SQLento

Database programming made easier

Master thesis of Roland Balk

rolandbalk@upcmail.nl, 4-1-2016

University of Twente, Formal Methods & Tools group

Supervised by: L. Wevers Msc, prof. dr. M. Huisman and dr. ir. M. van Keulen

Abstract

Relational databases take care of many problems, like storing digital information on a physical medium and providing search algorithms. However, programming relational databases can be improved a lot. One of the most discussed problems is the linguistic mismatch between the general purpose programming language (GPPL) and the relational database programming language, also known as the impedance mismatch. Since there exist a mismatch, we have explored concerns of programming relational databases from the GPPL side. For example, programming languages like Java, PHP and C# provide solutions to write SQL code in strings. Writing database code in strings increases the vulnerability for SQL injections. These concerns may not look crucial to solve and may already have a solution. But we don't expect that all found concerns have been solved by one solution. Therefore we propose a new database programming language, called SQLento, to solve these interfacing concerns.

These concerns have not been unnoticed and solutions have been proposed and developed to solve them. Ten different solutions have been summarized and compared to gain insights in the used concepts. For example, Microsoft LINQ embeds the declarative database language within the GPPL to prevent type and security concerns. But embedding the language results in a dependency with the general purpose programming language and its expressiveness. After comparing each solution we conclude that none of the solutions solve all concerns. Combining multiple solution does solve most of the concerns, but also increases the development complexity. We also realized that not all concerns can be solved by providing a new database programming language for relational databases. Thus we decided to design an architecture as well.

We propose an architectural solution to place the new database language, called SQLento, between the GPPL and the DBMS. We separate SQLento from the GPPL by providing a web service interface for the GPPL to execute database operations. Separating SQLento from the GPPL has two advantages: the architecture can optimize the database code and it makes the architecture more portable because it is not bound to any GPPL. The executable database code and the communication interfaces are produced by the compiler, which uses SQLento as input. GPPLs use the generated (web service) interface to call procedures and functions written in SQLento. When the architecture receives those calls, the corresponding database code is passed on towards the DBMS. Results of the DBMS are passed back to the GPPL and can be mapped to a nested data structure to prevent data model mismatches.

SQLento is a procedural database programming language and allows to build simple programs. Two new features have been introduced to make database programming easier: relationship declarations and nested queries. Relationships between tables can be declared once as a subfield of a table, instead of rewriting them in queries and statements. Relationships can be used within queries to reason forwards and backwards between tables, as long as the query starts with the table the relationship is defined on. This dependency introduces limitations since relationships are bound to a table. For example, if a relationship is declared on the "users" table towards the "addresses" table, then it is not possible to get from the "addresses" table back to "users" table by using the same relationship. A new unique relationship should be declared or the query has to start from the table the relationship is declared on. This constraint does not exist when using SQL and should be solved to prevent limitations in expressiveness.

Acknowledgements

Discussions and conversations about game development with Lesley Wevers brought up the idea to do research in improving relational databases. Thanks to my supervisors, Marieke Huisman, Maurice van Keulen and Lesley Wevers, I was able to do this thesis in the first place. I therefore want to thank them gratefully for their feedback and personal support.

I also would like to thank Lesley Wevers and Stefan Blom for the lectures and discussions about compiler and language design. Without them it would have taken more time to develop a prototype. The discussions with Saeed Darabi also contributed to improving the syntax of the new database language.

Last but not least, I would like to thank the whole FMT for the great Bocom meetings. I really enjoyed the conversations, discussions and experiences, which I will never forget.

Index

1 Introduction

Databases are used for storage of digital information, while taking care of aspects such as physical storage of data, security and concurrency. Developers don't even have to provide algorithms to search through the data structure. Instead developers can write declarative statements, in languages like SQL, to interact with the database. One of the most popular types of databases used today are relational databases [2, 10]. This type of database has a several advantages compared to the alternative solutions:

- The declarative language, SQL, makes queries easy to express [18]
- The mathematical expressiveness of SQL allows specifying relationships precisely [18, 24]
- The investments of companies has lead to the development of solutions to concerns of relational databases [8, 24]

But one of the most discussed problems are mismatches between the general purpose programming languages (GPPL) and the programmable interface of relational databases. For example, a GPPL like Java uses the Object Oriented data model. SQL at the other hand is a domain specific language for relational database programming. Combining a general purpose programming language with SQL has its trade-offs. For example, there exists a solution where SQL expressions have to be written in a string format to allow execution in a GPPL [22, 30, 40]. Strings can contain any kind of data and therefore increases chances of having type mismatches and SQL injections. This shows it is useful to explore the difficulties of programming relational database and using the database language. We propose to develop a database programming language between the GPPL and the database management system (DBMS), to make programming relational databases easier.

1.1 Goals

Our goal is to develop a new database language to make database programming easier by solving concerns regarding to relational databases. To achieve the main goal, the following sub goals have to be achieved:

1. Explore concerns of relational databases
2. Evaluate the existing work for solving concerns
3. Provide concepts for an architectural solution
4. Provide a new database language to solve the concerns
5. Build a prototype of the database language

This thesis provides the required background information about databases, a list of concerns with examples, a comparison of existing solutions, the architecture of the solution and a prototype of the language.

1.2 Approach

This thesis has been split-up into four sections to make the defined goals possible:

- Background information (chapter 2)
- Concerns of relational databases (chapter 3)
- Evaluation of existing work (chapter 4)
- Concepts for the architectural solution (chapter 5)
- A prototype of the database language (chapter 6)

Before starting to work on a new database language, it is important to discuss background knowledge about data models, database management systems, database transactions, SQL and finally database programming. Each of these topics contribute to having knowledge about the current situation of relational databases and is not meant to answer any research question.

Exploring the concerns provides insights in the current situation of programming relational databases. The concerns have been divided up into two categories: relational database programming concerns and SQL concerns. For each category a number of concerns and its consequences are discussed.

Since relational databases exist for a while now, solutions have been developed to solve these concerns. A selection of solutions have been explored and compared to gain information about the used concepts and which concerns they target to solve.

The goal is to develop a new database language to solve the concerns, which requires an architectural solution. Instead of deepening out the implementation details, the concepts are discussed. The solution consists out of a several components, like providing a communication interface, which are described in detail.

Last not least is the prototype language we have designed, called SQLento, to write database programs. Information is provided to know what the concepts are, how it is mapped to SQL and what it can do.

1.3 Results

The exploration of concerns has shown a diverse set of possible improvements related to programming relational databases from a GPPL and the SQL interface language. For example, there exist solutions for various GPPLs to write SQL code in a string data format. This increases vulnerability to SQL injections and data type mismatches. Another concern is that SQL forces developers to write join conditions in queries and statements, even if the join condition has been provided before. Most of the found problems are not crucial to solve, but providing an all-in-one solution makes database programming easier.

We have evaluated ten existing solutions of different categories, to get an overview of the existing work. Each of the solutions provided insights in usable concepts and its advantages and disadvantages. For example, solutions like Ferry [52] map GPPL code to SQL to make database programming easier. Each of these ten solutions do not solve all concerns but rather a subsection of them. It is possible to combine multiple solutions, although it may increase the development complexity. We also realized that separating the new database language from the GPPL would give us a few advantages: the language can be optimized for database programming and the solution can be more portable. But separating the languages is easier to achieve at architectural level. Therefore we propose to design an architecture first, before designing a new database language.

We propose to design an independent architecture to place the new database programming language (SQLento) between the GPPL and the DBMS. The architecture requires connection interfaces, compilers and a definition section. The connection interfaces take care of any connection between the GPPL, the architecture and the DBMS. The compilers are used to map SQLento to executable database code. The last part is the definition section, which keeps track of the executable database code for the web service. There are various of implementations possible for this concept, like using an Object-relational mapper in combination with a GPPL. But the Object-relational mapper is bound to a specific GPPL. This can be prevented by making a web service for the Object-relational mapper, but the developer has to provide the logic behind this. Instead we propose to develop a new database programming language which is separated from the GPPL by providing a web service for executing database operations. The code written in SQLento's language is compiled, by the compilers, to the executable database code and the web service interface. By making the solution independent of the GPPL, the database code can be optimized for database transactions, like user defined stored procedures for the RDBMS. Another advantage of this separation is making the solution more portable, since it can be connected to any GPPL and any DBMS of choice. For example, if developers decide to connect a Java application and a Python application to the architecture, only the communication interfaces have to be generated from the SQLento code. The GPPL uses the interface to call subroutines and sends it towards the web service component of the architecture. The received call is processed and executed on the DBMS, whereby the information returned can be processed to a nested data structure. The disadvantages of this architecture are performance, additional complexity, additional data storage and its optimization for RDBMSs.

SQLento supports function, procedure, relationship and table declarations. Within functions a set of statements and expressions can be used to describe the control flow and database operations. The first introduced feature is join condition declaration to prevent having to write duplicate join conditions. Developers can add a join condition to a table as a subfield and use them for the nested queries. The second feature is providing a nested data view of the relationships between tables, which brings the data model closer to the object oriented data model. Since the relationship is bound to a table, queries are limited in expressiveness compared to SQL. For example, if a relationship is declared on the "users" table towards the "addresses" table, then it is not possible to get from the "addresses" table back to "users" table by using the same relationship. This can be solved by declaring a new relationship on the "addresses" table, which contains the same join condition. This constraint does not exist when using SQL and should be solved to prevent limitations in expressiveness.

All goals have been reached, but far from complete. Much work can be done to compare other existing solutions, to add more specific features to the architecture and to make SQLento more complete.

1.4 Contributions

The thesis describes the following contributions:

- a discussion and summary about concerns of programming relational databases and SQL
- a discussion and comparison of existing work to solve the concerns
- an architectural solution
- a prototype of SQLento
- a discussion of two introduced features of SQLento

2 Background

Before starting to dive into the issues of relational database programming and providing a new database language, it is important to know the background of databases and database programming. The database section describes data models, database management systems, database transactions and one specific database language called SQL. This database language in particular is important since this thesis focuses on solving issues of relational databases, which often use the SQL language.

2.1 Databases

A database is a structured collection of records of data that is stored in a computer system [53]. Any kind of data can be stored into the database, such as text or images. To manage a database, which includes storage, reliability and security, the database management system has been invented. Since the data model is a fundamental part of the database, the next section describes this further in detail.

2.2 Data models

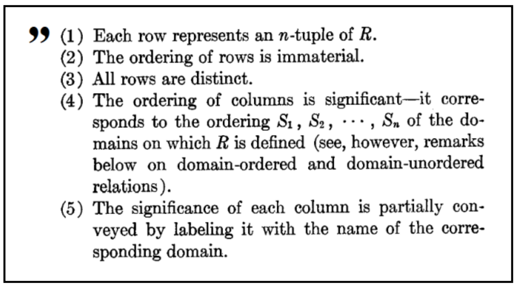
A data model is a logical model to describe in an abstract way how data is organized in a database, which is independent of the used technology. Another fundamental part of the data model is describing the set of operations to be used for the data structure. For example, the relational model has been designed for declarative statements to specify queries and data based on relational algebra. This makes database management systems responsible for supporting these statements and executing them on the database, to protect users from having to know how the data is stored and bothering about its implementation. Currently there exists many models, which are for instance:

- **Relational model** [12, 21] : using tables and relational algebra.
- **Object Oriented model** [24]: using the object oriented paradigm to represent data.
- **XML model** [35]: using an eXtensive Markup Language for data storage.

In this thesis we focus on the relational model.

