ENCAPSULATING A DATABASE WITH SAFE DATABASE METHODS

EYERUSALEM BERHANE HAILE
March, 2011

SUPERVISORS:

Dr. Ir. R.A. (Rolf) de By
Dr. J. M. Morales

ENCAPSULATING A DATABASE WITH SAFE DATABASE METHODS

EYERUSALEM BERHANE HAILE
Enschede, The Netherlands, March, 2011

Thesis submitted to the Faculty of Geo-information Science and Earth Observation of the University of Twente in partial fulfilment of the requirements for the degree of Master of Science in Geo-information Science and Earth Observation.
Specialization: Geoinformatics

SUPERVISORS:

Dr. Ir. R.A. (Rolf) de By
Dr. J. M. Morales

THESIS ASSESSMENT BOARD:

Dr. Ir. R.A. (Rolf) de By(chair)
Dr. Ir. M. van Keulen

ABSTRACT

Several studies have been made in database systems to provide good integrity management, however, it still is difficult to implement all kinds of integrity constraints and enforce them especially in the presence of complex constraints. Not much is done in the field of functional design of databases unlike the structural database design. The purpose of this paper is to present an approach of functional database design to provide a non-constraint violating transaction, by encapsulating the database with a suite of functions that are safe from violating constraints, assuming the database was kept at a consistent state initially, and allowing users to access the database through those functions only.

The methods that are to encapsulate the database are developed in three steps, initially by creating a solid and robust object method, then lift it to become a table method or develop a table method by directly coding it using the object methods, and finally create a solid and robust database method using the table methods. There are four lifting operators developed, three of which are to lift object methods to become table method while one is from table to database method. Each of the methods listed is responsible for controlling constraint maintenance at its own granularity level.

Initially the conceptual design was made using end-user-oriented expressions that are defined using fundamental building blocks of semantics designed to express that intuition of "take the whole dataset leave it intact except for some intended changes". Next a methodology on how to implement the conceptually designed methods in PostgreSQL, using PL/pgSQL is provided. With some of the limitations present in PostgreSQL we have provided some implementation tricks in the methodology. At the end a case study was develop to show how to use the methodology in action.

Keywords

Database methods, lifting operators, encapsulation, constraint maintenance, set returning function (SRF)

ACKNOWLEDGEMENTS

First and foremost I would like to thank God Our Father in the name of our Lord Jesus Christ for everything that he has done in our life.

I wish to express my gratitude to the Royal Netherlands Government for awarding me this Fellowship. I also want to express my sincere appreciation and gratitude to my first supervisor Dr.Ir.R.A. (Rolf) de By, for his endless support, guidance and advice during the development of this research. I am also so grateful to my second supervisor Dr. J.M. Morales for all his co-operation and support.

My heartfelt thanks goes to my family, whose prayers, love, and encouragement have always been with me throughout my study. At last but not least, I would like to thank my friends here or back home for their support, and my classmates from all over the world for the good times we shared together.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

1.1 MOTIVATION AND PROBLEM STATEMENT

Database designing is a very complex and dynamic process that require wide ranging knowledge and understanding [5]. Database is a well-organized collection of data that is related in a meaningful way to represent some segment of the real world. It is used for acquiring timely and relevant information to draw decisions. Therefore it is important that it is protected from any redundant or wrong inputs or updates [14], that can be achieved by using semantic integrity assertions[17]. There are two ways to implement these integrity constraints, by enforcing them during execution time or by developing transaction programs that preserve constraints, assuming the database was consistent initially. Constraints are used for specifying the kind of data that is allowed to be present in the database [22]. Moreover, These integrity constraints are useful in implementing data check, ensuring database consistency, protecting the data from causing ambiguous outcome, and making database applications development easier and more reliable.

The first method, i.e., enforcing the constraints during execution time is a well realized approach [6, 20], however, it has some disadvantages, it consumes considerably high execution time, and it only applies to a small class of integrity constraints. The second method is an approach by which all the constraint checking are made in the transaction program, provided that the database was originally kept consistent, by definition transaction programs should maintain the integrity constraints imposed on a database. The main advantage of this method is only the necessary constraints are cheeked, reducing the overhead compile-time to the least possible and it supports a large group of constraints.

It has been the greatest promise of database systems to provide good integrity management, and several studies have been made, however, it still is difficult to implement all kinds of integrity constraints and enforce them in a database especially in the presence of complex constraints. Several approaches were made to generate integrity preserving transaction methods, among which one was to identify if any integrity constraints were violated in run time before transaction takes place [12]. Another approach was reducing runtime by using a transaction proved not to cause integrity conflict by using a theorem prover [18], and many more approaches were considered too, however, they were either very costly or with the complexity of the constraints and the transactions it was very difficult to acquire a safe transactions. Safe transactions are transactions that does not violate constraints.

The structural design of spatial databases is a fairly well-understood domain [15, 16], but this is not true of functional design of databases. Encapsulating a database with suits of functions which are designed in such a way they don't violate integrity constraints and limiting access of users to be only through those functions is one way to keep the database secure See fig 1.1. Making these functions safe from violating integrity constraints could be achieved by proper design method, in a sense of addressing all constraints these are attribute, object, table, and database constraints respectively. This could be through database structure, triggers of other functional means, explicit privileges, special coding of functional call hierarchy, in which functions are characterized in a

special way. Users access the database for various reasons, granting the appropriate privilege keeps the database safe from errors. The final expected output is to develop a well-documented database

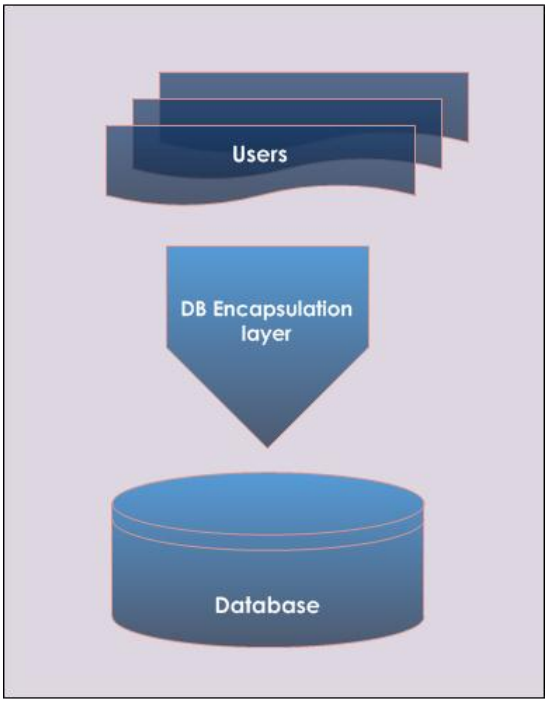

Figure 1.1: Encapsulating a database with functions

design case study as an illustration of correct method to design an encapsulated database. The design should be in the sense defined above, which will precisely describe the method to design functions which will not violet integrity constraints. These methods will grow from attribute method to object method then to table method and finally to database method respectively. In addition the project will work on identifying the method for designing the above process at a conceptual level and a method to transform it to platform-specific functioning, in our case the platform of choice is PostgreSQL.

1.2 RESEARCH IDENTIFICATION

The following issues will be discussed in this research project.

1.2.1 Research objectives

The main objective of this research is to formulate a design approach that allows encapsulating a database in suites of functions which are designed in such a way they preserve integrity constraints. And allow users to access and update the database through those functions or database methods only.

- To understand integrity constraint implementation technique.
- To develop a theory of robust attribute, object, table, and database methods.
- To develop a theory of designing conceptual database functions and their transformation to executable code.

1.2.2 Research questions

1. How are attribute, object, table and database constraints implemented in postgres?
2. what are the methods to develop robust object, table and database methods?
3. Is there a higher level language to conceptually design functions?
4. If there is a higher level language how do we transform this in to platform specific functioning.

1.2.3 Innovation aimed at

The novelty of this research is to design a methodology that will enable users to encapsulate their database in suites of function which are safe from violating integrity constraints. Furthermore, to restrict users to update using those functions only and secure the database from updates that violate the integrity constraints. Our platform of choice for implementing the research is PostgresSQL. Moreover conceptually designing the functions and finding a method to transform them to PSM which will allow interoperability among platforms will be dealt.

1.2.4 Related works

Several studies were carried out to solve the problem in maintaining database integrity and several methods were proposed as well. However, it has become very difficult to solve the issue with the increasing complexity of constraints. Aiming to reduce the amount of compile-time spent due to integrity constraint checking during transaction, Sheard and Stemple [18] used a procedure by which databases are updated using a transaction that does not violate the integrity constraints provided that the database was consistent initially. The transaction was checked by a theorem prover, Isabelle to make sure no integrity conflict arose in the meantime. We can acquire some knowledge from the method they used for producing a safe transaction. Spelt and Balsters [19] outlined a framework of generalizing the idea of automatic verification done by Sheard and Stemple [18], in order to extend it to object-oriented databases using modern theorem proving technology. This has also helped in reducing the need for extensive proof because it is based on the natural deduction method which is build up on Isabelle/HOL package.To create the functions which will be encapsulating the database in the conceptual state we need to uplift the database operators this has already been recommended by [8] to extend the generation of function code in the transformation process for reflecting the conceptual level functions in function code generation, and we can take some transformation procedures from this paper. And a good documentation can be found in the websites for Enterprise architect in [4] and for PostgreSQL in [3].

1.2.5 Methodology

The methodology that we shall follow to design a step by step procedure for users to use to encapsulate their database in PostgreSQL with suite of function which do not violate integrity constraints could be as follows.

1. Study the different options that database technology specifically our platform of choice PostgreSQL has on board that allows us to implement constraints by developing a complete understanding of how constraints can be implemented in this platform.
2. Analyze the database function functionality method, understand the hierarchical layering of designing object method, table method, and then database method respectively.

3. Find a way to design a robust object methods that can be defined and implemented, and develop the object method into table method and the table method to database method all in PostgreSQL coding template.

4. Execute a small but illustrative case study for a database application that demonstrates the feasibility of the methodology produced.

5. Finally try to do it in a conceptual level without resorting to real database programming language, find out if there is a higher level language in which to express this, if that language exist try to identify if we can define transformation that takes that high level language to platform specific coding PL/pgSQL. Although we want to do it in a highly transformational mode but for the sake of implementing it in this project we will work this out in PostgreSQL platform only and we are not looking at other database-platforms.

1.2.6 Structure of the thesis

The coming chapters are structured as follows:

- Chapter 2 this chapter discusses about how integrity constraints are expressed, and the different level at which they are implemented in database. Followed by the conceptual design of the database methods, that encapsulate the database. The basic expressions used to develop the conceptual design are also present in this chapter.

- Chapter 3 introduces the basic statements needed for building those conceptually designed functions in PostgreSQL. Then the methodology on how to implement those database methods using those PL/pgSQL statements is provided.

- chapter 4 is a database design case study, that illustrates the use of our methodology discussed in the previous chapters.

- chapter 5 presents the research achievement in the discussion and conclusion part, and it also provides some recommendations for future work.

Chapter 2

Integrity constraints and conceptual design of the functional database design

A database is a representation of some part of the real world, and how well it represents the real world depends on its fitness for the intended application and its capacity to encompass the continuous change in the world without violating the integrity of the datasets. The logical consistency of data in a database is identified by defining consistent database states. Several research efforts have been made to study the basic logic, architecture and implementation of constraints, and it is stated that an improved data consistency is accomplished by specifying the constraints that must hold in the database using integrity constraints [21]. Furthermore database consistency is kept by performing updates only through transactions that do not violate constraints assuming the database was consistent initially.

In this chapter we will look at how constraints are expressed and the different levels at which they are implemented in a database. This will guide us into developing simple functions (or methods) for producing non-constraint violating transactions. Based on the constraint implementation those functions are further decomposed into smaller parts of object, table and database methods. In addition we will look at the class universe for each granularity level and provide the conceptual design of those methods as well.

2.1 INTEGRITY CONSTRAINTS

Integrity constraints are logical statements that limit the set of allowable relations in a database. They are specified either in the schema and are checked automatically or they are expressed in the application program using a constraint language, and are checked during the creation of the database upon data entry or during transactions, i.e., when the database changes from one state to the next [21]. Depending on this at an abstract level constraints are classified into: i) inherent constraints and ii) explicit constraints [13]. However according to [10] depending on where the integrity constraints are applied they are classified into: i) inherent constraints, ii) implicit constraints and iii) explicit constraints See Figure 2.1. Both classifications by the two authors [13] and [10] are the same in the explicit constraint definition except the class inherent constraint by [13] is divided in to implicit and inherent constraint by [10] definition.

2.1.1 Levels of Integrity constraints

- Inherent (or structural) constraints are rules that define the data model constructs and are enforced by the system automatically. For instance since a relational model is based on mathematical set theory it has the inherent constraint that state no duplicate tuples are allowed in a relation, hence primary key constraints are valid. In the same way not null, foreign key and referential integrity are all examples of this kind [13]. This information need not be written in the schema as there definition will be inherited from the data model. To make it more clear for example a string attribute field can not be used for geometry

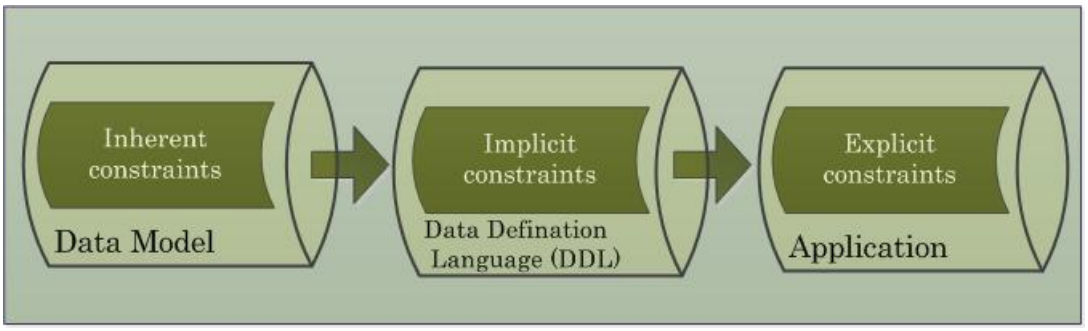

Figure 2.1: Constraints classification based on where they are applied, (Adopted from [10]).

calculation this information need not be written in the schema as it will also be inherited from the data model.

