Using a library of database functions. A library of functions is made avail-able to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on. The actual database query and update commands and any other necessary information are included as parameters in the function calls. This approach provides what is known as an application programming interface (API) for accessing a database from application programs.

Designing a brand-new language. A database programming language is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full-fledged programming language. An example of this approach is Oracle's PL/SQL.

In practice, the first two approaches are more common, since many applications are already written in general-purpose programming languages but require some data-base access. The third approach is more appropriate for applications that have intensive database interaction. One of the main problems with the first two approaches is impedance mismatch, which does not occur in the third approach.

2. Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model. For example, the practical relational model has three main constructs: columns (attributes) and their data types, rows (also referred to as tuples or records), and tables (sets or multisets of records). The first problem that may occur is that the data types of the programming language differ from the attribute data types that are available in the data model. Hence, it is necessary to have a binding for each host programming language that specifies for each attribute type the compatible programming language types. A different binding is needed for each programming language because different languages have different data types. For example, the data types available in C/C++ and Java are different, and both differ from the SQL data types, which are the standard data types for relational databases.

Another problem occurs because the results of most queries are sets or multisets of tuples (rows), and each tuple is formed of a sequence of attribute values. In the pro-gram, it is often necessary to access the individual data values within individual tuples for printing or processing. Hence, a binding is needed to map the query result data structure, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a query result in order to access a single tuple at a time and to extract individual values from the tuple. The extracted attribute values are typically copied to appropriate program variables for further processing by the program. A cursor or iterator variable is typically used to loop over the tuples in a query result. Individual values within each tuple are then extracted into distinct program variables of the appropriate type.

Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model. One example of such a language is Oracle's PL/SQL. The SQL standard also has a proposal for such a database programming language, known as SQL/PSM. For

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 6 | Issue: 8 | August 2020 || Journal DOI: 10.36713/epra2013 || SJIF Impact Factor: 7.032 ||ISI Value: 1.188

object databases, the object data model (see Chapter 11) is quite similar to the data model of the Java programming language, so the impedance mismatch is greatly reduced when Java is used as the host language for accessing a Java-compatible object database. Several database programming languages have been implemented as research prototypes (see the Selected Bibliography).

3. Typical Sequence of Interaction in Database Programming

When a programmer or software engineer writes a program that requires access to a database, it is quite common for the program to be running on one computer system while the database is installed on another. Recall from Section 2.5 that a common architecture for database access is the client/server model, where a client program handles the logic of a software application, but includes some calls to one or more database servers to access or update the data. When writing such a pro-gram, a common sequence of interaction is the following:

When the client program requires access to a particular database, the pro-gram must first establish or open a connection to the database server. Typically, this involves specifying the Internet address (URL) of the machine where the database server is located, plus providing a login account name and password for database access.

Once the connection is established, the program can interact with the data-base by submitting queries, updates, and other database commands. In general, most types of SQL statements can be included in an application program.

When the program no longer needs access to a particular database, it should terminate or close the connection to the database.

A program can access multiple databases if needed. In some database programming approaches, only one connection can be active at a time, whereas in other approaches multiple connections can be established simultaneously.[5]

MODUS OPERANDI

The technology is changing fast, the new paradigms in the database system as well as in the programming languages makes programmers to shift their focus completely, this leads to time consuming to develop new applications further the old applications will be sidelined due to mismatch of supporting files.

There are several GUI interfaces available with database systems which can generate the applications through wizard.

Selecting right Interface for the database system is the critical step, the right methodology is to have an interdependent easy to use interface with a built in wizards for developing database driven applications.

CONCLUSION

The programming languages ar developed for two reasons one is funded by the government of a particular country to take up the new researches and second is due to lack of unavailability of tools makes frustration.

Most of the languages really on third party softwares for database driven application. The users should feel easy to develop applications. In this regard there is a scope to develop an GUI Interface where a normal user can develop an application with ease

REFERENCES

1. *i-manager's Journal on Information Technology, Vol. 2 l No. 1 l December 2012 – February 2013*
2. *Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks Retrieved on June 12, 2012,from http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf*
3. *Sumathi, S., Esakkirajan, S., (2007). Fundamentals of Relational Database Management Systems. (pp. 96-97).Springer-Verlag Berlin Heidelberg.*
4. *The Database Normalization Theory and the Theory of Normalized Systems: Finding a Common Ground - Baltic J. Modern Computing, Vol. 4 (2016), No. 1, 5-33*
5. *http://www.brainkart.com/article/Database-Programming--Techniques-and-Issues_11480/*