E.F. Codd's relational model

The relational model has been designed to store information in N-dimensional tuples which are distinct from each other. These tuples are stored into tables with a unique key, known as the primary key, to allow the definition of relationships. Tuples stored in a table can refer to other primary keys. To make the model usable, relational algebra is used to allow reasoning over these tuples of data, like



” (1) Each row represents an n-tuple of R.
(2) The ordering of rows is immaterial.
(3) All rows are distinct.
(4) The ordering of columns is significant—it corresponds to the ordering $S_1, S_2, \cdots, S_n$ of the domains on which R is defined (see, however, remarks below on domain-ordered and domain-unordered relations).
(5) The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

Figure 1: Five properties defined by Edgar F. Codd for any kind of data relation

selections and joins. This concept was proposed by Edgar F. Codd, to prevent users from having to deal with how the data is organized [12] and dates back from the 1970's. E.F. Codd defined five properties for relational databases, shown in Figure 1.

UserID	Name	Address
1	Willem van Nassau	Somewhere 34, Grou
2	Isabelle de Graaf	Anywhere 9, Enschede

AccountID	UserID	Banknumber
19	1	546372
54	2	734542

Figure 2: Simple bank system

Figure 2 shows a simple bank system with two tables to explain how the relational model works. The first table keeps track of all registered users and the second table keeps track of all bank accounts. These tables are related to each other by the userID column (domain). The first table uses the UserID as a primary key to allow relationships. The other table, containing the bank account information, uses the UserID as a reference to the table of users. This kind of relation is called a one-to-one relation. There exists also an one-to-many relation, a many-to-one relation and a many-to-many relationship.

2.3 Database management systems

The database management system (DBMS) has been designed to prevent developers to invent ways to handle data storage, security, reliability and access. This makes the DBMS responsible for all state changes of the database. It provides a data (storage) model, a programmable interface [53, 21] and determines the number of supported features.

An example of an architectural solution with a DBMS is shown in Figure 3. It represents an abstraction of how components communicate, from users to data storage. Users do not interact with the database management system but use an application like a website instead, which is more user friendly. The (web) application takes care of the communication with the DBMS to perform the required actions. The interaction between the applications and the DBMS is made possible by the introduction of a high-level database language. The language allows to specify and interact with the database, where the number of features depends on the DBMS.

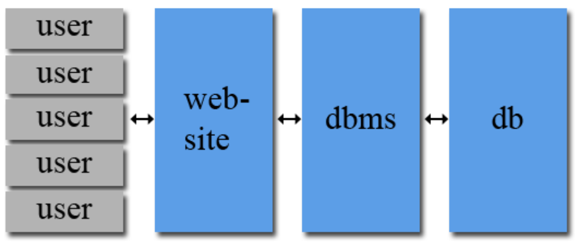


Figure 3: Example architecture between users, applications and the database management system

2.4 Database transactions and reliability

The DBMS is responsible for defining and manipulating the database and there are many cases in which this could go wrong. An invalid database state can be reached when multiple users try to get data from the database while the information is being updated, or when a server stops working while inserting new data. Database transactions have been developed to prevent these invalid state transitions. A database transaction represents a unit of work, which is written in the database language, to be performed on a database. The DBMS provides a way to isolate the application's request for database access and the database itself, to ensure no other execution can interfere with the state of the database. In case of problems, a database transaction is easy to recover. Most of the DBMS's guarantee the ACID properties [27] as a model for ensuring these reliability features. These properties are:

- **Atomicity:** entities of work are fully executed or not at all.
- **Consistency:** the database changes only from valid states to other valid states.
- **Isolation:** insurance that concurrent execution of transactions achieves the same state as if executed serially.
- **Durability:** if a transaction has been committed, the results remain visible even if events occur that causes the database to stop working.

Not all DBMS's guarantee this model and some allow to tweak how well this model can be guaranteed. For example, HBase [3] allows to trade off some guarantees for more performance.

2.5 SQL database language

SQL is one of the most used domain specific languages for relational databases, which is based on E.F. Codd's relational model. The language consists of six categories of statements [39]:

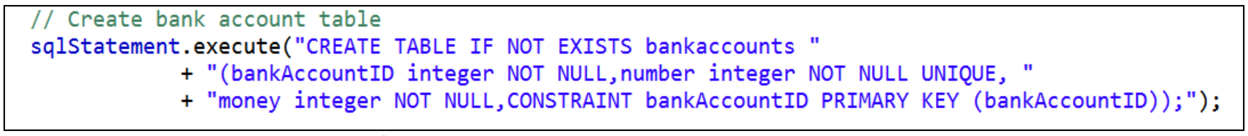
1. Data Definition Language
2. Data Manipulation Language
3. Transaction Control Statements
4. Session Control Statements
5. System Control Statement
6. Embedded SQL Statements.

In this section the data definition language (1), data manipulation language (2) and transaction statements (3) are discussed, because these contain the most fundamental statements to define and manipulate databases. After these three categories, this section continues with comparing E.F. Codd's relational model with SQL, describing the procedural language extension (PL/SQL) and finally describing the concepts used in SQL.

Data definition language

The data definition language, also known as DDL, allows to specify the structure and properties of the databases. It supports the following actions:

- Create, alter, and drop schema objects
- Grant and revoke privileges and roles
- Analyze information on a table, index, or cluster
- Establish auditing options
- Add comments to the data dictionary



```
// Create bank account table
sqlStatement.execute("CREATE TABLE IF NOT EXISTS bankaccounts "
            + "(bankAccountID integer NOT NULL,number integer NOT NULL UNIQUE, "
            + "money integer NOT NULL,CONSTRAINT bankAccountID PRIMARY KEY (bankAccountID));");
```

Figure 4: SQL DDL statements in Java code

The CREATE, ALTER and DROP statements are used for defining or modifying schema objects like databases, tables and its relationships. As you may expect, the create statement is meant for defining schema objects, the alter statement is used for modifying the schema object and the drop statement is for deleting the schema object. For example, the create statement can be used to define a new table as shown in Figure 4, which shows a specification of a table including the columns, uniqueness constraints and data types. Another important part of the CREATE, ALTER and DROP statements is the definition of indices. The SQL language also allows to enhance query performance by defining indices and by query optimizers. An index is a look-up table which contains values from specific columns of a table and a pointer towards row the data is stored in. Just like the index of a book, the location of the information can be found by searching for the information or keywords in the index. But keeping track of indices requires storage space as a trade-off. In the context of the user table example in Figure 2, queries could search faster for users with a specific name when an index is created for the name column.

The DDL supports statements for other purposes as well, like creating and declaring new privileges for database uses by using the GRANT statement or gaining knowledge in the statistics of the database by using the ANALYZE statement.

Data manipulation language

The data manipulation language, known as the DML, supports statements to adjust the state of the database by inserting new data for example [38]. It supports the following statements: CALL, DELETE, INSERT, SELECT, UPDATE, EXPLAIN PLAN, LOCK TABLE and MERGE. The call statement is used to execute a (user-defined) function or a stored procedure. The delete, insert, select and update allow developers to manipulate the stored information in the database. It can be used for inserting new rows or selecting information to be retrieved.

One may ask, why does the SELECT statement belong to the DML? Oracle has decided to put this statement to the DML because it allows operations on the data before returning the information or as Oracle states: "The SELECT statement is a limited form of DML statement in that it can only access data in the database. It cannot manipulate data in the database, although it can operate on the accessed data before returning the results of the query" [39].

The other statements, LOCK TABLE, CALL, EXPLAIN PLAN and LOCK TABLE, provide developers more control over tables, more insights in optimizations and allows merging expressions. For example, the lock table statement manually overrides the automatic locking mechanism in SQL to prevent data access or manipulation from other users.

Transaction Control statements

The SQL specification provides a set of statements to give developers control over database transactions. These statements are: commit, rollback, save and set transaction. The commit statement is a signal to send an entity of work, which contains one or more database expressions. The rollback statement is used in case of failure, to bring the database back to its previous state.

The save statement is used to store the current state of a database, usable for the rollback and set transaction statement. The set transaction statement allows the developer to specify the properties of a transaction more thoroughly. For example, the set transaction statement supports to specify to which state the database should be restored in case of a transaction failure.

SQL versus the relational model

Relational databases generally use a modified version of E.F. Codd's proposed relational model. There are a few properties of Codd's list, described in Figure 1, which do not match with the DBMS implementations. The first difference can be found in the second property: "The ordering of rows is immaterial". Relational databases often have options to order datasets, which can be useful for analytical purposes or scoreboards. Another difference can be found in the third rule, which defines that each row should be distinct, while the SQL specification allows to add duplicate data in a table. This can be achieved by creating a table with no constraints on the distinction of its values. There are still properties that do hold, like the first one: "Each row represents a N-tuple of R". Each row in a relational database still represents a tuple of data.

Extensions to SQL

The SQL language has been extended with procedural languages such as PL/SQL for Oracle's databases, Transact-SQL for Microsoft databases and PL/pgSQL for PostgreSQL databases. Each of these extensions allow developers to combine SQL statements with procedural language constructs, like functions, procedures, triggers, conditions and try-catch blocks [36]. A stored procedure is a group of SQL statements compiled into one execution entity. A trigger is comparable to a stored procedure but, as you may expect, is only executed or triggered under specified conditions. A function is similar to stored procedures except for a few properties: a function has to return a value and can't have side-effects like adjusting the state of a variable outside the function. By having this property, functions can't contain all the statements supported by the extension. For example, try and catch blocks can't be used within the function. These execution entities can be defined by using the "create" DDL statement [34, 37] and can be called by using the "call" statement of the DML. Many of the DBMSs support the procedural extension, but each of them differs in supported statements. For instance, SQLite doesn't support functions and stored procedures [50] while PostgreSQL does.

Programming paradigm of SQL

A programming paradigm is the concept or style used for programming languages. SQL's database language consists of declarative statements and is therefore a declarative language [58]. For example, the code shown Figure 4 provides a command to define a new table, its columns and the constraints on the columns. Developers don't have to provide code to tell the database how to create the tables and which algorithms to use. The PL/SQL extension, and equivalent extensions, introduce the procedural paradigm and some imperative statements to have control over statement execution. For example, loops, try-catch and condition statements can be used to adjust the execution order. This matches with the imperative paradigm, which is the concept of writing how the statements should be executed. Developers can also influence the statement execution by using try-catch, if and while statements [42].

3 Relational databases' concerns

Programming relational databases can be improved a lot, and to prove this we have gathered concerns from different sources: papers, books and user experiences from websites like "stackoverflow". One of the most discussed problems is the impedance mismatch [2], which is a linguistic mismatch between the GPPL and the declarative relational database languages. Since there exists a mismatch between the languages, it is important to explore the concerns further. This chapter starts with explaining the impedance mismatch and the source of the problem. Then it continues with summarizing concerns from programming relational databases from a GPPL and from the SQL language itself. Each concern is explained by own created examples.

3.1 Impedance mismatch

When the relational model was invented, database languages have been optimized for taking care of data storage without bothering developers about how it is done. General purpose programming languages at the other hand have been developed to provide freedom of expressing how programs have to work by influencing the execution order and state changes of the program. Programming languages use a variety of paradigms and data models to provide this capability. These differences between general purpose programming languages and relational database languages causes a linguistic gap to appear which is known as the impedance mismatch [2, 19]. The term "impedance mismatch" does not point to one problem, but is used to address multiple problems. One of the most discussed problems is the mismatch between the object data model and the relational data model. Engineers and scientists found out that mapping objects to the relational model can be challenging, when objects and pointers have to be mapped to tables and relations [18]. Currently a variety of solutions are available like Object Oriented databases and Ferry, which are discussed in section 4.

```
String insertUser = "INSERT INTO users (name, username, password) VALUES (?,?,?)";
PreparedStatement createUserStatement =
        dbConnection.prepareStatement(insertUser);
createUserStatement.setString(1, name);
createUserStatement.setString(2, username);
createUserStatement.setString(3, password);
dbConnection.commit();
```

Figure 5: Impedance mismatch example

The Java code shown in Figure 5 inserts a new user into a database, by declaring an SQL statement in a string format and setting the values for the exclamation marks. Java uses objects to store information and methods to operate on data, while SQL requires plain information and DDL or DML commands to operate on the data. Java objects have to be flattened to be usable for database commands. The linguistic differences causes concerns to appear, like writing commands in a string format to allow execution when using the Java Database Connection library. Strings can be easily adjusted or manipulated and may contain code to cause unwanted behaviour.