- Implicit constraints are constraints that are specified during the creation of the database schema using the Data Definition Language (DDL). They are stored in the DBMS catalog to be enforced automatically every time there is an update [7]. These constraints should be specified on individual relations. Primary key, not null, foreign key and referential integrity are examples of this kind [9]. This is more like specifying the constraints to be enforced using the inherent constraints.

- Explicit (behavioral) constraints are those constraints tangential to the data model. They are managed by the database designer or the application programs; they are beyond the scope of the DDL [13]. Further action needs to be taken to ensure their non-violation, which is specified in a procedural approach or declarative approach. The procedural approach is a method that involves specifying the checks for the updates on the database, whereas declarative approach is a formal method of storing constraints in a suitably encoded form. These could be achieved through roll-back, triggers, or explicit transaction checks [10, 21].

2.1.2 Integrity constraints classification

Explicit integrity constraints can also be further classified into i) Static or state constraints and ii) Transitional constraints .

1. Static integrity constraints are rules that the database must satisfy in every single state of the database. They express a valid state of the database. [7] stated that static constraints can further be brokedown into domains, entity integrity rules, attribute structural constraints, referential integrities, superclass/subclass and other user-defined constraints.

2. Transitional constraints are rules that apply to sequences of database states, and are mainly concerned on the implementation domain. These constraints are rules that the database enforces to restrict the data from entering an impossible state due to its previous states. [11] introduced dynamic constraints as another group of integrity constraint, however, there is an overlap with the definition of the transitional constraints therefore are considered in a single group. An example of transitional constraint can be if a person's state was married, the valid transaction should be to change its state to "divorced" or "widowed" but not to "single or never married" [10].

2.1.3 Integrity constraint taxonomies

There are two useful integrity constraint taxonomies.

Based on support level by the DBMS :

Implicit/ user-defined constraints.

1. User-defined integrity constraints are constraints usually involving arithmetic operators, and are mainly used to enforce company regulations; e.g. the constraint for some employee database that the sum of the salary plus the commission an employee receives should not be greater than the salary of that employee's manager is a user defined constraint [13].

2. Implicit constraints are defined by the concept of the underlying database model implicitly and need not be specified explicitly [11]. They are made during the need analysis state, i.e., a discovering process wherein designers discover critical insights into the design task, and they are very important and difficult to recognize [13]. They are found by making a life-cycle environment study and making the order of magnitude calculations.

Based on data structure granularity :

Attribute/ tuple/ relation/ database constraints [11]

1. Attribute or domain constraint specifies the domain of possible values an attribute can have independently of other attribute values of the tuple. It can further be restricted by conditional or existence constraints. Not-null is a typical example.

2. Object or tuple constraint applies to a tuple in a relation without reference to other tuples in the relation. It restricts the combinations of attribute values in the same tuple.

3. Relational or table constraint limits the value that a set of tuples can have. It applies to a single table without referencing to other tables in a database. Functional dependencies, multivalued dependencies, key constraints, aggregate constraints are examples of this type.

4. Inter-relational or database constraints apply to a table in reference to other tables in the same database. It restricts the content of the table in relation to other tables in the database. Well known representatives are referential integrity (foreign key), inclusion dependencies, and redundancies.

2.2 FORMAL SEMANTICS DESCRIBED INTUITIVELY

This subsection was provided by my Supervisor (Dr.Ir.R.a.(Rolf) de By) as a formal language basis to start from. The formal semantics of the expression forms that we propose is defined in such a way that it allows us to express the intuition "take the whole dataset,make the indicated local change, and return the dataset thus obtained." We follow the classical set-up of type theory. Our language has expressions, members of a set E, and types, members of a set T. Examples of expressions are: SD, 2, $25 + 12$, $SD.T_1$ and so forth. Examples of types are: integer, real, $\langle a : \text{integer}, b : \text{real} \rangle$, and so forth. Observe that record types are explicitly included, as we need them to formally describe datasets and tables in datasets.

There is an important relationship between E and T, known as the well-typedness relationship, and denoted as ":". When we write $44 : \text{integer}$, we are saying that the expression 44 is

well-typed, and has type integer. We thus know that $: \subseteq E \times T$. Obviously, not all expressions have all types, but some expressions may have multiple types. No examples given here.

Any formal language that uses type theory for its precise definition does not only define E, T and the well-typedness relation : between them, but also defines a formal semantics, often expressed as $[[\ldots]]$ of expressions $e \in E$ and types $\tau \in T$. So, $[[e]]$ is the formal semantics of e, and $[[\tau]]$ is the formal semantics of τ. An expression semantics is normally a value, while a type semantics is normally a set of values. Whenever the rules of our language say that $e : \tau$ holds, one is assured that we also have $[[e]] \in [[\tau]]$, i.e., the semantics of an expression is a member of the semantics of its type.

Important is to understand that a type, like integer, has a formal semantics written as [[integer]]. In this case, that $[[\ldots]]$ semantics of the type is a set of values. With integer, that set of values is $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. One can assume that all basic types have such semantics.

We need to say something specific about the semantics of records and record types. Above, we used as an example record type

$$\langle a : \text{integer}, b : \text{real}\rangle,$$

a type that identifies two attributes a and b, with respective types integer and real. An example record of that type is the following record r:

$$r \stackrel{\text{def}}{=} \langle a = 44, b = 3.14159265\rangle.$$

Observe that these are valid statements in our formal language.

What now is the semantics of that record r, and what is the semantics of its type? We assume that the attribute names for records and their types are drawn from a previously defined name set A, and so in the example case we have $a \in A$ and $b \in A$. the trick to look at records semantically, is by considering them functions that map attribute names nicely onto values. In the case of record r, that function is $[[r]]$, for which we know the domain:

$$dom[[r]] = \{a, b\},$$

which is a subset of A, obviously. The function maps to the semantics of expressions associated with the attribute names:

$$[[r]](a) = [[44]] \text{ and } [[r]](b) = [[3.14159265]].$$

This discussion leads us intuitively to the semantics of record *types*. Such a semantics can only be a collection of functions of the nature of $[[r]]$. And we can thus define the semantics of r's type as follows:

$$[[\langle a : \text{integer}, b : \text{real}\rangle]] \stackrel{\text{def}}{=} \left\{ f \;\middle|\; \begin{array}{l} f \text{ is a function} \wedge \\ dom(f) = \{a, b\} \wedge \\ f(a) \in [[\text{integer}]] \wedge \\ f(b) \in [[\text{real}]] \end{array} \right\}$$

These examples naturally generalize to arbitrary records and record types. We needed to discuss this because below we are going to define additional expression formats that can only be precisely defined once we understand the set-up sketched above.

2.2.1 Basic expressions needed

This section lists a number of expressions that are needed to define schema transformation expressions. They provide the fundamental 'building blocks' to construct larger, end-user-oriented expressions. Assume $r : \langle a_1 : \tau_1, \ldots a_n : \tau_n \rangle$ is a record, assume that R_1 and R_2 are tables of type $I\!P\langle a_1 : \tau_1, \ldots a_n : \tau_n \rangle$ with primary key $\{a_{i_1}, \ldots, a_{i_h}\}$, and that S, T and U are union-compatible value sets. We will use the notation keys(R_i) to denote the set of primary key values present in table R_i.

1. The expression

$$r \textbf{ except } a_j = e_j, \ldots, a_k = e_k \quad \text{where } \{a_j, \ldots, a_k\} \subseteq dom[[r]] \tag{2.1}$$

   denotes a record r' obtained by copying r and replacing its attribute value for a_j by the value of e_j, and so on, until the attribute value for a_k, which value is replaced by the value of e_k. The except expression allows local attribute value overwriting. The formal semantics of this expression is a function shaped like $[[r]]$ that maps onto different semantic values for the listed attributes.

2. The expression

$$r \textbf{ dropatt } a_j, \ldots, a_k \quad \text{where } \{a_j, \ldots, a_k\} \subseteq dom[[r]] \tag{2.2}$$

   denotes a record r' obtained by copying r and removing from it the listed attributes. The formal semantics of this expression is a function shaped like $[[r]]$ but with a reduced domain, mapping onto the same values for remaining attributes.

3. The expression

$$r \textbf{ addatt } a_j = e_j, \ldots, a_k = e_k \quad \text{where } \{a_j, \ldots, a_k\} \cap dom[[r]] = \emptyset \tag{2.3}$$

   denotes a record r' obtained by copying r and adding to the record the named attributes with associated values. The formal semantics extends that or r by adding to the domain of the function, and mapping onto the values of the respective expressions.

4. The expressions

$$S \textbf{ union } T, S \textbf{ setminus } T, S \textbf{ intersection } T \tag{2.4}$$

   denote the regular and well-known set-theoretic expressions.

5. The expression

$$R_1 \textbf{ exceptset } R_2 \quad \text{where keys}(R_2) \subseteq \text{keys}(R_1) \tag{2.5}$$

   denotes a set R' obtained by copying R_1 and replacing those records that have a key in keys(R_2) by the corresponding records in R_2. Observe the similarity with the except expression, and the fundamentally different semantics.

6. The expression

$$R_1 \textbf{ addset } R_2 \quad \text{where keys}(R_1) \cap \text{keys}(R_2) = \emptyset \tag{2.6}$$

denotes a set R' obtained by copying R_1 and adding to it the records in R_2. Observe that the addset expression has a semantics similar to that of union, however, that its use comes with a precondition of non-overlapping key sets.

7. The expression

$$R_1 \textbf{ dropset } R_2 \quad \text{where keys}(R_2) \subseteq \text{keys}(R_1) \tag{2.7}$$

denotes a set R' obtained by copying R_1 and removing from it those records that have a record in R_2 with matching primary key. Observe that the dropset expression has a semantics similar to that of setminus, however, that its use comes with a precondition of key set inclusion.

8. Assume U denotes a set, u is a variable, ϕ is a predicate and μ an arbitrary expression. The predicative set expression

$$\{\ u \in U | \phi(u) \bullet \mu(u)\ \}, \tag{2.8}$$

denotes the set obtained from U, letting u range over its members, and producing those $\mu(u)$ values for which $\phi(u)$ tests true. This expression is a combination of a filter (ϕ) and a map (μ), both of which are only optional parts of the expression.

The filtermap expression naturally generalizes to the use of multiple sets, as follows:

$$\{\ u \in U, \ldots | \phi(u, \ldots) \bullet \mu(u, \ldots)\ \}. \tag{2.9}$$

We may be needing more forms later, but we will leave it here for now. What we might be needing is a bag extension to the above where we have only provided notation and expressions for handling sets.

The reader should observe that many of the above expression forms have an approach of copying the original r or R, and leaving it as much intact as possible, applying only local change.

2.3 FORMAL STRUCTURE OF THE DATABASE IN FUNCTIONAL DATABASE DESIGNING

The structural design of databases is a well-known science and several researches have already been done about it; however, about the functional design approach, which is a very useful method for producing a non-constraint violating transactions, no much has been done yet. In this and the coming sections of this chapter we will try to look at the conceptual design, i.e., the concept of functional database design without considering the available technologies. We will start by defining the class universes and next we will continue to look at the concepts of the encapsulating methods design.

Conceptual design of a database specifies the classes of objects, the relationships among them, the database rules, i.e., the database integrity constraints amongst the datasets, and the intended application, i.e., the description of intended query and transactions. The way to get a consistent database is by first defining the consistent database states and ensuring the initial state of the database to be consistent and finally ensuring no transaction violates it.

In principle, to find a constraint violation minimally the following checks need to be made based on their granularity:

- For attribute constraint violation a single attribute needs to be checked.
- For object constraint violation a single object needs to be checked.
- For table constraint violation only a single table needs to be checked.
- For database constraint violation a collection of tables or all the tables in the database needs to be checked.

Based on this principle a functional design approach to develop functions that encapsulate the database that perform transactions that will not violate integrity constraints can be formulated. This design approach would be a method that has one big function composed of several other small methods. Each method will be at different granularity level and will control constraint violation on its level.

Building the structure of the database from ground up: a class universe states all the datasets that are eligible to be in that class. The logical expression that states the class universes at each granularity level contains, the data type of each class and the constraints they should obey at each level. The levels are object, table and database, and their class universes are defined as follows:

2.3.1 Attribute universe

Attribute universe U_a states all the values that can be a member of the values of an attribute, it is defined by stating the datatype of the attribute and the constraints associated with it. It is formulated as :

$$U_a = \{e : \tau' | \delta(e)\},$$

where e represents an attribute state of attribute a. It is read as, for all attribute e member of attribute a, a's attribute universe U_a is defines as those attributes having datatype of τ' and obey the rule $\delta(e)$.

2.3.2 Object universe

Knowing the attribute universe, an object class universe U_C contains all those objects that can be a member of the class,it is defined by stating the datatype of the object class and the object constraints associated with it in a similar way as the attribute universe. An object datatype τ is defined as the list of all attributes in an object along with their datatypes:

$$\tau = \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$$

Object constraints are constraints that are applied between attributes of the same objects, they are represented by ϕ. Assuming availability of some logic to express constraints, it will be discussed in later section. Therefore U_C is defined as:

$$U_C \ = \ \{t : \tau \mid \phi(t)\},$$

where t (for tuple) represents an object state of class C.
The expression is read as for each object t member of class C, C's class universe U_C is defined as those objects of type τ that obey the rule $\phi(t)$.

2.3.3 Table universe

Knowing the attribute and object class universes, the table universe T_C for each class can be defined by providing the table's data type and the constraints available, and its formulated as follows: tbl for table, which represents a table state, is a collection of object or class C, having a datatype set of objects, represented as:

$$tbl : I\!P\tau,$$

where $I\!P$ is a power type operator and τ is object datatype See Section 2.3.2.
The criteria for an object t to be a member of the table universe T_C is t should be a member of the class universe U_C, and the table tbl should still obey the table constraint $\Phi(tbl)$ after t is being added to it. U_C is defined in Section 2.3.2.

$$T_C = \{tbl : I\!P\tau | tbl \subseteq U_C \wedge \Phi(tbl)\}$$

The table constraint $\Phi(tbl)$, which might be needed in addition to the object constraints such as key uniqueness, primary key or any other set constraint on the table are given in $\Phi(tbl)$.

2.3.4 Database universe

A database state is a consistent collection of table states that obey the database constraint. Knowing the object and the table universes, the database universe DB for each object t can be defined by adding a check on the inter-relational or database constraint on the list of tables which are members of there own table universes:

$$DB = \{tbl_1 : I\!P\tau_1, \ldots, tbl_n : I\!P\tau_n \mid \ tbl_1 \in T_{C_1} \wedge \ldots \wedge tbl_n \in T_{C_n} \wedge \Psi(tbl_1, \ldots, tbl_n)\}$$

tbl_i represents a table state for class C_i for all i. It is a member of the table universe T_C and its datatype is $I\!P\tau_i$. The database constraint Ψ is a predicate that takes all the tables in the database as an argument $\Psi(tbl_1, \ldots, tbl_n)$ and make database constraint checks. Assuming there is a logical expression for Ψ, it is used to express across-table rules such as referential integrity and more.

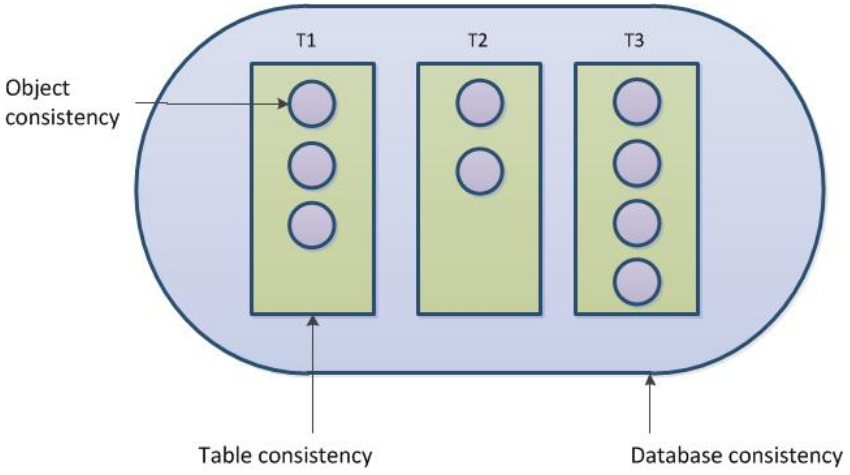

Figure 2.2: Database consistency at different granularity level

2.4 DEVELOPING THE METHODS THAT ENCAPSULATE THE DATABASE

Database consistency is built up at different level of granularity and it can also be destroyed at different level of granularity (see Figure : 2.2). A set of consistent objects is not necessarily a consistent set; a collection of consistent tables is also not necessarily a consistent database. To make an update on a single object, a properly designed transaction would be one that checks object constraint, table constraint, and database constraint before the transaction takes place. Assume we change

$t_1 : C_1 \in tbl_1 \ to \ t_1^{'} : C_1$

Which is read as: change object t_1 which is an element of table tbl_1 and has datatype C_1 with an object $t_1^{'}$ having the same datatype C_1. The required checks before committing this transaction are:

1. Object constraint checks if $t_1^{'} \in Uc_1$.
   Where Uc_1 is object t's class universe.

2. Table constraint Φ checks if the table after updation will remain consistent, we call the table with the new object values $tbl_1^{'}$. The table constrain will be $\Phi(tbl_1^{'})$; however, the value of $tbl_1^{'}$ varies in cases of update, insert and delete. See each cases below:

   - Update $tbl_1^{'} = tbl_1$ exceptset $\{t_1 = t_1^{'}\}$
   - Insert $tbl_1^{'} = tbl_1$ addset $\{t_1^{'}\}$
   - Delete $tbl_1^{'} = tbl_1$ dropset $\{t_1 = t_1^{'}\}$

3. Database constraint checks if $\psi(tbl_1^{'}, tbl_2, ...tbl_n)$ holds. Where $tbl_1^{'}$ is the table whose object is changed and still $\Phi(tbl_1^{'})$ holds, and $tbl_2, ...tbl_n$ are list of all the tables in the database.

Here we look at a more complex case by which we have several updates in a single transaction. For example, lets say we have a database with 5 tables tbl_1 to tbl_5, and we want to make the following transactions:

- tbl_1 exceptset R_1
- tbl_2 addset R_2
- tbl_3 dropset R_3
- tbl_4 exceptset $\{r = r$ except $a_j = e_j, \ldots, a_k = e_k\}$

The required checks before committing this transaction are:

1. Object constraint check it is the same predicate for all objects with in the same table. It is needed for the update and the insert functions only, not useful for the delete function. It checks whether the new values to be inserted are elements of the class universe U_C of the table they are going to be inserted in. In our example it checks whether R_1's and R_2's records are all members of tables tbl_1's and tbl_2's object class universes respectively, and in the case of tbl_4 it checks if the record r after its attribute values are changed to $a_j = e_j, \ldots, a_k = e_k$ would still remain to be a member of the class universe of the table tbl_4. For the dropset function the only criteria is if the primary key of R_3 is element of the primary key set of tbl_3. There is no need of object constraint check.

2. Table constraint Φ is different for all tables in a single database. It checks whether the tables after the transaction would still remain consistent. The tables tbl_1, tbl_2, tbl_3 and tbl_4 "after the transactions are made" are represented by tbl'_1, tbl'_2, tbl'_3 and tbl_4 respectively. The table constrain checks if $\Phi_1(tbl'_1)$, $\Phi_2(tbl'_2)$, $\Phi_3(tbl'_3)$ and $\Phi_4(tbl'_4)$ holds. Here when we say "after the transaction", it does not mean it is already committed in the database, however we get the result from the table method on the fly, we will see how we can get the values at run time in Section 2.4.1.

3. Database constraint checks if $\psi(tbl'_1, tbl'_2, tbl'_3, tbl'_4, tbl_5)$ holds, where ψ is a predicate that specifies the database constraints. It checks whether the updated tables when brought in to one database still obey all the inter relational constraints. The list of expressions with primes represent the tables with the new updated values. This is also done at run time. After the three levels of constraints check, then all the transactions can be committed depending on the outcome of the checks.

2.4.1 Methods without constraint checking

The following section discuss how to build database methods that encapsulate the whole database and check the overall consistency before committing the transaction. A database method is composed of object method, table method, and database method that are responsible for constraint checking their own level. The methods to be composed are chosen based on constraint data structure granularity (as discussed in Section 2.1.3) and the principle of the minimum amount of data needed to be checked for constraint violation check (discussed in Section 2.3). This brings efficiency in the development and maintenance of transaction programs while working with large programs having complex transactions with complex constraints. Though our logic is based on the availability of constraints the following sections addresses the pseudocode for the various methods as if there are no integrity constraints to be verified. The pseudocode for methods with constraint checking will be addressed in Section 2.4.2.

Object methods without constraint checking

An object method M is a function that takes an object as an argument and returns the object with some or all of its attribute values changed. It will take a record t type τ, and return the object intact, except for the attributes $a_i, \ldots, a_j$ replaced by the respective given values of $e_i, \ldots, e_j$. The algorithm would be as follows:

M($t : \tau$) : τ
begin
  return t except $a_i = e_i, \ldots, a_j = e_j$
end

Object datatype τ is defined by listing the attributes with their datatype, for which See Section 2.3.2. Object methods are protected functions, i.e., a function that will not be affecting the database. It will only be used for modularity purpose to return a value which can be used later for object constraint checking and to develop the table method by being lifted up to table method. To develop the table methods first we need lifting operators. Some lifting operators and their mechanisms will be defined and discuss in the next section.

Lifting operators

Lifting operators are meta-functions that allow us to apply or map other functions to all or some members of a list. Object methods cannot do anything by themselves, they need to be lifted to

became a table method and eventually a database method. Table methods cannot do anything by themselves either, as a result they too need to be lifted to become a database method. Database methods are the only methods that can be executed to have impact, i.e., change, of the database state.

We define four kinds of lifting operators, three object method lifting operators and one table method lifting operator. The first three that are used to lift object methods to table methods are explained as follows while the last one, i.e., the table lifting operator will be discussed in the coming sections.

Singular lifting operator (S) : It is applicable for lifting an object method to table method to define the case of a single object value change with in a table. It is an operator that takes as an argument an object method M, an object t of type τ and a table tbl of type $\mathbb{P}\tau$, where we will normally assume that $t \in tbl$. The return value of S is the table tbl itself unchanged except the value of t changed to t' that is obtained by mapping the function M to t.

$$S(M, t, tbl) \stackrel{\text{def}}{=} tbl \text{ exceptset } \{M(t)\}$$

The algorithm of the lifting operator S that would lift the object method M stated above to a table method is formulated as:

S (M, t, tbl) : $\mathbb{P}\tau$

begin

return tbl exceptset $\{M(t)\}$

end

$S(M, t, tbl)$ gives an anonymous table method that takes a table tbl of type $\mathbb{P}\tau$ as a parameter and returns the table tbl intact except one object changed. The algorithm is as follows:

tblmethod $(tbl : \mathbb{P}\tau,\ t : \tau)$: $\mathbb{P}\tau$

begin

return tbl exceptset $\{M(t)\}$

end

If an object method M that takes an object $t : \tau$ need additional parameters $e_i : \tau_i, \ldots, e_j : \tau_j$ then lifting it to a table method using S would look like:

$$S(M, t, tbl, e_i, \ldots, e_j) \stackrel{\text{has code body}}{=} tbl \text{ exceptset } \{M(t,\ e_i,\ \ldots, e_j)\},$$

where $M(t, e_i, \ldots, e_j) \stackrel{\text{has code body}}{=} t$ except $a_i = e_i, \ldots, a_j = e_j$

Universal lifting operator (A) : this operator is useful to lift object method to become table method in time of updating all the objects of a table using a single object method. The intention of A is to map the object method M on all objects of the table tbl. A is an operator having the following as an argument, an object method $M(t : \tau)$, an object t of

type τ and a table *tbl* of type $\mathbb{P}\tau$ where we assume $t \in tbl$, and it returns the table *tbl* with all its objects t changed by t' by mapping M to all $t \in tbl$.

$$\mathrm{A}(M, t, tbl) \stackrel{\text{def}}{=} \{t \in tbl \bullet M(t)\}$$

The algorithm of the lifting operator A that would lift the object method M to become a table method is:

A (M, t, tbl): $\mathbb{P}\tau$

begin

return $\{t \in tbl \bullet M(t)\}$

end

If an object method M that takes an object $t : \tau$ need additional parameters $e_i : \tau'_i, \ldots, e_j : \tau'_j$ then lifting it to a table method using A would be almost the same as those with just $t : \tau$ argument, it is formulated as follows:

$$A(M, t, tbl, e_i, \ldots, e_j) \stackrel{\text{has code body}}{=} \{t \in tbl \bullet M(t,\ e_i,\ \ldots, e_j)\},$$

where $M(t, e_i, \ldots, e_j) \stackrel{\text{has code body}}{=} t$ except $a_i = e_i, \ldots, a_j = e_j$

Filtered lifting operator (F) : this is used to lift an object method to become a table method in times of updating selective objects from a table based on some criteria expressed as a predicate $\varphi(t)$. The intention of F is to lift M to become a table method by mapping the object method M on some objects *t* of the table *tbl* for which the predicate $\varphi(t)$ holds where $\varphi(t)$ is the selection criterion.

$$\mathrm{F}\ (M, t, \varphi, tbl) \text{ returns } tbl \text{ exceptset } \{t \in tbl \mid \varphi(t) \bullet M(t)\}$$

The algorithm of the lifting operator F is as follows:

F (M, t, φ, tbl): $\mathbb{P}\tau$

begin

return *tbl* exceptset $\{t \in tbl \mid \varphi(t) \bullet M(t)\}$

end

If the Object method M that takes an object $t : \tau$ need additional parameters $e_i : \tau'_i, \ldots, e_j : \tau'_j$ then lifting it to a table method using the filtered lifting operator F would also be more or less similar to those with just $t : \tau$ argument, it is formulated as follows:

$$F(M, t, tbl, e_i, \ldots, e_j) \stackrel{\text{has code body}}{=} tbl \text{ exceptset } \{t \in tbl \bullet M(t, e_i, \ldots, e_j)\},$$

where $M(t, e_i, \ldots, e_j) \stackrel{\text{has code body}}{=} t$ except $a_i = e_i, \ldots, a_j = e_j$

Table methods without constraint checking

We have seen how table methods are developed from object methods using the lifting operators above it can also be produced by direcly codding it without the lifting operators. A table method is a function that takes a table and either returns the table with some of its objects changed, removed or some new objects added to it. We will look at those three cases independently, where R is a set of objects and r is a single record.