3.1.1 Source of trouble

The impedance mismatch problem is caused by the difference in data models used for general purpose programming languages and relational databases. General purpose programming languages often use the Object data model or Object Oriented programming, which uses objects for data storage and uses one-to-one pointers to define relationships. The relational model uses tables for data storage and use primary and foreign keys to define relationships. There are ways to convert the relational model to the object oriented model [18], but the differences in structure complicates the mapping process. For example, objects containing complete user information about the name, address and bank information could be split up into three tables to contain the information, including the relationships. But it is also possible to store all information into one table.

The difference in data model is mainly the source of the problem, but the inequality between the paradigms complicates the matter as well. As stated before, database languages like SQL are often described as declarative languages while it supports a limited imperative way of writing statements. Technically it is possible to make database languages fully imperative, but that wouldn't make sense at all. The DBMS has been designed to prevent developers from bothering how data is stored, how the reliability can be maintained and etcetera. By using the imperative paradigm the developer would have been forced to design algorithms for queries and other expressions to retrieve data. For SQL expressions it is easier to describe what has to be achieved rather than how. Another reason why the declarative paradigm works well for relational databases is the ability to optimize the expressions. For example, the SQL interpreter can decide to execute certain statements in a query parallel by using multiple threads to enhance the performance [6]. General purpose programming languages at the other hand can use a variety of data models and paradigms, since they have been designed to describe the behaviour of an application. The most popular are the imperative languages that use the Object Oriented data model, where developers can describe in detail how the program works and optimize it for specific tasks. These differences in paradigm have an impact on how well the languages can work together.

SQL (Declarative)	UPDATE users SET username="default" AND password="123" WHERE uid >= 0 AND uid < 10000
Java (Imperative)	`HashMap<Integer,ArrayList> users = new HashMap<Integer,ArrayList>(); for (int index = 0; index < 10000; index++) if (users.containsKey(index) && (users.get(index).get(0) == "Roland")) { users.get(index).set(1, "default"); users.get(index).set(2, "123"); }`

Figure 6: Declarative versus imperative

Figure 6 shows the differences between declarative and imperative programming and shows the impact of the used paradigm. Imagine the bank needs to reset the username and passwords from userID 1 to 10.000 because of a security breach. The SQL language requires only three lines of code to tell the DBMS to update users with "default" as username and "123" as password. In Java you will need more code to describe actually how this happens, like checking if the userID exists and iterating through them. Now if the Java code was actually executing SQL code for each step, it would require 10.000 update executions. If a database was able to know the Java code is going to execute many SQL statements sequentially, it could optimize it. But the database is unaware of what the application is going to execute, how many times and which commands. It just responds by its transactions as applications are responsible for communication with the database. This makes optimizations of external imperative programs rather difficult.

3.2 Concerns

In the previous section we have described linguistic mismatches between relational database languages and general purpose programming languages. Since there exist mismatches, we have gathered a selection of concerns divided over two categories:

- Relational database programming concerns: issues related to programming a relational database from a GPPL.
- SQL concerns: issues related to the relational database programming language.

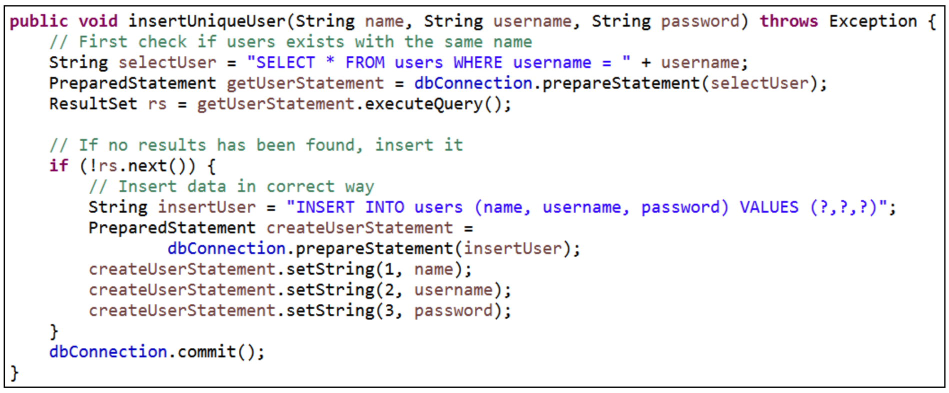
Each of the found concerns have been explained with an example, written in Java or Python by using the JDBC and SQL connector library.

3.2.1 Relational database programming concerns

Programming databases from a GPPL or writing queries can be challenging, especially when solutions are provided to write database code in strings. We have found the following concerns:

1. Errors and safety concerns of mixing code

Programming languages like Java [40], C# [30] or PHP [22] often provide solutions (like libraries) to write SQL code in a string data format, even if there exists alternative ways to write queries. SQL code written in strings are not checked for mistakes, as shown in Figure 7. The code shown in the example is used to check the uniqueness of the username, by retrieving all users with the same username. Then an uniqueness check is used before inserting a new user. All SQL statements are written in strings, which could lead to SQL errors like misspelled columns or wrong SQL commands for the database. In practice this would occur more often when the queries are more complex compared to the example in Figure 7. These errors are captured by the database management system but are only visible at runtime. It requires developers to execute the code to verify the expected behaviour.

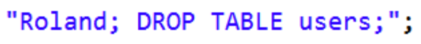


```java
public void insertUniqueUser(String name, String username, String password) throws Exception {
    // First check if users exists with the same name
    String selectUser = "SELECT * FROM users WHERE username = " + username;
    PreparedStatement getUserStatement = dbConnection.prepareStatement(selectUser);
    ResultSet rs = getUserStatement.executeQuery();

    // If no results has been found, insert it
    if (!rs.next()) {
        // Insert data in correct way
        String insertUser = "INSERT INTO users (name, username, password) VALUES (?,?,?)";
        PreparedStatement createUserStatement =
                dbConnection.prepareStatement(insertUser);
        createUserStatement.setString(1, name);
        createUserStatement.setString(2, username);
        createUserStatement.setString(3, password);
    }
    dbConnection.commit();
}
```

Figure 7: Method for inserting an unique user

Another issue is the possibility of adjusting the string containing the SQL code, like shown in Figure 7. A variable has been added after the SQL code which contains information about the searched user. The variable could contain a username but also SQL injections. This leads to serious security vulnerabilities and unsecure program development if the developer is unaware of the problem. There are serious consequences if the "username" parameter contains the value shown in Figure 8.



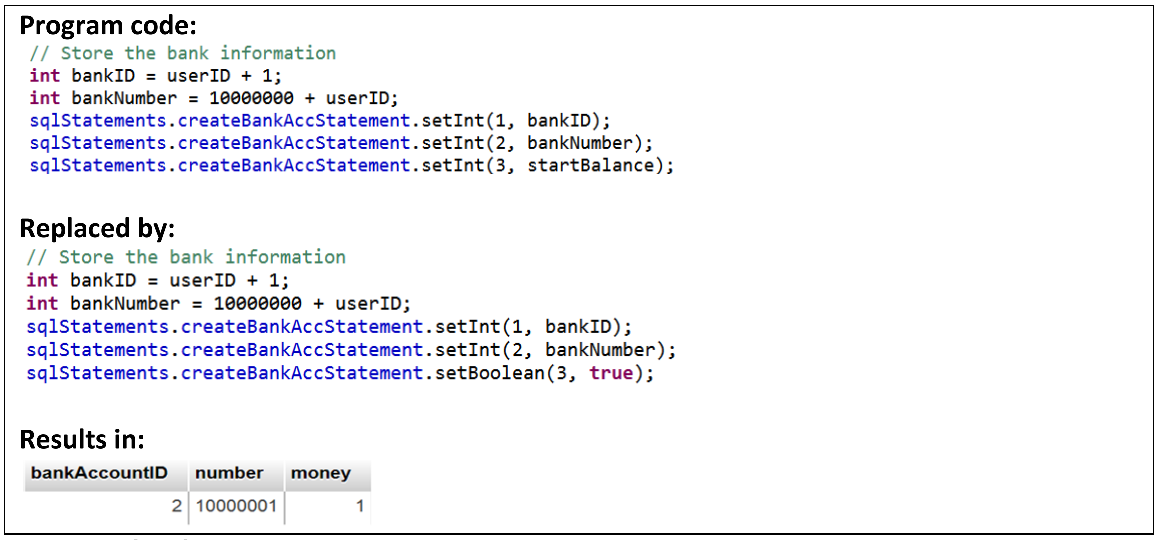
```java
"Roland; DROP TABLE users;";
```

Figure 8: SQL injection example

Execution of this statement would immediately lead to deleting the users table, which should be prevented. Currently there exists solutions to this problem like specifying which values belong to which value placeholder, as shown in the setString methods in Figure 7. The SQL statement should contain question marks to describe where values can be added. Each of these placeholders has a number, where number 1 is the first exclamation mark and so on. This methodology prevents SQL injections and is advised by many sources, like Oracle [40]. But developers can still be unaware of this solution and it introduces more complexity to the program code.

2. Types and values mismatches

Strings can contain any type of information, like floating points or characters and therefore cause type and value mismatches. This means it is possible to insert a floating point number in a column from a table that only allows integers. Some databases like MySQL have an automatic conversion to the right data type, which could lead to unwanted database states, explained by the example shown in Figure 9.



Program code:

```
// Store the bank information
int bankID = userID + 1;
int bankNumber = 10000000 + userID;
sqlStatements.createBankAccStatement.setInt(1, bankID);
sqlStatements.createBankAccStatement.setInt(2, bankNumber);
sqlStatements.createBankAccStatement.setInt(3, startBalance);
```

Replaced by:

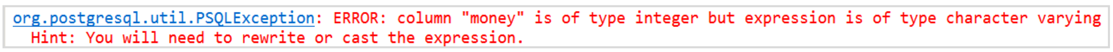
```
// Store the bank information
int bankID = userID + 1;
int bankNumber = 10000000 + userID;
sqlStatements.createBankAccStatement.setInt(1, bankID);
sqlStatements.createBankAccStatement.setInt(2, bankNumber);
sqlStatements.createBankAccStatement.setBoolean(3, true);
```

Results in:

bankAccountID	number	money
2	10000001	1

Figure 9: Code replacement

Typing mistakes with setting the wrong type of value can easily happen if the developer isn't notified, as shown in Figure 9, where the setInt is replaced by setBoolean. Some relational databases like MySQL will automatically convert the value to the other types, which results in storing a 1 or a 0 into the integer field. This is not the case for every data type because not every case is convertible. For example, storing Strings into columns that require integers would result into type-mismatch errors. If PostgreSQL is used instead of MySQL, the outcome of the result would be different because this DBMS doesn't try to convert the values automatically. It will return an error instead if type mismatches occur, as shown in Figure 10.

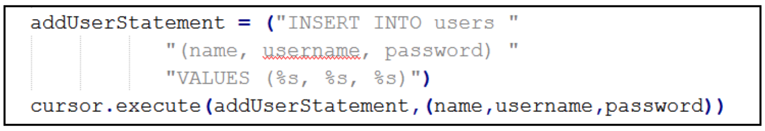


```
org.postgresql.util.PSQLException: ERROR: column "money" is of type integer but expression is of type character varying
  Hint: You will need to rewrite or cast the expression.
```

Figure 10: Type mismatch error in PostgreSQL

But for both SQL databases the mismatch is only noticeable when executing the statements and introduces additional complexity of setting values to value placeholders. Type checking at forehand could prevent issues before execution.