- An update table method ($tblmethod_u$) takes table $tbl : I\!P\tau$, and it returns the table tbl intact except some of its objects changed by another set of objects R having the same identity, and the same data type set of objects $R : I\!P\tau$.

$$\text{tblmethod}_u(tbl : I\!P\tau, R : I\!P\tau) \overset{\text{has code body}}{=} tbl \text{ exceptset } R$$

$$\text{tblmethod}_u(tbl : I\!P\tau, r : \tau) \overset{\text{has code body}}{=} tbl \text{ exceptset } \{t = r\}$$

- Delete table method $tblmethod_d$ takes table tbl with data type set of objects ($tbl : I\!P\tau$), and it returns the table tbl intact except one or more objects deleted. The values to be deleted could be based on some criteria or some listed set of values.

$$\text{tblmethod}_d(tbl : I\!P\tau, R : I\!P\tau) \overset{\text{has code body}}{=} tbl \text{ dropset } R$$

$$\text{tblmethod}_d(tbl : I\!P\tau, r : \tau) \overset{\text{has code body}}{=} tbl \text{ dropset } \{t = r\}$$

- Insert table method $tblmethod_i$ takes table tbl with data type set of objects ($tbl : I\!P\tau$), and it returns the table tbl intact except some objects added to it. The values to be added could be either single object r or set of objects R provided they have the same primery keys.

$$\text{tblmethod}_i(tbl : I\!P\tau, R : I\!P\tau) \overset{\text{has code body}}{=} tbl \text{ addset } R$$

$$\text{tblmethod}_i(tbl : I\!P\tau, r : \tau) \overset{\text{has code body}}{=} tbl \text{ addset } \{t = r\}$$

All the above mentioned kinds of updates can be developed using the three lifting operators or by directly coding them. The pseudocodes for the various table methods in cases of updating, deletion and insertion of single or several objects can be formulated according to the mathematic expressions stated above. The table methods also need to be lifted to become a database method or be called inside a directly coded database method, otherwise it can not make any change to the database, because it only returns table or set of objects.

Database methods without constraint checking

Database method is the only externally public function, and is is responsible for committing the transactions and guaranteeing overall consistency. It takes the whole database as an argument, i.e., the set of tables, and execute updates, insertions or deletions depending on the various cases, and return a message that inform the success or failure of the transaction. It provides the whole database with the updated values at runtime in order to feed it into the database constraint predicates, then the transaction in the database will be committed, depending on the result of the predicate. We will look at the constraint checking in detail in the next section.

Database methods are developed either by directly writing the code or by lifting a table method to become a database method using a table lifting operator. When directly coding a database method the table methods are called on the tables whose values need to be changed, inside the function code. The algorithm for a database method D that takes a database of type Δ, and that commit the transaction and return a message of confirmation of either a success or failure of the transaction would be as follows. The function also includes producing in run-time the database intact except for some of its tables tbl changed, for constraint checking purpose.

D (db, tbl) : text
begin
$db :=\ db$ except $tbl = tblmethod(tbl)$
update tbl set...
if found then
return 'transaction completed successfully';
else raise exception 'transaction failed'
end if;
end

The other approach by which database methods are developed is using lifting operator. Here only one kind of table lifting operator is needed to lift a table method to become a database method unlike the object lifting operators. This is due to the variation of table universes among tables of a database while object universes are the same for all objects in a table. As a result one table method can be applied only to a single table in a database, i.e., different tables require different table methods. Therefore universal or filtering operators are irrelevant at this stage.

Table lifting operator (T) : It is useful to lift a table method to become a database method. It is similar to the singular lifting operator (S) in a way since both are applied to map a function on a single value from the set of inputs provided. T is an operator that takes a table method $tblmethod$, a table tbl and a database db as an argument where we assume $tbl \in db$ and it returns the database intact except for the table tbl changed by tbl' obtained by mapping the table method to tbl.

$$\text{T}\ (tblmethod, tbl, db) \overset{\text{has code body}}{=} db \text{ exceptset } \{tbl = tblmethod(tbl)\}$$

The database method could also need additional input parameters, the algorithm of T is formulated as:

D $(db, tbl_i, \ldots, tbl_j, t_i, \ldots, t_j, e_i, \ldots, e_j)$: text
begin
$db :=\ db$ exceptset $\{tblmethod(tbl_i, t_i, e_i), \ldots, (tblmethod(tbl_j, t_j, e_j)\}$
update tbl set...
if found then
return 'transaction completed successfully';
else raise exception 'transaction failed'
end if;
end

The final database method responsible for all the constraint checking and committing the transaction is formulated in such a way that it uses the return values of the object method, table method and the database produced by the database methods in run-time to check for integrity constraint violation. These return values will be an input for the predicates defined to check integrity constraints.

2.4.2 Methods with constraint checks

In the above sections we looked how the methods are grouped in three different levels, one calling the other and at the end the the database method committing the transaction. However, in addition to this at each level computing the conditions that must hold prior to updating must be considered, to ensure no constraint violations occur. Generally speaking this is done by calling a predicate in the if clause for checking the list of criteria. The predicate would take attribute, object, table or database as an argument, depending on the constraint level, and return boolean(true or false). To minimize run time spent for constraint checking due to unnecessary repetition, the object methods that has been called on the same objects should be organize in to a single group and must be provided with only one constraint check for all of them, the same holds for the table and database methods too.

The methods, provided above in cases of no constraint check (Section 2.4.1) depicts that the return values of the methods are object, table and text message for the object, table and database methods respectively, furthermore, the database method returns the whole database at run time. These return values are the means for the constraint checking predicates to have access to the whole database indirectly. Here the whole updated database is returned to be an input for the predicates therefore the predicates will check if the database will remain consistent after the transaction took place. If a positive result is found from the predicates then the transaction will be committed, However if there is a violation it give an error message and exits the transaction. This can be done in two ways.

Method one is to report the error one at a time i.e., at the first violation it exits the transaction. Error could be from the object method, table method or database method, at any of these levels when violation occurs it report the error message and exits the transaction. While the second approach is to attach an error message to each predicates and raise notice each time there is a violation till the end of the transaction program and finally abort the transaction without committing. The second method is better in most of the cases because it enables mass correction to the transaction program. However, sometimes the errors might be merging due to the first violation that by just editing only the first the other might be solved, i.e., violation in object methods can be propagated to the table method and then to database methods. Thus depending on the scenario it is good to consider both approaches.

Example lets say we have an object method M, that takes an object t type τ and returns an object of type τ with changed attributes, inside the function there are predicates that take an object and checks for object constraint violations the algorithm for this will be.

- According to the first method, it will be coded with raise exception. When a violation occurs it will exit the transaction and send an error message.
  M(t:τ) : τ
  Begin
  $t := t\ except\ a_1 = v_1, ...a_n = v_n$
  if $predicate\_1(t) \wedge ... \wedge predicate\_n(t)$ **then**
  return t
  else

$raise\ exception\ 'constraint\_1\ violated'$
end if

End

- According to the second method, it will be coded with raise notice. It will not exit the transaction but it produce the error messages.
M(t:τ) : τ
Begin
$t := t\ except\ a_1 = v_1, ...a_n = v_n$
if $predicate\_1(t) \wedge ... \wedge predicate\_n(t)$ **then**
$return\ t$
else
if $predicate_1(t) = false$ **then**
$raise\ notice\ 'predicate\_1violated'$
if $predicate_n(t) = false$ **then**
$raise\ notice\ 'predicate\_nviolated'$
end if
end if
Using our methodology both static and transition constraints can be checked, by coding the right predicate for all. The predicates are programmed following the standard programming language, to define them we can use arithmetic operators, comparison operators, aggregate functions, and logical connectors etc depending on the implementation platform. They take attribute, object, table of database as an argument and return boolean, after checking the consistency of the database. Error messages can be attached to them when they return false otherwise a message which include the successful checking could be generated. In the next sections we will see what the methods look like when the predicates are included.

Object methods with constraint checking

As discussed in Section 2.4.1 an object method takes an object and additional optional parameters as an argument and it returns an object. This return value will be an input for the predicates defined for checking the object consistency. Attribute constraints are also checked using this return value. Furthermore, to check transitional constraints we can have both the previous object value and the updated object value, therefor with a proper predicate definition this also can be checked. If the IC predicates hold, then the object method will return the new object with the new values, however, if it does not hold then we can choose between the two error message discussed above. We can choose to exit the transaction, or we can send an error message and still return the new object.
An object method M takes a record t type τ, and return the object intact, except for the attributes $a_i, \ldots, a_j$ replaced by the respective given values of $e_i, \ldots, e_j$ when the object constrain ϕ returns true and it returns an error message when false. The algorithm would be as follows:
M(t:τ) : τ
Begin
$t := t\ except\ a_1 = v_1, ...a_n = v_n$
if $\phi(t)$ **then**
return t

else
$raise\ exception$ '$object\ constraint\ \phi\ violated$'
end if
End
For example, if we have object t with tuples of $\langle id : int,\ pid : int,\ geom : geometry\rangle$ and constraint that states pid should be greater than id then our object constraint checking function (ϕ) is formulated as a function that takes an object and return boolean.

- The pseudocode for object IC predicate ϕ.

  $\phi(t : \tau)$: boolean

  begin

  t.id >t.pid

  end

- Then the pseudocode for an object method that update the value of pid by twice its value would be:

  M(t:τ) : τ

  begin

  t:= t except t.pid = 2*t.pid

  if $\phi(t)$ **then**
  return t
  else
  $raise\ exception\ '\phi\ is\ violated'$
  end if

  end

Table methods with constraint checking

The approach is more or less similar to the object method with constraint checking except the predicates here take a table as an argument and check for the table consistency. The design include the table method without constraint checking as described in Section 2.4.1, with a table constraint checking predicate added to it. The table that is to be an argument for the predicate is the return value of the various lifting operators, just before the value is returned it will be used as an argument for the predicate defined for checking a table constraint. Depending on the result from the predicate the return value of the table method will be either a table or a message that inform that a constraint violation has occurred, in a similar manner to the object method. The algorithm of a table method produced by lifting the object method M to become a table method using a universal lifting operator A and that would use a predicate Φ to check table consistency is as follows:
A (M, t, tbl): $\mathbb{P}\tau$
Begin
$tbl := \{t \in tbl \bullet M(t)\}$
if $\Phi(tbl)$ **then**
tbl
else
$raise\ exception$ '$\Phi\ is\ violated$'
end if

End
Example on how table constraints checking predicate (Φ) generally look like: If we have a table tbl type set of objects t where t is type
$\langle id : int,\ pid : int,\ geom : geometry \rangle$, and have table constraint where it states the sum of all the area of the geom must be less that 1000km^2 then The pseudocode for the table IC predicate Φ would be:
$\Phi(t : \mathbb{P})$: boolean
Begin
$\sum$ area(T.geom) $>$ 1000
End

Database methods with constraint checking

When it comes to the constraint checking process this is also similar to the above object and table methods except that the predicate takes a database as a parameter and the database that is to be an input for the database constraint checking predicate is produced at run time. The database method is responsible for checking overall database consistency and committing the transaction. The predicate at this stage takes all the tables both those whose values are changed by the table methods and those left intact. The table and the object constraints been checked already by the object and the table methods therefore, here it only checks for database level consistency. The algorithm is as follows:
D(db:δ) : $text$

Begin

$db := db$ exceptset $tblmethod(tbl_i, \ldots, tblmethod(tbl_j)$
if $\psi(db)$ **then**
$updatetblset...$
return $message$ '$transaction\ committed\ successfully$'
else
$raiseexception$ '$database\ constraint\ \psi\ violated$'
end if

End

Chapter 3

Functional database design on PostgreSQL using PL/pgSQL

As stated in the introduction part of this paper, in this project our main emphasis is in implementing the functional design of databases in PostgreSQL database system only. We will be using PL/pgSQL, a loadable procedural language for PostgreSQL database systems as our programming language.

3.1 BASIC PL/PGSQL STATEMENTS FOR BUILDING THE METHODS

In this section we will describe the tools that Postgres has for designing functional database. We will also state some statement types that are explicitly understood by PL/pgSQL necessary for our design. Statements not explicitly understood by the PL/pgSQL are presumed to be an SQL command and thus are sent to the main database to execute. Most of the document below is summarized from [2]

3.1.1 Supported arguments and result data types

PL/pgSQL functions can take as an argument any scalar or array data type supported by the server, and they can return a result of any of these types as well. They can also take or return any composite type. Furthermore they can return record types and setof all the above mentioned types. However a setof some type cannot be an argument to PL/pgSQL functions.

3.1.2 Object Types (Row-Types)

Definition As stated in PosgreSQL server documentation, an object type represents the structure of a row; it is composed of a list of field values and their datatypes. A row variable or a row-type variable is also called a variable of a composite type. Defining a composite type is similar to creating table, except for the absence of constraints and the AS keyword. Composite types are used in PostgreSQL the same way as all other types are used. For instance they can be used to declare a column of a table. Moreover, it can be used to declare a variable, and this variable will be called a row-type variable and it can hold a row of a for loop or select query results. Here is an example on how to define a composite type:

```
CREATE TYPE parcel-owner AS (
  first-name  text,
   last-name   text )
```

Row-type Declaration There is no predefined datatype in PL/pgSQL for row-type variables, because each field has a unique name and datatype. Therefore, declaring a composite type can be done either by creating a composite type or using an already existing table name, since a composite type having the same name and row type as the table is created automatically when a table is created. Then a variable can be declared to be of that composite type using $composite\_type\_name$ or $composite\_type\_name\%ROWTYPE$ notations, but it is more portable when declaring with $\%ROWTYPE$. In a similar way a row variable can be declared to have the same type as table or view, using $table\_name\%ROWTYPE$ notation. As a result the fields of the row-type inherits the table's field size or precision and data types as well. Example

$name\ \ table\_name\%ROWTYPE;$

$name\ \ composite\_type\_name;$

Accessing a row-type In a row type variable the OID or other system columns cannot be accessed because the row could be from a view, therefore, only the user-defined columns of a table row are accessible. To access a field of a composite type (complete table row) the syntax is written using a dot and then the field name, for example the first_name field in name type is referenced as $name.first\_name$. However, to access a field in a column declared as composite type we need to parenthesize the table and the column name separated by a dot then the field can be written separated by another dot outside the parentheses otherwise, it can't be accessed. Since it is much like selecting a field from a table a parentheses is needed to save the parser from confusion. For example,

select name.last_name from employee where name.last_name = Josh

Does not work because the name $name$ is considered as a table while its just a column. It must be written as follows:

select (name).last_name from employee where (name).last_name = Josh

Executing a query with single row result The result from a single row returning function, can be assigned to a $target$ of type record variable, row-type variable, or list of scalar variables by writing the code and adding an INTO clause.