One may argue that the increase of complexity is partly caused by the used programming language, since Java is a statically typed language. In a statically typed language the types have to be declared as well and users have to set the value types in the given example (setString method). By using a dynamically typed language the code complexity is indeed reduced as shown in Figure 11. The example shows Python code to insert a new user in the database and uses the "%s" tag instead of question marks, but it means exactly the same. In Python it is not required to use the set method to add values to the "%s" tag, which does reduce the code complexity. But it is still possible to insert the wrong value types and inject the wrong type of data. This problem is visible after executing the SQL code and therefore requires additional code to prevent this concern.



```
addUserStatement = ("INSERT INTO users "
                    "(name, username, password) "
                    "VALUES (%s, %s, %s)")
cursor.execute(addUserStatement,(name,username,password))
```

Figure 11: Python code for inserting a user in MySQL

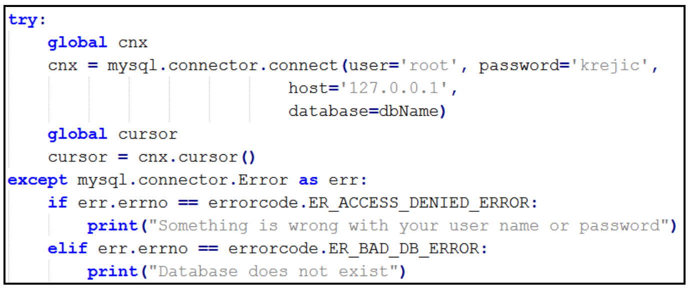
This concern shows that programming languages and relational databases handle data types differently. A solution to this issue is to make the new database language responsible for the type checking and reporting mismatches. The advantages would be:

- **Specific error reporting:** mismatches could be reported back to the user, dependent on the used DBMS.
- **Type checking as a separated concern:** the new database language could take care of it, independent of which programming or scripting language.

There are consequences as well as the new language should be statically typed. It will force users to define the type used for any kind of data but also prevents mismatches.

3. Database connection handling

Managing database connections from the GPPL side can be difficult when many connections have to be managed. Connections have to be set-up for every new database connection as shown in Figure 12. This methodology forces the user to be responsible for handling each database connection correctly. This could be error prone since developers may accidently close a connection at the wrong moment. Efficiency is also an important concern because opening a new connection is an expensive operation. It takes time to set-up a new connection. Execution time can be a crucial factor for some cases, especially for websites. Currently there exists solutions to manage database connections and handle them efficiently by allowing to reuse existing database connections, called database or connection pooling. But it may require developers to use additional libraries to support this, like JDBC or DBCP [4]. It would be an advantage if developers do not have to be bothered with database connections and its optimizations because it would add extra complexity.



```
try:
    global cnx
    cnx = mysql.connector.connect(user='root', password='krejic',
                                  host='127.0.0.1',
                                  database=dbName)
    global cursor
    cursor = cnx.cursor()
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your user name or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exist")
```

Figure 12: Setting up a DB connection

4. Database transactions

The auto commit option allows developers to execute database operations separately. When auto commit is disabled, developers can bundle database operations by providing the commit signal at the end [51]. General purpose programming languages use libraries or plug-ins to connect to databases, which allows developers to enable or disable this auto commit option. Those solutions also provide the "commit" and "rollback" methods to provide control over the database transactions. But placing the commit signal at the wrong section of the code or forgetting to place this signal, has a large influence on the behaviour. The program code shown in Figure 7 uses an SQL query to retrieve users with an username, checks whether users exist with such an username and inserts a new user if the username is unique. When auto commit is disabled, the user's uniqueness cannot be guaranteed. Another computer application may insert the same user somewhere between executing the "select" and the "insert" statements of the code. By placing the commit signal at the end and setting the auto commit to false, this can be prevented. But the developer can accidently place the commit signal after the "select" and the "insert" statement, which results in a different execution sequence compared to the expected behaviour.

We have found the following concerns to the auto commit option:

- **Auto commit handling:** Enabling or disabling auto commit in the library or plug-in from the GPPL can be forgotten. This may result in a different execution behaviour.
- **Commit signals:** When the auto commit option is disabled, developers have to provide the commit signal manually. The commit signal can easily be placed at the wrong location or even forgotten.
- **Database transactions:** It could happen that a transaction suddenly fails because of an unstable internet connection or another type of intervention. A transaction is cancelled completely, while retrying may be useful in some cases. By cancelling, more communication traffic is required to retry the transaction.

First of all, disabling the auto commit option has consequences. Developers are forced to use the commit signal, which can be forgotten or misplaced. But disabling the auto commit allows to combine multiple database operations into one transaction and prevents data integrity issues. To have the advantage of bundling database operations and preventing misplaced commit signals, we could search for a solution to place the commit signal automatically. For example, the commit signal can be added at the end of the body of a function or stored procedure [47]. The consequences of this automated process requires more research. But automating this process reduces the freedom of the developers to decide where to use them and may even result in a suboptimal statement execution. The last issue, regarding to database transactions, can be solved by providing a retry option. When connection failure occurs, developers can choose to set an option to retry the database transaction. Developers therefore don't have to provide the logic for the GPPL to implement the retry option. But the use of the retry feature should be limited, because it is useless to retry a transaction when constraints are being violated.

5. Constraint checking

Constraint checks on values are often done at the GPPL side [22] before inserting it into the database, while it can be centralized at the database side. This creates a dependency between the GPPL and the database for managing data correctly. Another concern is that not all database management systems support constraints fully, like MySQL [33]. Making the DBMS responsible for constraint checking has a few advantages:

- **Shorter program code at GPPL:** Constraints can be checked at the data storage side. It makes the database more responsible for handling the data.
- **Centralized responsibility:** Applications are less responsible for data handling, which can be taken over by the database.
- **Error reporting:** DBMSs often support database constraints already and therefore take care of constraint violation errors as well. Developers don't have to provide the logic behind it.

Putting the constraints at the DBMS side has the following trade-offs:

- **Reduced performance:** Violation of constraints is detected at the DBMS side instead of preventing it at the GPPL side. A response has to be sent back from the DBMS to the application, which causes delays.
- **Increased complexity:** Database code gets more complex if constraints are added to the database program. Therefore the complexity has moved from the GPPL to the DBMS.

Still the reduction in performance has to be measured to know how large the impact will be and the increase in complexity is not expected to be of a concern either.

3.2.2 SQL concerns

We discuss the possible improvements for SQL in the following subsections.

6. Syntactical improvements

SQL allows declaration of commands in a simple way as shown in Figure 13, but the syntax can be simplified. Queries could end up as long declarative statements, which can be written in shorter commands which are more comparable to programming languages.

Language	Code
MySQL	`SELECT EXISTS(SELECT 1 FROM `users` WHERE username = 'Roland')`
Possible syntax	`Users[username = "Roland"].exists`

Figure 13: SQL versus Java

The code shown in the example checks if a user with the name "Roland" exists in the database. The SQL statement is twice as long as the Java Hash Map operation. If the syntax is adjusted, it could lead to three improvements:

- **Shorter syntax:** it is possible to shorten the syntax without sacrificing functionality or readability.
- **Decreased complexity:** if the length is decreased, the overall readability of the code would increase.
- **Similarity with GPPL syntax:** the syntax can be adjusted to match more closely with the GPPL to make it easier for software developers to program.

The syntax adjustments could bring the database language closer to the GPPL's syntax. But it requires more research to know which language elements can be mapped without consequences. For example, imperative loops can't be mapped directly to the declarative database language. One way of checking whether SQL code can be written more shortly, is by dividing the statement up in essential and not essential parts. For example, the query could be shortened by addressing the table the query is executed on instead of putting the "SELECT" keyword in front of it. But the goal is to keep the expressiveness of the language and not to break any good solutions.

7. Database functions and procedures

Developers tend to design subroutines used for databases at the application side rather than the database side. For example, a log-in function for websites is often written in PHP instead of using user-defined functions of the PL/SQL extension [22]. One of the reasons may be the complexity of writing functions and procedures. We have found a several improvements to make development of database functions and procedures easier.

The first noticeable improvement is the simplification of the syntax of stored procedures and functions. General purpose programming languages like Java and Python allow the user to write a function in a shorter way, which lowers the amount of code as shown in Figure 14. By getting rid of information like the $$ sign or the "CREATE" statement, the code could already be reduced in length. Since the method in Java and the function in SQL share similarities, the syntax could even be mapped from one to another. They both have an identifier, parameter support, a body and the return type. By replacing syntactical elements no harm can be done to the SQL language. But in SQL there is a fundamental difference between stored procedures and functions. Stored procedures have more possibilities related to the error handling and feature support [39]. In particular, the body of user-defined functions can't contain try- and catch blocks for error handling because it isn't allowed to have any side effects.

Language	**Code**

SQL	`DELIMITER $$`
	`CREATE FUNCTION hello_world()`
	`  RETURNS TEXT`
	`  LANGUAGE SQL`
	`BEGIN`
	`  RETURN 'Hello World';`
	`END;`
	`$$`
	`DELIMITER ;`

Java	`public String helloWorld() {`
	`    return "Hello World";`
	`}`

Figure 14: Functions in SQL and Java

The second issue is about not being able to add try-catch blocks to user-defined functions. This could be useful because developers may want to validate parameters of a user-defined function before using them. There exists a work-around to solve the issue, by returning numerical error messages in case of constraint violations [14]. Adding try-catch blocks support would make error reporting easier for developers.

8. Relationships and data model

Relationships between tables are defined by primary and foreign keys in SQL. When relationships are used in queries, the developer has to point out how the tables are related to each other, even if the exact same query has been executed before. SQL therefore forces the developer to define the relationships repeatedly, which could result in duplicate code. One way to reduce this duplicate code is by putting repeatable queries in subroutines, but this does not prevent developers from having to write the same relationship over again. Developers may still have to write the same relationship in an update statement and a query in the same subroutine. Figure 15 shows an example where a relationship has been used to retrieve bank information from a user and to update the user's name with a certain bank account number. This means that the same relationships can be defined multiple times for different cases. A better solution would be to declare the relationship between tables and columns once after the table definitions, instead of declaring the relationships in statements and queries. Solving this concern results in less duplicate code.

Query

```
SELECT b.*
FROM users u, bankaccounts b
WHERE u.uid = b.uid AND u.name = 'Roland'
```

Update

```
UPDATE user u
INNER JOIN bankaccounts b on
    u.uid = b.uid
SET u.name = 'Lesley'
WHERE b.number = 123456
```

Figure 15: Query and update statement using relationships

3.3 Conclusion

One of the most discussed problems is the impedance mismatch, which is the linguistic difference between the GPPL and the declarative database language. GPPL's often use objects and pointers for their data structure, while relational databases use tables and relationships. When an object oriented GPPL has to access a relational database management system, concerns appear related to these linguistic mismatches. Writing database code in an imperative way would not be a good solution, since it ends up in many database operations, which can be written in one single SQL query.

We have gathered and explored eight different concerns from relational database programming from a GPPL and the SQL language. For example, the developer is forced to put the join conditions in queries when multiple tables are used, even if they have been provided before. But most of the problems are not crucial to solve, because each of the concerns may already have a solution. But providing a total solution for all concerns makes database programming a lot easier. Some concerns may be easier to fix by providing an architectural design. Using a database language within a GPPL could create security concerns and data type mismatches. One solution is to separate those languages, which is easier to do at architectural level. Therefore it may be useful to design an architecture first before designing a new database language.

The next chapter continues with evaluating the existing work on solving these concerns (see chapter 4).