Select sqlcommand INTO [strict] target from ...;

This can be used for select, insert, update or delete functions. It is written the same way SQL command is written except for the INTO clause.

The into clause can be placed anywhere in the select statement. But mostly it appears just before FROM or just after or before the $sqlcommand$. If the query assigned to the record variable has zero rows, null values will be assigned to the variable, but if it has several rows only the first row will be assigned and the rest will be deleted, but the first row is not well defined unless ORDER BY is used. When we want to restrict the query result to be only one row we can write $into\ strict\ target$ this will give an error message if the query results many rows or no rows at all.

Functions on row-type Composite types can also be an argument to a function, in this case they are accessed by the identifier $n dot the field name, example $1.first_name. Furthermore, to select a field from a composite value returned from a function it should be specified as follows, the function in parentheses dot the field name.

```
select (func(...)).field from ...
```

Example on how to use row-types in functions. tbl1 and tbl2 are existing tables having at least the mentioned fields:

```
CREATE FUNCTION merge_fields(t_row tbl1) RETURNS text AS $$
DECLARE
 t2_row tbl2%ROWTYPE;
BEGIN
  SELECT * INTO t2_row FROM tbl2 WHERE ... ;
  RETURN t_row.f1 || t2_row.f3 ;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM tbl1 t WHERE ... ;
```

Comparison between row-types For two composite type instances to be equal their type name must be equal and their items' name and type must also be equal. It is coded using the $IS\ DISTINCT\ FROM$ function, it can be used with $NULL$ but if the value is empty it cannot be used, it generates syntax error. For example

```
SELECT ROW(1, 2) IS DISTINCT FROM ROW(3, 5); RETURNS TRUE
```

3.1.3 Assignment

Assignment of a value to a variable or row/record field is written as:

variable := expression;

To evaluate the expression statement, an SQL select command needs to be sent to the main database engine. It is important to remember that the expression should yield a single value and the returned value's datatype should match the variable's datatype, furthermore, the variable's specific size or precision, if it has, should also match. Otherwise the result value would be implicitly converted by the PL/pgSQL interpreter which can cause the input function to generate a run-time error. Example of an assignment:
color:= blue;

3.1.4 Set returning functions (SRF)

Set returning functions are functions that return a set of rows, by which the rows could be made of either scalar types or composite data types. They are also sometimes called table functions. These functions are mainly used in the From clause of a query, like a table, view or subquery. Example

*SELECT * FROM some SRfunc();*

They can also be called in the select list of a query, and for each row that the query generates by itself, the SRF produce a row for each element of the function's result set. However, its capability is deprecated that it might be removed in future releases[1].

They are mainly useful in functions returning composite types. If a function is defined to return a composite type, the SRF produces a column for each column of the composite type, and the resulting columns get the same names as the individual attributes of the composite type. But if the table function is defined for a scalar type, then one column table will be produced and the name of the column will be the name of the function.

Return, Return next and Return Query Table functions are achieved by declaring a function to return $setof sometype$, and each row returned by the function becomes an element of a set, set of rows if it is a composite SRF. In PL/pgSQL to create an SRF, a bit different procedure is followed from the normal functions. Here the individual items to return are appended to the result set using RETURN NEXT or RETURN QUERY commands, and the completion of the execution is specified by final RETURN command with no argument. Both RETURN NEXT and RETURN QUERY can be used in one function and their results will be concatenated.

In cases of working with dynamic queries the RETURN QUERY can be changed by RETURN QUERY EXECUTE, and the parameter expressions can be provided through USING into the computed query string, in a similar way to an Execute command See Section 3.1.4. The returned result sets are stored until the entire result is generated before they are returned, this causes poor performance. The result will be written to disc to reduce memory exhaustion, and this limitation is expected to be removed in the coming versions [2, 1]. Example to create a simple PL/pgSQL function that returns setof an existing table's row with some changes, you could use the following:

```
CREATE OR REPLACE FUNCTION parcels() RETURNS SETOF parcels AS  $$
DECLARE
 myrec  parcels%rowtype;
BEGIN
 FOR myrec IN SELECT id, pid+10, p.geom FROM  parcels LOOP
    RETURN NEXT myrec;
 END LOOP;
RETURN;
END;
$$LANGUAGE 'plpgsql';
```

Executing dynamic commands

There are cases that a command might need different tables or datatypes, every time it is executed. To cope with this problem we can use the *execute* statement.

```
EXECUTE command_string [ INTO [STRICT] target ] [ USING expression [, ...] ];
```

where *command_string* is an expression providing a string that holds the command to be executed, and it is of type text. *Target* is optional and is used for storing the result of the executed command. *Target* could be of the following types, a row variable, a record

variable, or a comma-separated list of simple variables and record/row fields. $USING$ is also an optional expression, it provides the values to be inserted into the command.

When using the $Execute$ statement for executing commands, the command is prepared every time the statement is run. Therefore it can be used to perform actions in different tables or columns within one function, since the command string is dynamically created. The result of the execution is assigned to the target using the $INTO$ command, but since select into is not supported inside $Execute$ the $INTO$ is specified as part of $Execute$ itself. When passing column or table names to a dynamic query the parameter symbols, i.e., $1, $2, etc, cannot be used, therefore they must be inserted textually. However, in cases of passing data values the parameter symbols can be used. Example,

```
EXECUTE 'SELECT * FROM'
|| table_name::regclass
|| 'WHERE inserted_by = $1 AND inserted <= $2'
INTO c
USING checked_user, checked_date;
```

Quoting values in dynamic queries

Often times when working with dynamic queries there are cases that we might want to escape single quotes. Such issues could be solved by using dollar quotes in cases of having fixed text in the function body although it is not always possible. Care needs to be taken when quoting values as they might already contain quote marks. For this reason, to insert expressions containing column or table names in a dynamic query, they should be passed through $quote\_ident$. While those expressions that have values that should be literal strings in the constructed command should be passed through $quote\_literal$ before insertion. The final out put will be the input text enclosed in double or single quotes as needed and any embedded special characters properly escaped. Using the function $quote\_literal$ when having null value causes the whole dynamic query to be nullified and cause an error message, therefore, in such cases its better to use $quote\_nullable$, that is the same as $quote\_literal$ except when null values are passed to it, it returns $NULL$.

3.2 CONSTRAINTS

Here constraints are predicates that will be placed in the if clause of a function. In general we will divide them into four groups, attribute, object, table, and database constraint. Attribute and object constraints will be placed in the if clause of the object method, while the table constraints will be placed in the table methods and the database constraints in the database method. Attribute constraints are defined in such a way they take an attribute and additional optional parameters and return boolean. In the same way object, table, and database constraint predicates will take object, table and database respectively and some optional parameters as an argument.

PostgreSQL has several tools to facilitate coding of those predicates, as a result diverse kinds of constraints can be coded. We will look at the operators available to construct the constraints.

3.2.1 Arithmetic operators

Constraints having arithmetic expressions can be handled using the built-in arithmetic operators in PostgreSQL. For such cases PostgreSQL, which has a large number of built-in operators, gives you the privilege of using all sorts of mathematical, spatial, and relational operators. The operators are summarized as follows, detailed information can be found in Postgres manual [2].

- –General Operators these are operators which are useful for comparison and description of the native data types, ranging from numeric types to data/time types. Less than, greater than, equal, concatenate string, not in which are indicated by the symbols $<, >, =, ||, !! =$ respectively are few examples of this class.
- –Numerical operators are operators useful for making mathematical computation on different numerical values. To mention some examples, factorial, modulo, multiplication, addition, subtraction, and absolute value which are symbolized by $!, \%, *, +, -, @, ||$.
- –Geometric operators are operators useful to make geometric analysis on spatial objects. Like is below, is above and same as, are few examples. Furthermore currently there are 681 built-in functions on Postgres for computing all sort of geometric calculations.
- –Time interval operators are operators which are used for making calculations with time datasets. Interval less than, Interval not equal, Interval equal, Interval greater than or equal to, Start of interval, Same as, and time inside interval which are symbolized as$\# <, \# <>, \# =, \# >=, |, =$ and $<? >$ are few examples of this class.

However, care must be taken when putting the operators in order as there is a hardcoded Lexical precedence in the parser. However most operators have the same precedence and are left associative. This may cause non-intuitive behavior; for instance the Boolean operators " $<=$ " and" $>=$ " have a different precedence than the Boolean operators " $<$ " and " $>$ ". Sample example for creating a predicate that use arithmetic operator, in table ensparcel, pid must be greater than id.

```
CREATE OR REPLACE FUNCTION Ar_pred(p ensparcels)
  RETURNS boolean AS
$BODY$
declare c boolean;
begin
  select  p.id >  p.pid into c;
  return c;
end;
$BODY$
  LANGUAGE 'plpgsql'
```

3.2.2 Aggregate operators

Aggregate functions are functions that provide a single output from several row inputs. Constraints having computations based on these aggregate functions are called aggregate constraints.

To code aggregate constraint checking functions PostgreSQL provides the following aggregate functions. These are count, sum, avg (average), max (maximum), min (minimum), sort ascending/ descending (order by ASC/DESC) etc. Furthermore in the spatial domain there are several operators such as St_polygonize, collect etc., they are covered in the section above under the geometric operators.

3.3 OBJECT METHODS

Object methods are functions that take an object and some optional parameters as an argument and return the same object with some or all of its values changed.
$M(t, e_i, \ldots, e_j) \overset{\text{has code body}}{=} t$ except $a_i = e_i, \ldots, a_j = e_j$
Object methods are created the same way normal functions are created in PostgreSQL, except its argument and return values are both of type row-type. It is similar to creating functions using row-type as explained in Section 3.1.2. Object methods are responsible for checking object constraint violations. They contain predicates that check constraint violation in their if clause within there code. They return a row with its values changed if the predicate returns true, otherwise it raises some error message and exits the transaction.

There are cases that we might want to call several object methods on a single row, and checking the object constraint every time, is both time consuming and error prone, since possible violations might be repaired by the proceeding object methods. Therefore, all the object methods except for the one which is to be lifted to become table method, should be coded without constraint checking predicates.
Example of creating an object method, $ensparcels$ is an existing table having

$\langle id : int, pid : int, geom : geometry \rangle$ fields:

```
CREATE OR REPLACE FUNCTION func1(p ensparcels)
  RETURNS ensparcels AS
$BODY$
declare pp ensparcels%rowtype;
begin
select p.id, p.pid+10, p.geom into pp;
if Ar_pred(pp) then
  return pp;
else
 raise exception 'object constraint Ar_pred violated';
end if;
end;
$BODY$
LANGUAGE 'plpgsql' VOLATILE
```

$func1$ takes p as an argument, p is a single row having a datatype of table $ensparcels\%rowtype$. $func1$ returns p with some or all of its values changed if the predicate $Ar\_pred(pp)$ return true, i.e., if there is no constraint violation. However, if there is a violation it will send the error message 'object constraint Ar_pred violated'. This predicate $Ar\_pred(e\ ensparcels)$

is coded in Section 3.2.1. To look at their return value, object methods are called as either of the two ways, it is discussed in Section 3.1.2.

$$SELECT(func1(ensparcels.*)).*FROM\ ensparcels\ where\ id = 20300$$

$$SELECT(func1(p.*)).*FROM\ ensparcels\ p\ where\ id = 20300$$

id	pid	geom
20300	31788	010600020407

```
CREATE OR REPLACE FUNCTION func2(p ensparcels)
  RETURNS ensparcels AS
$BODY$
declare pp ensparcels%rowtype;
begin
select p.id, p.pid+7, p.geom into pp;
return pp
end;
$BODY$
LANGUAGE 'plpgsql' VOLATILE
```

Func2 does not have constraint checking predicate in its code, however, the constraint checking could be handled by calling func2 inside func1, i.e., func1(func2(p ensparcels)).

3.4 TABLE METHODS

Table methods are created in two ways by lifting object methods using lifting operators or by directly coding them. They can do nothing on the database except to return setof rows, which will be used in the database method at the next stage. They also check the table constraint before returning, we will see how the checking is done in section 3.4.4 they follow similar steps, therefore, can be applied to all in a similar manner for simplicity we will begin with the three lifting operators without table constraint check:

3.4.1 Singular lifting operator S

In theory this operator takes an object method M, an object t, and a table tbl where $t \in tbl$ and some additional parameters $e_i, \ldots, e_j$ as argument and it maps the object method M on t and returns the table tbl intact except for the value of t changed by $M(t)$.

$$S(M, t, tbl, e_i, \ldots, e_j) \overset{\text{has code body}}{=} tbl \text{ exceptset } \{M(t,\ e_i,\ \ldots, e_j)\},$$

where in most cases $M(t, e_i, \ldots, e_j) \overset{\text{has code body}}{=} t$ except $a_i = e_i, \ldots, a_j = e_j$. M can be inserted as an argument which is declared with the same datatype as its return value, and since it return the same datatype as the object t then the function will be as follows:

S (t_m : rowtype, t : rowtype, tbl : setof rowtype)

As discussed in Section 3.1.4, in PL/pgSQL a set returning function SRF should be considered as table and therefore, is placed in the from clause 3.1.4, and to see its return value it can be called as

SELECT FROM tablemethod,

thus it cannot take a from clause for its arguments since it is in the from clause by itself, therefore the object method with its from and where clause should be placed as an argument in text form. Furthermore the object t needs a from clause as well, and need to be treated the same way as the object method is treated, i.e., putting it in text form. The last argument is the table tbl which is of type setof row-type. PL/pgSQL capability to support setof type as an argument for a function is depreciated, see Section 3.1.4, therefore tbl will too need to be input as text. To illustrate the code for Singular lifting operator we will look at the following example,

```
CREATE OR REPLACE FUNCTION funcs2(M_arg text, t_arg text, tbl text)
  RETURNS SETOF enscadeparcels AS
$BODY$
declare
pp  enscadeparcels%rowtype;
ob  enscadeparcels%rowtype;
tt  enscadeparcels%rowtype;
sqlcommand text;
begin
execute M_arg into ob;
execute t_arg into tt;
sqlcommand := 'select * from ' || quote_ident(tbl);
for pp in execute sqlcommand
loop
  if pp.id is distinct from tt.id
  then return next pp ;
  else return next ob ;
    end if;
end loop ;
return ;
end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
```

This function loops over all the rows of the table tbl and stores the row using return next in the disc until it finds the row, which is equal to t, then it store m(t) and continues with the rest of the rows when it reach at the end of the row then the value will be returned using the return command. Therefore, we have the whole table with a single row changed by m(t). The return value of the function can be seen using the following syntax

$$SELECT * FROM\ funcS('M\_arg', 't\_arg', 'tbl')$$

Where $'M\_arg'$ is the object method with all its $from$ and $where$ clause in text form, and t_arg is an object with its from and where clause in text form, produced by a simple select

function that return one row from a table in text form and tbl is the name of the table in text.

$funcS2$ returns the whole table after it substitutes one row by the return value of the object method $obm1$ and returns the remaining rows as they initially were. For example the function can be called as follows to see its return value,

```
SELECT * from funcS2($$select (obm1(p.*, line2)).* from
    enscadeparcels p, (select 'srid = 28992; linestring
    (254922.64892 474499.963925, 255324.4419 473012.9989)'
    ::geometry as line2) as b where id=52372$$, 'select *
    from enscadeparcels where id = 52372', 'enscadeparcels')
order by id
```

id	pid	geom
52371	21733	"0106000020407100000100
52372	48110	"010300002040710000A00
52373	34708	01060000204071000010000
52373	14775	01060000204071000010000

3.4.2 Universal lifting operator A

A maps a single object method over the whole set of rows. It takes an object method M, an object t and a table tbl where we assume $t \in tbl$ as an argument, and it returns the table tbl with all its objects t changed by t' by mapping M to all $t \in tbl$.

$$\mathrm{A}(M, t, tbl) \stackrel{\text{def}}{=} \{t \in tbl \bullet M(t)\}$$

To implement this in PostgreSQL the type of the arguments of A has to be changed to text, due to the limitations of SRF as discussed in Section 3.4.1 and Section 3.1.4. Therefore, the trick we used here is to insert just the name of the object method M in text and its arguments will be built inside the function using an execute clause.

```
CREATE OR REPLACE FUNCTION funcA4(M_n text, tblname text)
  RETURNS SETOF enscadeparcels AS
$BODY$
declare
pp  enscadeparcels%rowtype;
ob  enscadeparcels%rowtype;
sqlcommandt text;
sqlcommandp text;
q text;
begin
sqlcommandt := 'select * from ' || quote_ident(tblname);
for pp in execute sqlcommandt loop
sqlcommandp:= 'select * from ' ||M_n|| '('||pp||'.*)';
execute sqlcommandp into ob;
```

```
    return next ob;
end loop ;
return ;
end;

$BODY$
  LANGUAGE 'plpgsql' VOLATILE
```

select * from funcsA4('func1' , 'enscadeparcels') Here the function loop over the rows of the table $tblname$ and maps the object method M on each row. M is developed in the execute statement by concatenating the name of M with the brackets. Then it stores the the $m(t)$ where $t \in tblname$ in the disc using the return next command until it reaches the end of the row. Finally the setof rows can be returned using the return command. To see its return value it can be called as follows:

$$select * from funcsa4('func1', 'enscadeparcels')$$

3.4.3 Filtered lifting operator F

It maps a single object method over some set of rows selected from the whole table. It takes an object method M, setof objects T and a table tbl where we assume $T \in tbl$ as an argument, and it returns the table *tbl* with all its set of objects T changed by T' by mapping M to all the elements of $T \in tbl$.

$$F(M, T, tbl, e_i, \ldots, e_j) \overset{\text{has code body}}{=} tbl \text{ exceptset } \{T \in tbl \wedge t \in T \bullet M(t, e_i, \ldots, e_j)\},$$

To implement this in PostgreSQL the argument type of the function has to be changed to text, due to the same limitations occurred in the other lifting operators above, that a set of objects cannot be an argument and an SRF is considered as table, therefore the object method with its from and where clause cannot be parametrized. Here T could be expressed as a criteria W imposed on t to be an element of the subgroup T. This group has to pass through the object method while the rest of the table will be returned unchanged. Therefore, this predicate can be placed in the if clause of the function body and decides which one goes where.

The same trick as the universal lifting operator can be used here i.e., to insert just the name of the object method M and the criteria as a function W in text and its arguments will be built inside the function using an execute clause and the table name as text as well.

```
CREATE OR REPLACE FUNCTION funcf2(p text, w text, tblname text)
  RETURNS SETOF enscadeparcels AS
$BODY$
declare
pp  enscadeparcels%rowtype;
ob  enscadeparcels%rowtype;
sqlcommandt text;
sqlcommandp text;
sqlcommandw text;
```

```
q text;
wt text;
begin
sqlcommandt := 'select * from ' || tblname ;
for pp in execute sqlcommandt loop
 q:=  pp;
sqlcommandw:= 'select * from ' ||w||'('||q||') WHERE id = q.id' ;
if execute sqlcommandw then--boolean
then
sqlcommandp:= 'select * from ' ||p||'('||q||') WHERE id = q.id' ;
execute sqlcommandp into ob;
    return next ob;
else
    return next pp;
end if;
end loop ;
return;
end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
```

3.4.4 Table method directly coded

The last option to create a table method is by directly coding it. Here the object methods are not passed as an argument but will be called fron inside the code. The arguments are a table tbl, a row t and a linestring $line$ all in text. This function is adopted from the next chapter from the case study it will be discussed in detail there but, we will use it for illustration. Here we will also look at how table constraints are implemented

The table method $tblm$ loops over the rows of the table tbl and stores the row in a temporary table tb created inside the the function, and when it finds rows meeting some criteria they will pass through the object method and be inserted in tb and the loop continues until it reaches the last row. A table constraint will use the dataset stored in the temporary table tb to check for any table constraint violation. In this function the constraint was the id and pid of the new row to be inserted has to be distinct. Therefore, the row to be inserted was created as follows using the values of tb

$$max(tb.pid) + 1, (obm2(cc, line1)).geom\ from\ tb;$$

The same technique could be used in all the table methods discussed above, those created using the lifting operators could do the same by inserting the return value in a temporary table and access it later. The code is as follows:

```
CREATE OR REPLACE FUNCTION tblm(tbl text, t text, line text)
  RETURNS SETOF enscadeparcels AS
$BODY$
declare
pp enscadeparcels%rowtype;
tt enscadeparcels%rowtype;
```

```
cc enscadeparcels%rowtype;
sqlcommand text;
line1 geometry;
begin
drop table if exists tb;
create temporary table tb(id int, pid int, geom geometry);
sqlcommand := 'select * from ' || tbl;
execute line into line1;
execute t into tt;
for pp in execute sqlcommand
loop
 if pp.id is distinct from tt.id then
   insert into tb values (pp.*);
 elseif  pp.id is not distinct from tt.id then
   select pp.* into cc;
   insert into tb values ((obm1(cc, line1)).*);
 else
   return next pp ;
 end if;
end loop ;
insert into tb (id, pid, geom) select max(tb.id)+1,
   max(tb.pid)+1, (obm2(cc, line1)).geom  from tb;
return query select * from tb;
end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
```

3.4.5 The database method D

This is the only externally public function, it will be responsible for checking database constraints and making the transaction. It makes use of the table methods by calling them inside its code. It will take the tables that are to be changed and additional parameters that are needed for the table method's argument. It checks the database constraints using the returned dataset from the table methods and the tables that are to be left intact, but have relation to the other tables, depending on the required argument for the constraints they will be called. The following database method *dbm* is also adopted from the case study made in the next chapter. This function takes two table names *tbl_bu* and *tbl_pa*, an object and additional parameter linestring. Three of the arguments are needed by the table method called inside the function. Here the predicate cross checks between the two tables and if there is no violation, then the transaction will be committed.

It loops over table *tbl_bu* rows and checks against the table that is returned from the table method, for constraint checking. If no violation occurs then the transaction will be committed. A save point is set inside in case there exist a constraint violation so that it can role-back to the save point and exit the transaction.

```
CREATE OR REPLACE FUNCTION dbm(tbl_bu text, tbl_pa text, t text, line text)
  RETURNS text AS
$BODY$
declare
pp enscadeparcels%rowtype;
tt enscadeparcels%rowtype;
zz enscadeparcels%rowtype;
```

```
sqlcommand text;
sqlcommand2 text;
begin
savepoint s1;
sqlcommand := 'select * from ' || tbl_bu;
sqlcommand2 := 'select * from tblm(tbl_pa, t, line ) where id = '|| cast(tt.id as text);
execute t into tt;
for pp in execute sqlcommand
loop
execute sqlcommand2 into zz ;
 if st_intersect (pp.geom, zz.geom)
      --zz.geom is the first half of the geometry of the parcel that is divided into two
 then
    execute 'delete from tbl_pa where id='|| cast(tt.id as text);
    execute 'insert into tbl_pa values (zz.*)';
    execute 'insert into tbl_pa values  tblm(tbl_pa, t, line ) where id = (select max(id)
      from tblm(tbl_pa, t, line))';
    execute 'update tbl_bu set pid = (select max(id) from tblm(tbl_pa, t, line))as d';
     -- updates the value of building pid by the pid of  the table returned from the table
     --method where id is max
 elseif (st_intersect (pp.geom, zz.geom)) = false then
 rollback to s1;
   raise exception 'building in more than one parcel found';
 end if;
end loop;
return 'transaction completed successfully';
end;
$BODY$
Language plpgsql
```

Chapter 4

Sample case study

4.1 INTRODUCTION

As it has already been mentioned our objective is to create a solid method of database design that allows to make an updates on databases by taking into account both the expressible and inexpressible constraints in the UML diagram Figure 4.1. The general philosophy we are going to follow is as discussed in the previous two chapters, to develop a solid database method that are constructed from solid, robust table methods where the tables are made from solid and robust object methods.

In order to illustrate the use of our methodology we now introduce a simple spatial database having just two tables, $enscadebuildings$ and $enscadeparcels$. The database describes how the buildings are associated with the parcels for documenting it in the city administration. Entities in the database include buildings and parcels and their geometry. Their relationship is described as follows

- –There is primary key in both tables as a table cannot be created without it. Attribute name $enscadebuildings.id$ and $enscadeparcels.id$ are the primary keys of the two tables.
- –The buildings are located inside the parcels, and the parcels have distinct id numbers. The buildings in the $enscadebuildings$ table have associated parcel id values from $enscadeparcels$ table. Therefore, the two tables are referentially integrated on $enscadebuildings.pid$ and $enscadeparcels.pid$, i.e., for all pid in $enscadebuildings$ table there exists pid in $enscadeparcels$ table.

  $$\forall b \in enscadebuildings \; \exists p \in parcels :$$

  $$b.pid = p.pid$$

- –Furthermore, the two tables can be spatially interrelated by their geometry, however, we cannot express it by the existing technology of constraints writing in the database. To do this we need a check constraint that can work between two tables, i.e., we need a database check constraint which will enforce the geometry of the building to be within the geometry of the parcel.
- –There are also two tables where the spatial properties of the geometries are registered and they are referentially related to the $enscadebuildings$ and $enscadeparcels$ tables.