4 Evaluation of existing work

The concerns described in section 3.2 are not unknown and many engineers and scientists have been working on the issues [1, 2]. Solutions have been developed to solve relational databases' shortcomings, like Google's BigTable [13] or Object-Relational databases [23]. Each solution provides a way to cope with the issues of relational databases and has its advantages and disadvantages. This section describes ten solutions of different kinds, from new database solutions to middleware technology. Each of these solutions are compared to gain insights in which concerns they address to solve.

4.1 Object Oriented database management systems

Object Oriented database management systems (OODBMS) are fundamentally different from relational database management systems, since they differ in the used models [1]. The relational databases are based on the Mathematical Relational model for expressing relations between data, while Object Oriented databases focuses on the aspects of the Object Oriented paradigm [23, 44]:

- **Inheritance:** making subversions of a specific class which helps out factoring shared specifications.
- **Encapsulation:** packing data into one class and hiding data.
- **Polymorphism:** allows to create variations of a class.

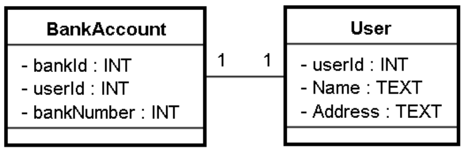


Figure 16: Object Oriented design of Table 1 and 2

These features allow us to specify the data structure more naturally compared to relational databases, as the data can be defined as classes, subtypes and relations. An OODBMS uses pointers in objects to refer to other objects to create relationships, which can be represented as an interconnected graph. An instance of a class may refer to another instance of a class, which is visible in Figure 16. This example shows an object oriented design of Figure 2. Object Oriented databases have two advantages: the underlying data structures can be more complex since Object Oriented features are supported and the impedance mismatch has been solved since the data model matches with object oriented programming languages. How well the impedance mismatch has been solved, depends on the DBMS. Most OODBMSs use the Object Query Language (OQL), which is a variant of SQL for Object Oriented databases [44, 57]. OQL was designed to work more closely with programming languages like Java and C++ because it uses objects. There also exist newer solutions which use different way of expressing queries, like the Realm DBMS [45]. So it is hard to tell how well it solves the impedance mismatch because it depends on the solution. But for several reasons it didn't replace the relational databases yet:

- **Investments** [24]**:** Object Oriented databases arrived when relational databases are up and running. Converting databases to a different type costs time and money. It could even create more complicated scenarios.
- **Performance** [23, 44]**:** Companies haven't succeeded in developing a competing OODBMS in terms of performance and scale in comparison to RDBMS's.

- **Complexity** [44]**:** Optimizing queries can be challenging, especially when the graph of object structures gets complex. A query can contain many classes and objects to query on, which generates a longer list of referred objects in the statement.

The complexity of queries and data structures may be the main issue because performance issues can be solved by improving the implementation. Also the costs should not be a major problem if these DBMSs are better than relational DBMS's. Even if this solution isn't the most popular choice, there are still design choices which can be reused. Inheritance and polymorphism can be useful for databases, as it can be useful to have data subtypes. For example, if you have different types of users you'll easily get a hierarchical structure of shared properties. In relational databases it would be useful to inherit tables with certain properties and make subtypes of a user. Encapsulation is less interesting because it wouldn't make sense to hide data or functionality, especially in databases.

4.2 Object-Relational database management systems

The Object-Relational database management system (ORDBMS) is a hybrid solution [8, 26] between the relational database management systems and object-relational mapping solutions. It uses the SQL language for database definition and manipulation, it uses the relational model for data storage and extends the data model with object oriented features [44]. To put it more simply, it is a RDBMS with OO features to extend the data model. Which features are supported to extend the data model depends on the ORDBMS, but the most common features are inheritance and object representation of relational information. By supporting user defined data types, it is possible to bring the data model closer to the data model used by the GPPL, just as shown in Figure 17, compared to relational databases.

```
CREATE TYPE paymentStatus AS ENUM ('new', 'open', 'closed');

CREATE TABLE payments (
    id integer PRIMARY KEY,
    fromAccoutId integer,
    toAccoutId integer,
    description text,
    status paymentStatus
);
```

Figure 17: User defined type to store payment status

These additional features lead to the possibility of writing more complex queries than the RDBMS allows to. A nested structure of objects and fields can be declared in a query for an ORDBMS [20, 48], while a query for RDBMS only requires direct references to tables and columns. Figure 18 shows an example query to retrieve users which are listed in the employee class. When this query is executed, the ORDBMS has to analyze the objects and has to map it to the relational data model as stored in the relational database. It is possible to retrieve a network of objects instead of rows of data as well, but this is optional.

```
stmt.executeQuery("SELECT u.name FROM Users u WHERE Employee(u.name) = 1");
```

Figure 18: An ORDBMS query written in Java

One of the concerns is clearly visible in Figure 18, because the ORDBMS's SQL code is written in a string format as suggested by IBM's data centre website [16]. This drawback is not directly related to ORDBMS's since the Java DataBase Connectivity plug-in is responsible for causing this concern. But the linguistic differences between the GPPL and the ORDBMS still exist, even if the data model mismatches are targeted to solve. But a bigger concern is the variety in support of features.

4.3 NoSQL

The NoSQL category databases are by no means attempts to get rid of the SQL language, but it stands for "Not Only SQL". It covers a wide range of data storage solutions that solve specific issues related with the relational databases today. For this reason NoSQL databases are not primarily meant to replace relational databases [25]. One example is Google's BigTable solution focusing on distributed data storage for handling large datasets more efficient than relational databases. It uses one big multi-dimensional sorted map as storage model, where data is sorted by row key, column key and a timestamp (3 dimensional). The data is stored as a key-value pair where multiple instances of the data can exist as long as the timestamp is unique [13]. There are many other examples, such as MongoDB (key-value storage) if more speed is required [32] and BaseX for document storage in the eXtensive Markup Language format [5]. Each of these NoSQL solutions are equal and vary greatly in which problem they solve. Although many tend to integrate relational features in their databases over time.

4.4 Microsoft LINQ

Microsoft has released LINQ [31] to make the gap between the GPPL and SQL smaller and focuses on solving the impedance mismatch. LINQ is a query language that allows developers to express queries in a general purpose programming language, where statements look similar to SQL statements. But LINQ also supports describing queries in a way comparable to Lambda expressions. Another key selling point is the length of the code, as Microsoft has shortened the expressions. But the declarative database code is mixed up with C# or Visual Basic code, and is therefore bound to specific GPPLs. An example of LINQ code is shown in Figure 19.

```
DatabaseContext context = new DatabaseContext();
// Create a new user
User newUser = new User();
newUser.name = "Roland";
newUser.username = "RolandBalk";
newUser.password = "test123";
context.users.InsertOnSubmit(newUser);

// Search for a user called Roland
var query = from a in context.users where a.name.Contains("Roland") select a;
```
Figure 19: LINQ code example

The program from the example shows two parts of the LINQ solution: an insert expression and a query. A User object is created and the values are set to make a new entity of a user. This User object contains fields for all the columns from the User table: the name, username and password. One of the columns isn't set, which is the userID, because it is automatically generated by the database. The "InsertOnSubmit" makes sure the new user is added after giving the submit signal. After calling the "insertOnSubmit", the results of the query are stored in a variable which searches for a user called "Roland". This example shows that LINQ uses two kinds of language concepts to close the gap between relational databases and programming languages: the Object Oriented paradigm to represent new data and the declarative paradigm to describe queries. It is an improvement in a several ways: the chance of writing wrong statements has been lowered, the syntax has been shortened for queries and type mismatches are prevented. But LINQ is only compatible with Microsoft's programming languages and the written database statements are bound to the expressiveness of the programming language, especially the programming concepts. Since multiple paradigms are used, data objects can get mixed up with declarative queries and could result in less readable code and long queries as shown in Figure 20. This example creates a User object and calls the constructor. The getUsers contains the result of asking the database for users with the name "Roland" and with friends that share the same name.

```
User searchedUser = new User(12,"Roland","University of Twente");
var getUsers = from users in svcContext.Users
                    join friends in svcContext.Friends
                    on friends.uid equals users.uid
                    where users.Name.Contains(searchedUser.getName())
                    where friends.Name.Contains(searchedUser.getName())
                    select new
                    {
                     name = users.FullName,
                     friendsName = friends.FullName
                    };
```

Figure 20: LINQ example of mixing up objects and methods with queries

4.5 Microsoft SQL Templates

Microsoft saw a chance to improve the reusability of code by using templates [28]. A template is a piece of reusable code in the form of a stored procedure. Microsoft's editing tool called Visual Studio, allows to create templates easily and helps with building up the basic structure. A template supports various SQL statements like queries, tables and if statements. Other database management systems like MySQL support stored procedures as well, but the name "template" is a bit misleading and can even be confusing because PostgreSQL uses templates to model databases [41]. It's a good idea to make development of stored procedures easier, as it may accelerate the development process.

4.6 Ferry

Ferry is a database programming language which is mapped to SQL:1999 statements [52]. The language shares syntax and idioms with programming languages like Haskell or C# and supports a variety of language entities to specify and write database transactions, like functions, tuples and lists. The goal of this language is to bring the two worlds, programming and database technology, closer together. Ferry is written in an environment called FerryDeck, which provides an editor, a compiler and an execution environment. Unfortunately the language is not available for the public which prevents a more in-depth research. But the concept shows possibilities of mapping programming language code to SQL code, even if Ferry uses an old version of SQL.

4.7 Iris Programming Language

Iris Programming Language (IPL) is a functional embedding of OSQL, which is a database programming language for the Iris OODBMS. The Iris language allows to write computational functions and procedures in a syntax comparable to GPPLs. The goal is to support more complex functions at the database side and to make them more reusable. Functions that have to be built at the GPPL side could be developed in IPL and is accessible from any GPPL. The syntax of the code is kept closely to C or Pascal to make it easier for developers to work with and the language uses an imperative syntax to write code. It supports basic features for defining and updating functions and defining queries. The language does also support recursion, conditions, iterations, sequences and variables. Therefore it is easier to define constraints as well. An interpreter is used to map the syntax to OSQL code to make it executable. Even though the solution looks quite promising, not much can be found about the language except the main paper, which dates from 1991 [17].

4.8 Middleware

Middleware is a software solution that acts as a layer between applications, like a web server which makes websites accessible for web browsers. There exists all kinds and forms of middleware, but we are mainly interested in database middleware. There exists solutions like:

Hibernate [46]**:** is an object-relational mapping framework for Java to let the Object Oriented paradigm work together with relational databases. It partly solves the impedance mismatch by mapping Object Oriented data to relational databases. This solution is used in the GPPL, unlike the ORDBMS that solves the mapping process at the database side. Tables, columns and rows are written and accessed as objects as shown in Figure 21. If data needs to be retrieved, queries can still be written in a String data format [55], which may cause security and type issues. It does provide an alternative option to solve this concern, by working with data objects instead of tables and providing information how to map the data objects to relational information.

```
String username = "Roland";
String name = "Roland Balk";
String password = "123";

Query query = session.createQuery("SELECT username FROM User u WHERE u.username =:
username");
query.setParameter("username", username);
List results = query.list();
if (results.isEmpty()) {
        try {
            tx = session.beginTransaction();
            User user = new User(name,username,password);
            int userID = (Integer) session.save(user);
            tx.commit();
        } catch (Exception e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
}
```

Figure 21: Hibernate example

One of its advantages is that no new database management system is required to use this mapping and it can be connected to any existing RDBMS. But Hibernate is built for Java, because it merges the declarative database language with the GPPL. Hibernate is not fully bound to Java, because there exists a comparable solution for .NET languages known as N-Hibernate. For other programming languages you need to find alternatives like Django [11], which is meant for Python.