We also constrain the database with constraints which are not expressible in UML class diagram Figure 4.1. For the sake of illustration we have put it in the stars in the diagram. We would like the system to enforce the following constraints in the database also. These are

1. A building should be only in a single parcel spatially.

   $$\forall b \in enscadebuildings \; \forall p \in enscadeparcels$$

   $$b.pid = p.pid \Rightarrow within(b.geom, p.geom)$$

   This implies that we have to be careful when dividing a parcel in to pieces that the buildings should remain in a single parcel. For example if we are cutting a parcel by passing a line

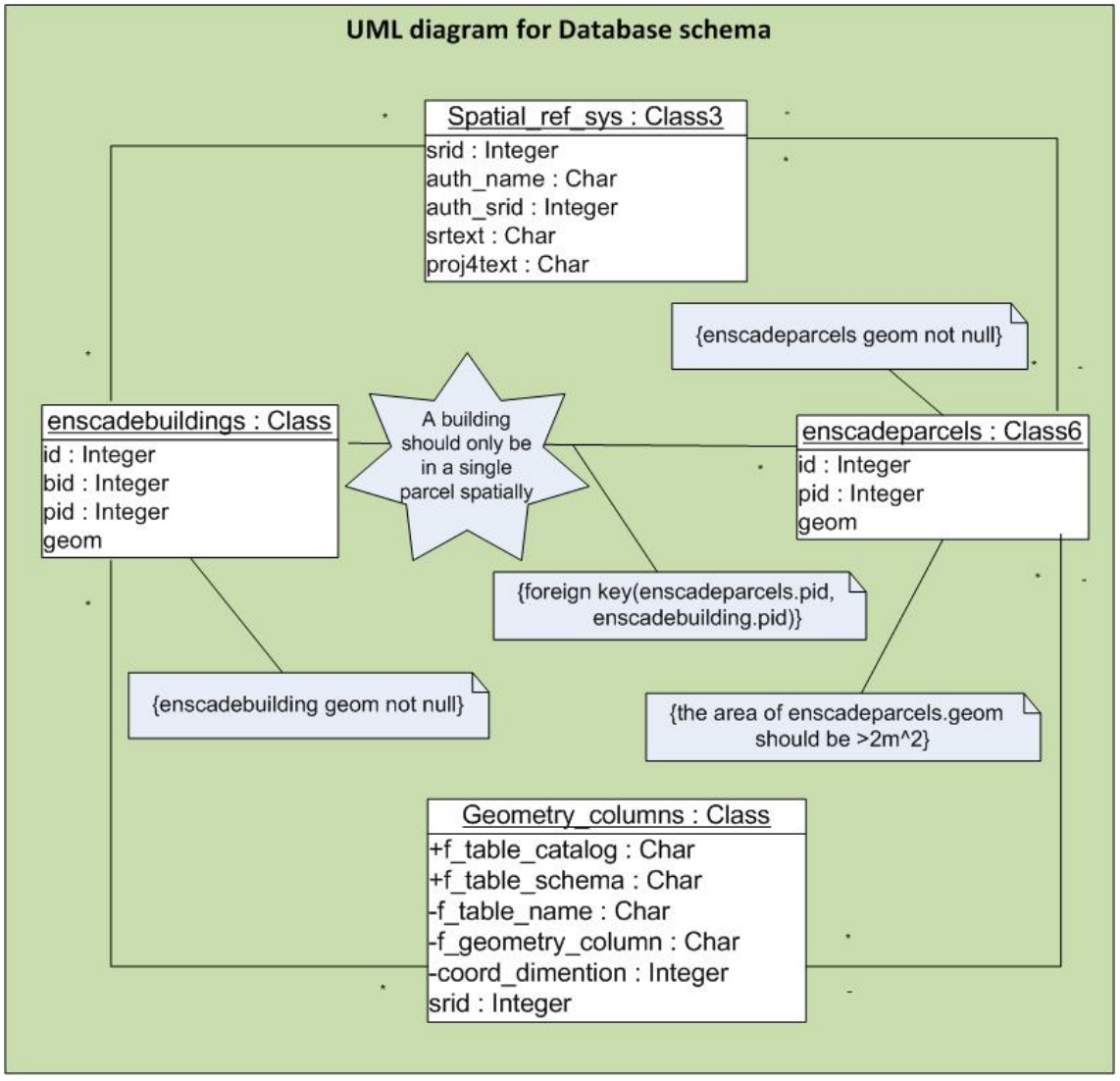

Figure 4.1: UML diagram for the administration database

through it then, a constraint that states the line should not pass through the buildings should be enforced.

2. The area of a parcel cannot be less than $2m^2$.

$$\forall g \in parcels :$$

$$St\_area(g.geom) < 2m^2$$

3. We also have a dynamic constraint that says that we can cut a parcel in to two only at a time.

4.2 SAMPLE TRANSACTION

Here we will look at a sample transaction: The city administration wants to divide a parcel in $enscadeparcels$ table where id=52372 in to two and register them as two separate parcels. The steps to follow are, initially cut the parcel in to two by passing a linestring through it, and make sure all the integrity constraints are maintained. Then update the parcel geometry by the new parcel segment, and insert the remaining half as a new record. Finally edit the buildings' pid value accordingly.

- The PL/pgSQL code for splitting the parcel in to two by a linestring without considering any constraints is as follows. split_geom(geometry, geometry) takes the parcel geometry and a line string by which the parcel will be subdivided as an argument.

```
CREATE TYPE holder1 AS (geom geometry, line geometry);

CREATE OR REPLACE FUNCTION split_geom(geometry, geometry)
  RETURNS holder1 AS $$
select st_intersection(st_buildarea(st_setsrid(st_linemerge(st_collect(
 split1_geom, astext($2))), 28992)),$1) as parcel1,
      st_intersection(st_buildarea(st_setsrid(st_linemerge(st_collect
      (split2_geom, astext($2))), 28992)),$1)as parcel2
from
(
  select astext(st_line_substring(loc.ex_geom, location1, location2))AS split1_geom,
         astext(st_linemerge(st_collect(st_line_substring(loc.ex_geom, location2,1),
        (st_line_substring(loc.ex_geom,0, location1))))) AS split2_geom
  from
  ((
   SELECT ST_Line_Locate_Point(ex_geom, pt.point0) AS location1,
    ST_Line_Locate_Point(ex_geom, pt.point1) as location2, ex_geom
   FROM
   (((
        select astext(ST_ExteriorRing(ST_GeometryN($1,1))) as ex_geom) as ex,
        (select astext( st_geometryN(st_intersection(st_exteriorring(
        astext(st_geometryN($1,1))), astext($2)),1)) as point0,
       astext(st_geometryN(st_intersection(st_exteriorring(
       astext(st_geometryN($1,1))), astext($2)),2)) as point1
  )))  as pt
)) as loc -- gives the location of the intersection points of the line
--  and the boundary of the parcle

) as e -- gives the two split sides of the exterior of the geom
$BODY$
  LANGUAGE 'SQL' IMMUTABLE STRICT
```

The two geometries created by using the above function can be called as follows and the resulting parcels are shown below.

```
select (split_geom(p.geom, line1)).*
From enscadeparcels as p, (select 'srid= 28992; linestring(
254922.64892 474499.963925, 255324.4419 473012.9989)' ::
geometry as line1) as b
Where id=52372
```

parcel1	parcel2
"0103000020407100000A0	"0103000204071000021

The above function $split\_geom(geometry,\ geometry)$ will be called to divide the parcel into two if the following criteria are all met. The criteria are coded as a predicate returning boolean for checking constraint violations. The criteria to be meet are listed as follows:

1. The line should only cut the parcel in to two, The predicate checks whether the linestring touches or intersects, the external boundary of the parcel only twice.

```
CREATE OR REPLACE FUNCTION inters2(geometry, geometry)
  RETURNS boolean AS
$BODY$
Select st_npoints(st_intersection(st_exteriorring(astext
(st_geometryN($1,1))), astext($2)))= 2
$BODY$
 LANGUAGE 'sql' IMMUTABLE STRICT
```

2. The resulting parcels after the splitting take place should have areas greater than $2m^2$, the predicate checks if the areas of the two sub parcels are greater than $2m^2$. This is an attribute constraint and will be checked inside the object method.

$$St\_area(subdividedparcel.geom) > 2m^2.$$

3. The linestring must not cut the geometry of the buildings, i.e., to ensure buildings remain in a single parcel after the parcel is divided. The predicate below checks whether the linestring intersects with any of the buildings that exist inside the parcel. This is a database constraint and will be checked inside the object method

$$St\_intersects(building.geom,\ linestring)$$

–Then one half will update the existing geometry while the remaining half will be inserted as a new record. When inserting the remaining parcel segment in to the $enscadeparcels$ table the parcel should be given a distinct id and pid values where id is the table's primary key while pid is a distinct representative value for the parcels in the table. Both are table constraints and will be handled by the table method in a similar way to the following.

```
INSERT  INTO enscadeparcels values
 (max(p.id)+1, max(p.pid)+1, split_geom(p.geom, line).parcel2)
FROM enscadeparcels as p where p.id=52372
```

–Finally move the buildings to the parcels they belong to by editing the *pid* of the buildings accordingly. This is done by the database method because it involves two tables. It select all the buildings that are within the second half of the parcel, i.e., parcel2 and update their pid value to the pid value of parcel2.

```
UPDATE enscadebuildings
SET  b.pid = p.pid
FROM enscadebuildings as b, enscadeparcels as p
WHERE st_within(b.geom, split_geom(p.geom, line).parcel2)
```

4.3 DEVELOPING THE DATABASE METHOD

To get the desired transaction our methodology is a 3 step process, that involve producing object, table, and database methods.

4.3.1 Object method

As it has been discussed in the previous chapters, object method is a function that takes an object and additional parameters as an argument and it return the object with a changed value. Furthermore all the object methods discussed above will be checked here. In our case two object methods are needed *obm*1 and *obm*2 one is for updating the value of the row while the other is for preparing the row to be inserted.

Both of the object methods take the whole row of which the parcel that we want to split belongs to, and the geometry of the linestring with which the parcel's geometry will be split as an argument. The return value of *obm*1 is an object with the parcel geometry changed by one half of the previous geometry. While the *obm*2 returns the same object as the *obm*1 except the geometry is the other half. For this row to be inserted in the database as a new entry it's *id* and *pid* values should be distinct from all the list of values in the table as discussed above. At this granularity level it is not possible to get distinct value since object methods only take a single row as an argument, therefore, table constraints cannot be checked. As a result the distinct values for the *id* and *pid* will be dealt in the table method.

•Creating *obm*1

```
CREATE OR REPLACE FUNCTION obm1(p enscadeparcels, line geometry)
  RETURNS enscadeparcels AS
$BODY$
declare pp   enscadeparcels%rowtype;
Begin
select p.id, p.pid, (split_geom(p.geom, line)).parcel1 into strict pp;
IF  inters2(p.geom, line)= false then  -- constraint 1
Raise Exception 'the line should touch the parcel geom only twice';
ELSEIF st_area((split_geom(p.geom, line)).parcel1)<2 then
Raise Exception 'area of new geom cannot be less than 2';
else
  return pp;
END IF;
End
$BODY$  LANGUAGE 'plpgsql' VOLATILE
```

The object method obm1 when called as follows check the object constraints placed in the if clause, and returns the object with it geom value changed as shown below, if no constraint violation is found.

```
select (obm1(p.*, line2)).*
From enscadeparcels p, (select 'srid= 28992; linestring (254922.64892
474499.963925, 255324.4419 473012.9989)' :: geometry as line2) as b
Where id=52372
```

id	pid	geom
52372	48110	"0103000020407100000A0

However if there is constraint violation, either of the following error message will be returned.

ERROR: the line should touch the parcel geom only twice, or

ERROR: area of new geom cannot be less than 2

- The second object method $obm2$

```
CREATE OR REPLACE FUNCTION obm2(p enscadeparcels, line geometry)
  RETURNS enscadeparcels AS
$BODY$
declare pp   enscadeparcels%rowtype;
Begin
select p.id, p.pid, (split_geom(p.geom, line)).parcel2 into strict pp;
IF  inters2(p.geom, line)= false then
Raise Exception 'line must touch the geom twice';
ELSEIF st_area((split_geom(p.geom, line)).parcel2)<2 then
Raise Exception 'area of new geom cannot be less than 2';
else
  return pp;
END IF;
End
$BODY$  LANGUAGE 'plpgsql' VOLATILE
```

It return the same result as the object method $obm1$ except for the value of $geom$ changed by the other half of the subdivided parcel. Its input value is the same as the input for the $obm1$.

id	pid	geom
52372	48110	"0103000020407100002100

4.3.2 Table method

After creating the object method, we need a table method that does the following

$$tblm(T : I\!Penscadeparcels, p : enscadeparcels, l : geom)$$

$$== (Texceptsetobm1(p, l))addsetobm2(p, l)$$

An appropriate implementation of this abstract code should ensure that the second use of p (in obm2) is not reading the p updated in the first part of the code. This means our code will do something like this:

q1 := obm1(p,l)
q2 := select max(p.id),max(p.pid), obm2(p,l) from enscadeparcels as p
T := T exceptset q1
T := T addset q2
return T
For this case it is better to develop the table method by directly coding it, i.e., by calling the object methods inside the codebody, instead of using the lifting operators. The following table method $tblm$ takes as an input table name tbl as text, an object t as text and a line geometry as text. All has to be passed as text because the function is a Set returning function and by itself it will be in the from clause that its arguments cannot have additional from clause see Section 3.1.4 and Section 3.4.