Java Object Oriented Querying [9] (JOOQ)**:** is an object-relational mapping for Java, which is similar to Hibernate. But this solution only maps tables to objects, while the Object Relational Mapper is able to map tables and rows to Objects. It targets to make database programming easier for software developers. Queries are written as method calls, as shown in Figure 22. This design choice has led to type-safety for creating queries but reduces the compatibility to SQL databases only [7]. Another implication is that the solution is bound to the expressiveness of Java. This means that the syntax of JOOQ can't be optimized because a long list of method calls could get complicated to read.

```
Factory create = new Factory(connection, dialect);
create.select(username)
      .from(Users)
      .where(Users.username.equal("Roland"));
```

Figure 22: JOOQ query

Database Connection Pool [4]**:** is a plug-in to optimize database connections for GPPLs like Java or C#. The goal is to make managing database connections easier and to make connections reusable. Setting up a new connection takes more time, compared to using an existing database connection instead. For each programming language there exist an implementation of this connection pool, but one of the challenges is managing the connection pool. Developers could close the wrong connection or reuse the wrong connection. Developers may also dislike the idea of having another component between their applications and databases.

4.9 Comparison of solutions

There exists a correlation between the concerns and the existing work and we decided to summarize it into a table, shown in Figure 23.

Issues	Relational database programming concerns					SQL concerns		
Solutions	1. Safety concerns	2.Type and value mismatches	3.Database connection handling	4.Database transactions	5.Constraint checking	6.Syntactic improvements	7. Functions and procedures	8. Relationships
OODBMS		✓						
ORDBMS		✓						
NoSQL								
Microsoft LINQ	✓	✓				✓		
Microsoft Templates							✓	✓
Ferry	✓	✓				✓	✓	
Iris					✓	✓	✓	
Hibernate		✓						
JOOQ	✓	✓				✓		
Database Connection Pooling			✓					

Figure 23: Relations between issues and solutions

The table shows four results:

- Impedance mismatch is the most solved concern
- None of the solutions solve all concerns
- Database management system optimizations are the least targeted issues to solve
- Database transaction optimization and table abstractions have not been solved by the compared solutions

We aim to solve all concerns to make our solution unique compared to the existing work. But it is not possible to solve all concerns within the given time for this thesis. The open issues are discussed in section 7.1.

4.10 Conclusion

We summarized and compared 10 different solutions of different kinds. Each of these solutions provide concepts to make database programming easier by solving impedance mismatch concerns or providing new ways to program databases. Various kinds of databases have been developed, such as the Object Oriented database management system (OODBMS), which stores information in a network of objects and introduces Object Oriented features such as inheritance and polymorphism. But companies haven't succeeded in making a competitive OODBMS in terms of performance and scalability. The Object-Relational database management system is a relational database management system which introduces object oriented features to the relational data model. Mapping objects to relational information is carried out at the database side. Ferry's approach has shown that it is possible to map GPPL code to SQL, which shortens the syntax and makes database programming easier for software engineers. The last selection of solutions, middleware, has shown solutions which are dependent and independent from the GPPL. Hibernate and LINQ are bound to a specific programming language, which makes the solution less portable when developers decide to change from GPPL. Therefore it would be better to provide a language which is independent from any GPPL to make it portable and usable for any GPPL of choice.

From the comparison of solutions and the concerns they solve, we can conclude that none of those solutions solve all of the concerns. Using multiple solutions would come close to solving the whole set, but also increases the development complexity. Instead it would be better to provide a new database language as a total solution. But some concerns are better to solve at architectural level. For example, mixing up the database programming language with the GPPL can result in safety and data type concerns. There exists solutions to solve these concerns like Microsoft LINQ or Hibernate, but those solutions are bound to a specific programming language. Instead we prefer to separate the database language from the GPPL to provide more freedom in which programming language to use. For this reason we propose to design an architecture for the language, before designing a new database language. The architecture and the new database language are described in the upcoming sections.

5 SQLento's architecture

The previous chapters have shown possible ways to improve database programming and various solutions to solve them. But not all concerns can be solved by using one solution and some concerns are easier to solve by using architectural solutions. This chapter describes the concepts for the architecture, the implementation and some of the possible alternative solutions.

5.1 The concept

We propose an architectural solution to place the new database programming language (SQLento) between the GPPL and the DBMS. In this way we can separate the new database language from any GPPL or DBMS and we can optimize our own language for relational databases mainly. The architectural solution consists out of compilers, communication interfaces for the GPPL and the DBMS and a definition section, as shown in Figure 24. The communication interfaces take care of communication between the GPPL, the architecture and the DBMS. The compilers are required to map the new database language to executable database code, like SQL. The definitions component keeps track of the executable database code. There exist many implementations for this architecture, where a few important ones are discussed in the next section.

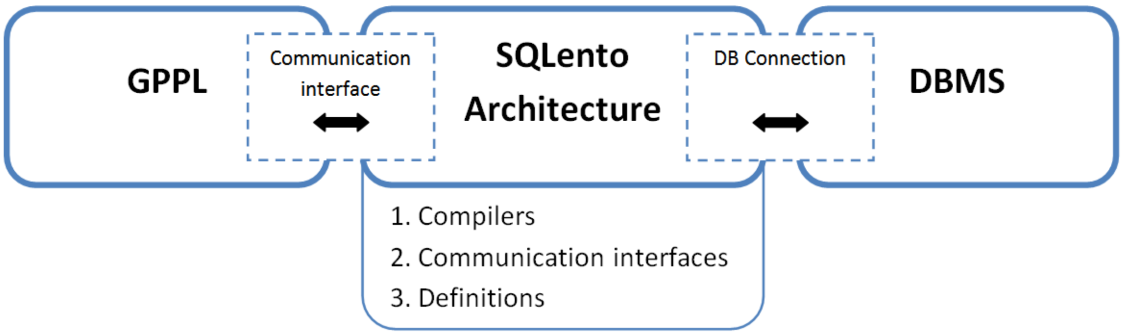


Figure 24: Concept of SQLento

5.2 Possible implementations

Existing work from section 4 can be combined to provide this architectural solution, as shown in Figure 25, to make database programming easier for relational databases. The first architecture shown is an application which uses an object-relational mapping solution, like Hibernate, to communicate with the database. This solution limits itself in portability, because the ORM solutions have been built for specific GPPLs. The second solution shows a possible way to separate the GPPL from the ORM by using a web service interface. Any kind of GPPL or application can communicate via the web service, the web service can call any kind of database code to execute and the ORM maps objects to the relational model and otherwise around. This separates the GPPL from the database language and allows developers to solve data model mismatches. But the developer is still responsible for the logic behind it, which we aim to provide as a solution. The architecture is also able to optimize SQLento code for optimal database transactions.

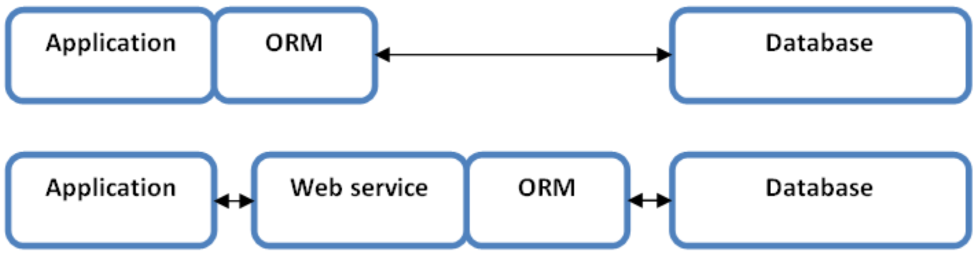


Figure 25: Possible architectures

Instead of using an architectural solution, Object-relational DBMSs could be used as well. But the ORDBMS extends the RDBMS with object-oriented features and therefore could be supported at language level (SQLento).

5.3 Proposed implementation

We propose to make the database language (SQLento) independent from the GPPL by providing a web service interface for any kind of GPPL. A program written in SQLento is compiled to executable database code, like SQL, and a communication interface for the GPPL (see Figure 26). By generating the web service interface for the GPPL, we can provide any kind of implementation for this communication interface, like Remote Procedure Calls, and we can optimize the database code for database transactions. The generated communication (web service) interface contains the logic to connect with the architecture and methods to call atomic database transactions, like user defined functions, which are written in SQLento.

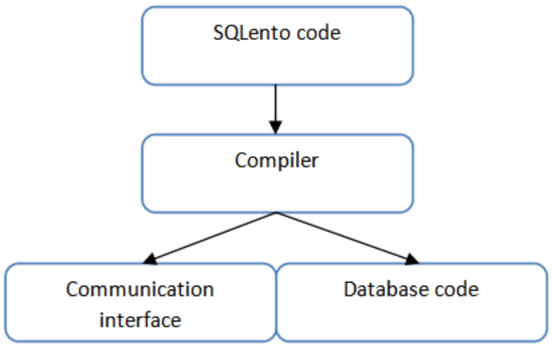


Figure 26: SQLento's compilation products

GPPLs use the generated communication (see Figure 27) interface to call executable database code, like database functions and procedures. When the web service component of the architecture receives these calls, the received information is checked in the definition component to decide which database code to execute and is mapped to executable database code. By using the database connection component the database code is executed on the DBMS. Results from the DBMS are communicated back via the architecture to the GPPL and can be mapped to a nested data model, just like an object-relational mapper, to prevent any data model mismatches. The architecture should be

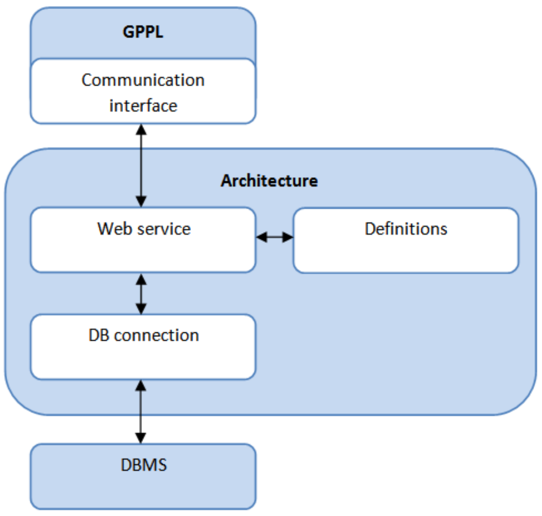


Figure 27: Communication flow diagram

the leading component to prevent unwanted data changes by other kinds of applications. Else tables can get modified by other applications and causing SQLento programs to fail.

5.4 Solving architectural concerns

Most of the concerns can be solved by providing a new database language. For example, the syntax can be simplified and features can be added to prevent duplicate join conditions. But concerns related to mixing up the GPPL with databases are easier to solve by providing an architectural solution. Differences between the used paradigm and data model makes combining database languages with GPPLs a more complicated process. Existing solutions like Microsoft LINQ solve this by embedding the declarative language within the GPPL, which makes the solution bound to the supported GPPL's. Instead of mixing the languages, the architecture separates the database language and provides an alternative communication interface for the GPPL, which is generated from the SQLento code. The differences in data models have to be solved by providing a compiler for mapping a nested structure to a relational structure and otherwise around.

Details about communication and compilers, like error reporting and the concepts, are discussed in the following sections. Error handling is explained for each of these architectural concepts, since they all have to solve specific error cases.

5.5 Compilers

The architecture contains compilers to map the new database language to a specific database language. The number of compilers depends on how many languages are supported, which is currently limited to one. Currently only SQL and the procedural extension is supported. The compiler is responsible for error reporting, the compilation process and generating the communication (web service) interface for the GPPL. The architecture is not limited to one compiler, since more database languages should be supported in the future.

Compilation types

Code written in SQLento can be compiled to any database language, with the focus on relational databases, but the DBMS may not support every language component. For example, MySQL does not support all kinds of SQL constraints. To prevent linguistic differences the code can be interpreted instead of compiled. Therefore it is necessary to have source-to-source compilers and interpreters to generate the executable database code and to provide the (web service) communication interface for the GPPL.

Compilation process

The compiler uses multiple steps to be able to map SQLento code to any other database language, as shown in Figure 28.

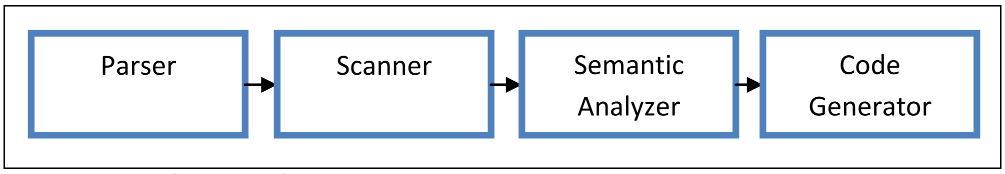


Figure 28: Compiling process layers

The parser generates a tree representation of SQLento's abstract syntax, also known as an abstract syntax tree. This process any syntactical mistakes are captured and reported to point out the cause and location of the problem. The scanner analyzes the abstract syntax tree and generates the environment, which maps SQLento information to a specific language. The semantic analyzer makes sure the correct information is provided by the code by inspecting the abstract syntax tree and using the environment. For instance, it may happen that an unknown table is used within a query, which has to be reported back as an error. Last but not least, the code generator provides the executable database code and the communication interface for the GPPL.

Error reporting

Two steps in the compilation process are responsible for providing error messages. The parser checks the syntax and the semantic analyzer evaluates if the correct information has been provided to generate the correct code. Syntax and semantic errors can be prevented and captured, but not all problems can be discovered at compile time because code can also fail at execution time. Those kind of problems are handled by the communication interface, because the compiler is only responsible for the mapping process and not the communication between the GPPL and the DBMS.

Synchronization feature

Databases may already contain tables and data, where the new database language (SQLento) has to know of. For example, SQLento has to know which tables exist to be able to develop further on. Therefore we introduce a feature to synchronize with existing data structures in the database and generate the database code from this information. This synchronization feature is optional because developers may want to override the existing state, like by providing new tables or data. Even though the synchronization feature is a nice attribute, it could fail on many different levels like missing specifications or not being able to retrieve information about the DBMS's tables. In case of failure, the language provides the option to specify the database manually.

Automatic transaction handling

The architecture handles database transactions automatically, so developers don't have to give the "commit" or "submit" signal in their code. We aim to make database programs atomic executable. The commit signal can be automatically added in procedures and functions, containing these database transaction statements. Still it requires some study in which cases this can be automated and which not. In some cases developers want to have control over the commit signals, which is also supported by SQLento.

5.6 Communication interface

The communication interface of the architecture consists of three parts: connections between the GPPL and the architecture, connections between the architecture and the DBMS and error handling.

Communication with the GPPL

General purpose programming languages such as Java or Python use the generated web service interface to call subroutines written in SQLento's code. Unlike Hibernate or LINQ, the solution is not bound to any GPPL because the interface can be generated for any specific GPPL. Subroutine calls are processed by the architecture's controller and definitions to check which database to address and which function or procedure to execute, as shown in Figure 29. Information retrieved from the DBMS can be transformed to a nested data structure, to solve the impedance mismatch for the information retrieval. But the nested data structure is optional to let the developer decide if this is required, because mapping the information costs time. Using this technology does have trade-offs, because it doesn't solve the connection handling concern. General purpose programming languages have to handle all used web service connections, instead of database connections. Another trade-off is that developers may have to use libraries or plug-ins to support this technology, which adds complexity to the GPPL.

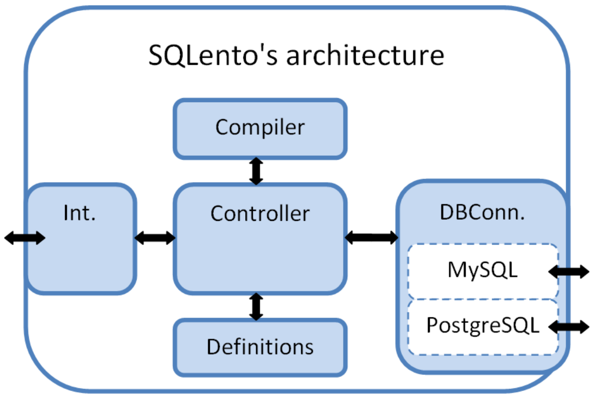


Figure 29: Exposed view of the architecture

Communication with the DBMS

The communication interface of the architecture takes care of connections with the DBMS. Since there exists various database management systems and even database programming languages, the architecture has to support many kinds of connections. When the architecture has received a database transaction, the compiled variant of the function or procedure is called and executed on the DBMS side. Information retrieved from the DBMS is passed back to the architecture.

Error handling

Communication can fail at any point between the GPPL, the architecture and the DBMS. There exist two kinds of problems which could occur: expected and unexpected problems. Expected problems can be prevented, like compilation errors, and require a notification of the problem. Unexpected errors are unpredictable like network failures or server failures and may require more actions to recover. In case of connection loss the architecture logs the problem to a file, sends a warning to the users and tries to recover the connection. If a transaction is in progress, the transaction is cancelled after recovery.

5.7 Advantages and disadvantages

The architecture we have provided has several advantages compared to other solutions:

- **Portability:** The architecture can be combined with any GPPL and DBMS of choice, because SQLento is separated from the GPPL and compiled to another executable database language. GPPLs can interact with database programs by using the generated web service interface.
- **Optimization:** SQLento's code can be optimized for database transactions because of its independence from the GPPL and the DBMS.

This makes the architecture more portable when developers want to change from programming language or connect new applications to it. Because SQLento is independent from the GPPL, the code can be optimized to lower the amount of database transactions. For example, a stored procedure can contain a whole number of statements and expressions to register a unique user, which can be combined in one transaction and compiled to a stored procedure on the DBMS.

Putting a web service architecture between the GPPL and the DBMS has consequences, especially when it provides a new database language. These disadvantages are:

- **Additional complexity:** Adding a new solution can be difficult, especially when it provides a new database programming language.
- **Performance:** Another layer is put between the GPPL and the DBMS which increases the latency.
- **Storage:** The architecture stores code written in SQLento and the compiled version of the code, which adds extra storage space on top of the GPPL and the DBMS solutions.
- **Optimized for relational databases:** the solution is optimized for relational databases, and may work sub-optimal for other solutions.

Even if the architecture has its pitfalls, it should solve some linguistic mismatches by separating the database language from the general purpose programming language. Issues like performance, storage and optimization could be explored in a later stadium, since there are more aspects to look at. For example, what kind of back-up capabilities could it provide?

5.8 Conclusion

We propose an web service architecture to make the new database programming language independent from the GPPL. The architecture is placed between the GPPL and DBMS, like a middleware system, and provides a new database programming language called SQLento. Code written in SQLento is compiled to executable database code and an web service interface for the GPPL. By generating the communication interface, we can provide any kind of web service implementation for specific GPPLs, like Java or Python, and we can optimize SQLento for database transactions. The GPPL uses this interface to communicate with the architecture, by calling subroutines written in SQLento. These execution calls are captured by the architecture and uses the definitions to evaluate which database code to execute, which has been compiled from the SQLento code. This piece of database code is then executed on the database via the database connection. For example, a function on the DBMS can be called from the architecture to get user information. The architecture retrieves the results, which can be mapped to a nested data structure. This nested data structure is optional, to let the developer decide to use it or not. It has implications on the time required to process the query, because the data has to be mapped.

The architecture consists of communication interfaces, compilers and definitions. The communication interface is generated by architecture's compiler, to allow communication between the GPPL and the architecture. The communication interface has also a component to connect itself to any DBMS, since the solution should not be limited to relational DBMSs only. Errors can occur expected and unexpected anywhere between the GPPL and the DBMS. The architecture should handle carefully and recover from any of these kind of failures. Errors are reported back in messages corresponding to SQLento or the architecture, to specify the error and the location of the problem. In the architecture there are multiple compilers present, which allows to map SQLento code to any kind of database programming language. The compiler process consists of several steps: parsing SQLento code to an abstract syntax tree, generating the environment, analyzing the semantics and generating executable database code. At the code generation part, the communication interface is generated as well. There are other solutions which could reach the same goal as SQLento's architecture, which is separating the database language from the GPPL language. A web service can be used to make database functionality executable without linguistic dependencies and an Object-Relational Mapper can be used to solve data model mismatches. But developers are still responsible for building the logic behind it, while the architecture we provide could even optimize SQLento code for database transactions. This makes our architecture more portable and optimized for RDBMSs. But there are also trade-offs to be found, like performance, additional complexity, additional data storage and it is only optimized for RDBMSs.

The architecture is still not complete because more concepts can be introduced and explained into more detail, like back-up capabilities, compiling methodologies and error recovery. If the whole architecture stops working for some reason, it should recover without harming the DBMS. How this could be done, has not been explored into detail yet. Only error reporting on local level at the compiler or the communication level has been described briefly. The missing topics are explained in the future work section (see section 7.1). The next section (chapter 6) describes SQLento, the new database programming language, in detail.

6 SQLento

The previous chapter described the architecture for SQLento, which solves concerns related to data model and linguistic mismatches. This section describes the language we have designed, to make database programming easier. It introduces two new concepts, to solve duplicate queries and to provide a nested view on queries. This chapter starts off with describing the concepts introduced by SQLento. The implementation of the prototype is described afterwards, by providing an example, the introduced features, the syntax and the semantics.

6.1 Concepts

SQLento introduces new concepts for programming databases to solve concerns of SQL's database language and the relational database management system. These concepts are:

1. Nested data structures
2. Relationship declarations
3. Difference between functions and procedures
4. Syntax simplifications
5. Automatic database synchronization feature

1. Nested data structures

SQLento provides a nested data model to match more closely with the object oriented data model. The nested data structure is mapped to the relational data model, which is comparable to the approach of object-relational databases and mapping solutions. Relationships between tables have to be specified as subfields to provide a nested overview of data, like pointers in an object oriented structure. This data model may constraint the query expressiveness since pointers are used to refer towards another object and can't be used to refer backwards. Relationships in a relational database are used to reason in both directions.

```
Users: {
        User: {
                Name: "Roland",
                Username: "RMB",
                Password: "123",
                Address: {
                        Street: "Somewhere
                        32",
                        Postal code: 1234,
                        City: "Enschede"
                }
                Contacts: { "Lesley" }
        }
        User: {
                Name: "Lesley",
                Username: "LSQL",
                Password: "456",
                Address: {
                        Street: "Somewhere 6",
                        Postal code: 3186,
                        City: "Enschede"
                }
                Contacts: ["Marieke","Roland"]
        }
}
```

Figure 30: Nested structure of users, addresses and contacts

Figure 30 shows an example of a nested data structure for representing a list of users, addresses and contacts. One of the possible implementations is the JavaScript Object Notation, which is supported by many programming languages like PHP or Python. But the consequences have to be studied further to know the pitfalls of this decision.

2. Relationship declarations

SQLento allows developers to specify relationships between tables just once, instead of repeating them in queries and statements, by adding the relationship as a field of a table. The compiler can fill in the missing join conditions as long as it knows the tables and the columns involved. To make this possible, relationships require an unique identifier. The identifiers of the relationships can be used in statements, like the insert, update and the remove statement, and queries. Extra information can be provided for removal procedures like the "delete on cascade" option to ensure all related rows are removed when the primary row is deleted. But this is not mandatory information to provide.

Figure 31 shows an example of defining an address subfield for a user by providing the join condition. The user's "userID" is joined with the Addresses.userID.

```
define users.address = (userId, Addresses.userId)
```

Figure 31: Defining a relationship

3. Difference between functions and procedures

Functions map input to output without causing side-effects, while procedures don't have to return output and may cause side-effects. Most of the procedural database extensions for SQL, like Microsoft Transact-SQL [29] and Oracle's PL/SQL, specifies these differences clearly. But not all DBMS's use the same extension and have differences in support. PostgreSQL does not support this distinction [43] and allows functions to cause side-effects like storing information into the database. The difference between a procedure and a function is kept in SQLento to prevent confusion.