$tblm$ creates a temporary table tb to store the return value of a loop statement that loops over the rows of the input table tbl. Then when the row equal to t is found that row will be updated by the return value from the object method $obm1(p\ enscadeparcels, line\ geom)$, that is the row with half the parcel geometry. At the end the second half of the geom is inserted into the temporary table tb as a new row with the following values (max(tb.id)+1, max(tb.pid)+1, (obm2(cc, line1)).geom from tb). This provides the distinct values for id and pid .

```
CREATE OR REPLACE FUNCTION tblm(tbl text, t text, line text)
  RETURNS SETOF enscadeparcels AS
$BODY$
declare
pp enscadeparcels%rowtype;
tt enscadeparcels%rowtype;
cc enscadeparcels%rowtype;
sqlcommand text;
line1 geometry;
begin
drop table if exists tb;
create temporary table tb(id int, pid int, geom geometry);
sqlcommand := 'select * from ' || tbl;
execute line into line1;
execute t into tt;
for pp in execute sqlcommand
loop
 if pp.id is distinct from tt.id then
   insert into tb values (pp.*);
 elseif  pp.id is not distinct from tt.id then
   select pp.* into cc;
   insert into tb values ((obm1(cc, line1)).*);
 else
   return next pp ;
 end if;
end loop ;
insert into tb (id, pid, geom) select max(tb.id)+1,
   max(tb.pid)+1, (obm2(cc, line1)).geom  from tb;
return query select * from tb;
end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
```

The table method *tblm* returns the whole table with one row updated and another row inserted when given the following input, the two rows with new value are shown below.

```
select * from tblm2('enscadeparcels', 'select * from enscadeparcels
where id= 52372', $$select 'srid = 28992; linestring
(254922.64892 474499.963925, 255324.4419 473012.9989)'
   ::geometry$$) where order by id
```

id	pid	geom
52372	48110	"0103000020407100000A0
52771	52771	"01030000020407100002100

4.3.3 database method

Database method is the final stage where all the database constraint will be checked and the transactions will be committed. We have one database constraint that involves both the two tables. Constraint number 3, from the list above, it checks whether the buildings remain in a single parcel i.e., check whether the line used to cut the parcel also cut any building. We will check whether the subdivided parcels cut the any of the building's geometry.

$$St\_intersects\ (enscadebuildings.geom\ enscadeparcels.geom)$$

If this constraint is maintained the next step is to delete the row and insert the two new rows from the temporary table created in the table method. Then move the buildings to the parcels they belong to by editing the *pid* of the buildings accordingly. It selects all the buildings that are within the second half of the parcel, and update their pid value accordingly. If in the middle constraint violation is found it roll back to its save point and raise an error message ERROR building in more than one parcel found

If the transaction is committed successfully a text message is returned

'transaction completed successfully'

It is coded as follows:

```
CREATE OR REPLACE FUNCTION dbm(tbl_bu text, tbl_pa text, t text, line text)
  RETURNS text AS
$BODY$
declare
pp enscadeparcels%rowtype;
tt enscadeparcels%rowtype;
zz enscadeparcels%rowtype;
sqlcommand text;
sqlcommand2 text;
begin
savepoint s1;
sqlcommand := 'select * from ' || tbl_bu;
sqlcommand2 := 'select * from tblm(tbl_pa, t, line ) where id = '|| cast(tt.id as text);
execute t into tt;
for pp in execute sqlcommand
```

```
loop
execute sqlcommand2 into zz ;
 if st_intersect (pp.geom, zz.geom)
      --zz.geom is the first half of the geometry of the parcel that is divided into two
 then
    execute 'delete from tbl_pa where id='|| cast(tt.id as text);
    execute 'insert into tbl_pa values (zz.*)';
    execute 'insert into tbl_pa values  tblm(tbl_pa, t, line ) where id = (select max(id)
      from tblm(tbl_pa, t, line))';
    execute 'update tbl_bu set pid = (select max(id) from tblm(tbl_pa, t, line))as d';
     -- updates the value of building pid by the pid of  the table returned from the table
     --method where id is max
 elseif (st_intersect (pp.geom, zz.geom)) = false then
 rollback to s1;
   raise exception 'building in more than one parcel found';
 end if;
end loop;
return 'transaction completed successfully';
end;
$BODY$
Language plpgsql
```

The database method is called as follows,

```
select * from dbm('enscadebuildings' , 'enscadeparcels', 'select * from enscadeparcels
where id= 52373', $$select 'srid = 28992; linestring
   (254922.64892 474499.963925, 255324.4419 473012.9989)'::geometry$$)
```

Chapter 5

Discussion, Conclusions and Recommendations

In this chapter, the results acquired, and the major dealings of this MSc thesis project relating it with the research questions stated in Section 1.2.2 are outlined and discussed. Followed by a conclusion part that contains the summing up of the points discussed and the findings. Finally, we will put forward some recommendations for future work to invite other researchers for the development of the science.

5.1 DISCUSSION

The main objective of this research is to formulate a methodology that allows encapsulating a database in suite of functions that is designed in such a way that they preserve integrity constraints, and allow users to access and update the database through those functions only. To meet this objective the following research questions were forwarded and were dealt as follows.

1. How are attribute, object, table and database constraints implemented in PostgreSQL?

   Based on data structure granularity, constraints are divided into attribute, object, table and database constraints. In PostgreSQL constraints are specified either in the schema and are checked automatically or they are expressed in the application program using a constraint language, and are checked during the creation of the database upon data entry or during transactions, i.e., when the database changes from one state to the next [6]. Depending on this at an abstract level constraints are classified into: i) inherent constraints ii) implicit constraints and iii) explicit constraints.

2. Is there a higher level language to conceptually design functions?

   There was not an existing language, we had to create that from scratch with the help of my supervisor, who provided the formal language basis to start from for making the conceptual design. The formal semantics of the expression forms proposes is defined in such a way that it allows us to express the intuition “take the whole dataset, make the indicated local change, and return the dataset thus obtained”. The classical set-up of type theory was followed and a number of expressions that are needed to define schema transformation expressions were listed. They provide the fundamental ‘building blocks’ to construct larger, end-user-oriented expressions.

   The methods were designed conceptually following the hierarchy of data structure granularity, from object to table then to database methods. Initially their class universes were defined, to indicate the criteria an attribute values has to meet to be a member of that class by considering all the constraints imposed in the database. Then to create a solid and robust methodology that leads to getting a final one database method that will check the overall consistency and perform all the transaction was formulated.

   The logic was based on building a solid and robust database method from solid and robust object and table methods. The object methods are designed in such a way they return the row intact except for some values changed, if the constraints will still be preserved even after the changes are made. If constraints are violated then it will exit the transaction and send an error message. The table methods are designed in a similar way, they return the whole

table intact except for some rows changed, after checking whether the table constraints will still be preserved even if the changes are made. Table method designing could be done either directly, or by lifting object methods to become table methods. The lifting operators we defined for addressing the various cases are singular, universal and filtered lifting operators. Directly coding the table methods also use the object method in a different approach, i.e., by calling them within the function. Database methods use all the tables to check for constraint preservation, both the tables returned from the table methods and the once untouched, and finally it commits the transaction.

3. What are the methods to develop robust object, table and database methods?

   Our methodology provides a functional design to build those methods by developing the methods following the hierarchy ladder based on the conceptual design answered in the first question. We will look at each method one by one.

   – Object methods are built as normal PostgreSQL functions except that it take an object as an argument and return an object, M($t : rowtype$) : $rowtype$. They are coded in such a way they take an object and return the object intact except for some attribute values changed. These methods are responsible for both attribute and object constraints checking. It is for simplicity case that the attribute changes and attribute constraints checking are done under the object method. Object methods are later either lifted to become table methods or are called in the table method. They have no impact on the database by themselves. All object methods do not have constraint checking predicates as it will cause repetition of work if several object methods are called in one row. Therefore only the last object method that will be called on the row, i.e., the one which is to be lifted, will have all the constrain checking predicates. If we have two object methods M_1 andM_2 then M_1 can be called inside M_2, $M_2(M_1())$ since the argument and the return values are both the same $rowtype$. Therefore M_2 could serve as a constraint checking method.

   – Table methods are built in two ways, either by lifting the object methods, or by directly coding them. For lifting object methods to table methods three lifting operators were discussed. They are summarized as follows:

     Singular lifting operators , these are used in times when we want to apply an object method on a single row of a table. These operators take a table, an object method and an object as an argument and it maps the object method on the object and return the whole table intact except for the value of the row changed. To implement this in PostgreSQL using PL/pgSQL some tricks had to be made for the following reasons. PL/pgSQL functions capacity to take setof some type as an argument is depreciated, therefore the table can only be passed as text and later be formulated and executed using the execute statement. The same applies to the object method, functions can not be passed where there argument is to be determined inside the function, therefore we input the object method along with its arguments as text as an argument. Finally the object where the object method is to be mapped is also passed as a text, for similar reason that the from clause that the object needs can not be provided there. After putting all the arguments in text form they will be built inside the function in an execute statement to provide the result, an example is provided in the document.
     Universal lifting operators , are useful to map a single object method over all objects of a table. The concept is it takes an object method and a table as an argument and it returns the table after mapping the object method on all rows of the table. It was implemented in PL/pgSQL using similar trick as in the singular lifting operator for similar reasons. The table is passed as text because it is a setof row type, and the object method is passed by only putting its name without its argument and bracket as text. Within the code there is a loop that provides the input rows one by one for the object method. And the object method is constructed with in an execute statement. The code we provided is not currently working, but similar other simple functions that take the

name of the a function and concatenate its bracket inside an execute statement works well so this has to work also but due to time constraint we are not able to show that in practice.
Filtered lifting operator , can be used in times when we want to map an object method only to selected rows of a table. The selection is made based on some criteria, which we can build as a function, then those rows meeting that criteria will pass through the object method while the rest of the rows of the table will be returned intact. In PL/pgSQL this are coded as a function that take a table, an object method, and a predicate that contain the criteria for filtering. In a similar way as the above two operators the table will be passed as text, and the two functions name will be passes as an argument and they will be formulated in an execute statement. There will be a loop that provide the object method with rows that meet the criteria by passing them through the predicate, and the rest of the rows will be returned unchanged.
Directly coding building the table method by directly coding it is also another option that we discussed, it will be designed in such way it calls the object methods inside the function rather than passing them as an argument. This is useful when having different case from the three cases discussed above. The logic is the same it takes the whole table and return it intact except for some values changed. We have also seen how the constraints are implemented in table methods. The table methods create temporary tables to store their return values and the constraint checking predicates check the constraint by access the data form the temporary table.

- Database methods are conceptually designed as functions that take all the tables, both of which has been through some changes and the others which did not. Then it check for the database constraint preservation and allow the transaction to be committed. It is important to have both the changed and unchanged tables since database constraint checking require all the tables in the relation, to check for constraint violations. This method takes all the required arguments needed for the table methods that it calls inside its code body. There is no direct call to object methods from the database, but the table method calls them and eventually they will be in the database.

4. If there is a higher level language how do we transform this in to platform specific functioning. We were only able to implement the conceptual design in PostgreSQL, due to time constraint we were not able to look at the possible ways to produce a general transformation rules. This question will be forwarded as a recommendation.

5.2 CONCLUSION

In this research project, a methodology that users can use to encapsulate there database with methods for safe transaction, has been provided. The project was a two step process initially it was designed conceptually and later it was discussed how it will be implemented in PostgreSQL using the procedural language PL/pgSQL. Finally a simple case study was put forward that worked in Eneschede spatial dataset, that deals with parcels and buildings relation.

5.3 RECOMMENDATION

The last research question of this research project has not been covered completely, the question was, if there is a higher level language how do we transform this in to platform specific functioning. We have tried to make a general conceptual design of the methodology. We recommend the transformation to platform specific functioning for future work.

In addition, when implementing the methods in PosgreSQL using PL/pgSQL there were some short comings of the system, such as it does not accept functions argument and its capability to support setof type as an argument is depreciated, therefore, if further development is made in this area the

task of functional designing would tremendously be simplified. PL/Python has the capability to accept functions as an argument therefore, this work might be also effective and user friendly if done with PL/Python as well.

LIST OF REFERENCES

[1]Postgresql reference manual - volume 2 - programming guide.

[2]Postgresql 9.0.3 documentation, 1996 to 2010.

[3]Postgresql global development group, 1996 to 2010.

[4]Enterprise architecture center of excellence, 2010.

[5]Richard P. Braegger, Andreas M. Dudler, Juerg Rebsamen, and Carlaugust Zehnder. Gambit: An interactive database design tool for data structures, integrity constraints, and transactions, 1985.

[6]Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system r. *Commun. ACM*, 24:632–646, October 1981.

[7]Sophie Cockcroft. A taxonomy of spatial data integrity constraints. *Geoinformatica*, 1:327–343, December 1997.

[8]Mazimpaka Jeab Damascene. Methodical spatial database design with topological polygon structure. Master's thesis.

[9]Birgit Demuth and Heinrich Hussmann. Using uml/ocl constraints for relational database design. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 598–613, 1999.

[10]Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[11]Christian Fahrner, Thomas Marx, and Stephan Philippi. Integration of integrity constraints into object-oriented database schema according to odmg-93. 1995.

[12]Piero Fraternali and Stefano Paraboschi. A review of repairing techniques for integrity maintenance. In *Rules in Database Systems'93*, pages 333–346, 1993.

[13]P.M. Gray. Advanced database systems. In R.J. Lucas, editor, *Lecture Notes in Computer Science*, pages 348–353, 1992.

[14]Paris Kanellakis. Database querying and constraint programming. *SIGACT News*, 25:22–87, December 1994.

[15]Khaled M. Khan, Mahesha Kapurubandara, and Urvashi Chadha. Incorporating business requirements and constraints in database conceptual models. In *Proceedings of the first Asian-Pacific conference on Conceptual modelling - Volume 31*, APCCM '04, pages 59–64, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[16]T. Sreekanta Murthy and J.S. Arora. Database management concepts in computer-aided design optimization. *Advances in Engineering Software (1978)*, 8(2):88 – 97, 1986.

[17]Joan A. Pastor. Deriving consistency-preserving transaction specifications for (view-)updates in relational databases. In *III International Workshop on the Deductive Approach to Information Systems and Databases*, pages 275–300, 1992.

[18]Tim Sheard and David Stemple. Automatic verification of database transaction safety. *ACM Trans. Database Syst.*, 14:322–368, September 1989.

[19] D. Spelt and H. Balsters. Automatic verification of transactions on an object-oriented database. In *Proceedings of the 6th Workshop on Database Programming Languages (DBPL)*, volume 1369 of *Lecture Notes in Computer Science*, pages 396–412, London, 1998. Springer Verlag. Imported from EWI/DB PMS [db-utwente:inpr:0000003045].

[20] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1975 ACM SIGMOD international conference on Management of data*, SIGMOD '75, pages 65–78, New York, NY, USA, 1975. ACM.

[21] Kalum Priyanath Udagepola, Li Xiang, Yang Xiaozong, and A. W. Wijeratne. Review of data consistency and integrity constraints in spatial databases. In *Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases*, pages 348–353, 2006.

[22] W.W.M. Vermeer and P.M.G. Apers. The role of integrity constraints in database interoperation. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB 1996)*, pages 425–435, San Francisco, CA, USA, September 1996. Morgan Kaufmann Publishers. Imported from EWI/DB PMS [db-utwente:inpr:0000003077].