4. Syntax simplification

The solutions Ferry and Iris, described in sections 4.6 and 4.7, have shown the possibilities of making database languages more comparable to general purpose programming languages. The independence from the GPPL allows us to optimize the language for databases and use the specifications of existing database languages. Figure 32 shows the declaration of a hash map in Java, a table in SQL and a possible solution. In Java it is not possible to declare the value constraints directly in the hash map, while SQL allows to specify data storage with constraints for each column. Our solution would be in between, declaring a table with constraints on the columns and using the Java-like syntax. This reduces the length of the code and the complexity.

Java	`HashMap<Integer,ArrayList> users = new HashMap<Integer,ArrayList>();`

SQL	`CREATE TABLE IF NOT EXISTS users (`
	`    userID serial primary key,`
	`    name varchar(255) NOT NULL,`
	`   username varchar(255) NOT NULL UNIQUE,`
	`   password varchar(255) NOT NULL`
	`)`

Possible solution	`Users = { userID Integer | name varchar(255) not null, username varchar(255) not null, password: varchar(255) not null)`

Figure 32: Imperative versus declarative way of declaring schemas

Most declarative database commands can be mapped directly to an equivalent in imperative code, but some statements and expressions may require more research. For instance, section

3.1.1 showed that loops in imperative GPPL's do not directly match and could result in non-optimal database operations. Instead we try to make the commands more comparable to GPPLs, by adjusting the syntax. For example, the nested data model is usable for queries and insert statements, as shown in Figure 33. The INSERT statement in SQL has to contain the "INSERT INTO" command, the table name and the values. The columns are not mandatory as long as the order of values matches with the column order of the table. The insert into command can be replaced by the table name and the "+=" symbol to specify the action. Since the column names are not mandatory, the information to be inserted can be put after the "+=" symbol in the correct order.

SQL	`INSERT INTO users (column1, column2, column n)`
	`VALUES`
	`       (value1, value2, …),`
	`       (value1, value2, …) ,...;`

Possible solution	`users += {`
	`    { value 1, value 2, value n},`
	`    { value 1, value 2, value n},`
	`    ...`
	`}`

Figure 33: SQL and JSON data declaration

5. Automatic database synchronization feature

SQLento provides a command to generate code for an existing database, which is part of the synchronization feature specified in section 5.5. The architecture is responsible for most of the synchronization process and creates the SQLento code from the specification. Developers can program further on the generated code from this synchronization feature. In case of failure, the document created from the existing database contains the error message and orders the developer to specify it manually.

6.2 Introduction of SQLento's prototype

The concepts introduced have been partially implemented for the prototype. The relationship declaration, syntax improvements and nested queries have been implemented. Nested queries are a new feature to provide a "nested" view on the data structure. But before diving into the syntax and semantics, we provide an example to get a first impression. SQLento's language has been designed as a procedural language on top of SQL and introduces syntax adjustments to make the language more comparable to GPPL's. This language allows you to write functions, procedures, table definitions and relation definitions. Within the functions, expressions can be used to map input to output values, similar to queries, simple equality checks and calculations. Within the body of procedures it is possible to use control flow statements like "if" structures, but it is also possible to adjust the table's state by adding new information, updating existing information or removing information.

```
users = { int id | string name, int age }
addresses = { int addrId | int userId, string street, int number, string city }
define users.address = (userId, Addresses.userId)

public function [string name, string city] getUserName(int userId) {
	Users { name, address.city } [id == userId]
}
```
Figure 34: SQLento program example

The example code shown in Figure 34 contains two table definitions, a relation definition and a function definition which contains a query. The "users" table has three columns called id, name and age. One of the columns is placed before the pipeline, which means it is a primary key. The "addresses" table uses the same structure to define its columns and primary keys. The relationship declaration is one of the introduced key features to prevent duplicate join conditions. Those can be used in queries and it allows to specify the relationship as a subfield of the "users" table. The next part of the code shows a function which looks similar to a method in Java, by providing the access modifier, the return type, the identifier and the parameters. Currently the information returned from functions and procedures is limited to tables and data types, which is a table format in this example. The body of the function contains a query to retrieve the user's name and the city of the address relationship when the parameter's value matches with the user's id. The city is called like a subfield of this relationship of the "users" table.

The program shown in Figure 35 represents a more complicated application for searching through users and addresses. It contains two user-defined functions to retrieve all users with addresses and to retrieve a user and address based on the searched ID number. The output of the compiler is shown afterwards as comparison. More examples can be found in the appendix in chapter 8.

SQLento

```
users = { int id | string name, int age }
addresses = { int uid | int postcode, int streetNumber }

define users.address (id, addresses.uid)

public function [string name, int postcode, int streetNumber]
getAllUsersWithAddress() {
	users {
		name,
		address {
			postcode,
			streetNumber
		}
	}
}

public function [string name, int postcode, int streetNumber]
getUsersAddress(int sid) {
	users {
		name,
		address {
			postcode,
			streetNumber
		}
	} [id == sid]
}
```

SQL

```
CREATE TABLE users ( id integer,name text, age integer, PRIMARY KEY
(id));
CREATE TABLE addresses ( uid integer,postcode integer, streetNumber
integer, PRIMARY KEY (uid));

ALTER TABLE addresses ADD CONSTRAINT addressesTousers FOREIGN KEY
(uid) REFERENCES users(id);

CREATE OR REPLACE FUNCTION getUserAddresses(integer) RETURNS  table
(name text, postcode integer, streetNumber integer) AS $$
BEGIN
RETURN   query
SELECT self.name,address.postcode,address.streetNumber FROM addresses
address,users self WHERE self.id = address.uid;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION getUserAddress(integer) RETURNS  table
(name text, postcode integer, streetNumber integer) AS $$
BEGIN
RETURN   query
SELECT self.name,address.postcode,address.streetNumber FROM addresses
address,users self WHERE self.id = $1 AND self.id = address.uid;
END;
$$ LANGUAGE plpgsql;
```
Figure 35: Example program for searching users and addresses

6.3 Relationship declarations

Relationships can be declared once after the table definitions and are reusable in queries. They contain the join information just as in SQL but with an additional feature. Relations are defined as fields of tables to provide a nested view of the tables and its relationships. The unique identifier of the relationship can be used in a selection and the projection of a query. The indirect result of the relation definition is allowing the compiler to add missing join conditions to the SQL queries. A direct result is generating the foreign key constraint to let SQL know about the relationship.

Relation definition	define Users.addresses = (userId, addresses.uid)	
BNF format	<relationDefinition> <innerJoin>	::= “define” <memberExpression> “(” <innerJoin> “)” ::= <identifier> “,”<memberExpr>

Figure 36: SQLento's syntax for specifying a relationship

Currently relationships are defined as shown in Figure 36. The relation definition requires the following information: the parent's table name, the parent's related column, the unique identifier of the relationship, the child's table name and the child's column name.

6.3.1 Foreign key generation

The information specified in the relationship declaration allows generating the foreign key, since it contains a relationship between columns of the referred tables. But it is still unknown which of the columns is the primary key, which is essential to know for generating the foreign key. For this reason it is required to define the tables before specifying the relationships, to make sure the environment contains the background information about the tables and their columns.

```
Users = { int userId | string name }
Addresses = { int addrId | int userId, string street, int number, string city }
Define Users.Address = (userId, Addresses.userId)
```
Figure 37: Declaring two tables and create a relationship in SQLento's language

Figure 37 shows an example of SQLento code to specify two tables and define a relationship as a subfield of the "users" table. The "Users" table contains a primary key called "userId" and a column called "name". The "addresses" table contains the "addrId" as primary key and has multiple columns to store other information. But it also contains the "userId" which should be a foreign key reference to the "users.userId" column. The relation definition afterwards specifies this relationship and allows to convert the information to the SQL code shown in Figure 38.

```
ALTER TABLE addresses ADD CONSTRAINT addressesTousers FOREIGN KEY (userId)
REFERENCES users(userId);
```
Figure 38: Generated the foreign key constraint in SQL

6.3.2 Join condition generation

The joins can be mapped by moving the join condition information to the SQL join format as shown in Figure 39. These relationships are stored in the environment under its identifier.

```
User.address = (userId, addresses.uid) ———————▶ Users.userId = addresses.uid
```
Figure 39: Mapping join informations to SQL format

This SQL format creates new concerns for queries when using self joins or relationships using the same table multiple times. In practice queries can contain references to the same table. For example, the user table may have a reference to itself to specify the friends of a user. Queries would end up ambiguous if columns would be retrieved from the same table, which is defined twice. SQL prevents these kind of concerns by allowing to add unique identifiers to each table [3]. Since we don't want to make our solution more complex, we have decided to use the unique relation identifier for referring to the related table in a relationship. Another option would be to generate the unique identifiers, which ensures the uniqueness of the relationship since the same relationship identifier could be used multiple times. But when a relationship identifier has been used for other relationships as well, we add a number to make it unique again. How these relationships are inserted, is specified in section 6.4.

6.3.3 Completeness

The SQL language allows to specify the foreign key more thoroughly in comparison to SQLento. For example, SQL allows to remove the row containing the primary key when a foreign key row with the same reference has been removed, by using the "delete on cascade" option. Another limitation is the use of the relationship declarations, since they can only be used for queries. It requires more research to specify which parts are actually missing from the language, to have an overview of the completeness of the relation definition.

6.4 Nested queries

Queries in SQLento consists of a projection and a selection. The projection of the query is specified differently compared to SQL. It uses the nested structure of columns and relationship declarations, as shown in Figure 40.

```
Users {
	name,
	Address {
		postalcode,
		street
	}
} [id == 1 & Address.postalcode > 2000]
```

Figure 40: Nested query on users

The query starts with the identifier of the table, which contains the relationship towards the "address" table. In the table the developer can access its columns and the relationships with other tables, as long as they have been declared as fields of this table. The query shown in the example accesses the "Users" table and retrieves the name of the person. It also accesses the "address" relationship as specified earlier in Figure 37. The "address" relationship is a subfield of the "users" table which is a join condition between the "users" and "addresses" table. From this joined table the postal code and street number is retrieved. This part of the code belongs to the "projection" of the query. After specifying what to retrieve, the developer can also provide the constraints or filters, which belongs to the "selection" of the query. Columns of the table started with and the relationships can be constrained, by providing the column name or relationship with column name and the filter. Currently it is not possible to address multiple columns under one relation identifier in the selection and it is required to put the relationship identifier in front of every column with constraint.

6.4.1 Limitations

Relations between objects in GPPL's like Java or C# work as one-directional relationships. When accessing a pointer from an object towards another object, it is not possible to use the same pointer to go back to the object started from. SQLento allows to use such relationship to refer backwards. The query shown in Figure 41 retrieves the name and address information from the Users and the Addresses table and constraints the related table. Therefore it retrieves the user information based on the address constraints. In SQL relationships can be used in both ways as well, as shown in the example too.

SQLento	`Users {`
	` name,`
	` Address {`
	`  postcode,`
	`  street`
	` }`
	`} [Address.addrId == 1]`

SQL	`SELECT u.name, a.postcode, a.street FROM users u, addresses a WHERE u.userId = a.userId AND a.addrId = 1`

Figure 41: Using a relation to refer backwards

This way of using relationships in queries does also have consequences. The query has to start with the table containing the relationship. Since the Users table has the relationship with Addresses, the query can't start with the Addresses table and then use the Address relationship to refer to the Users table. A solution is to define a new relationship on the Addresses table as shown in Figure 42, which adds a relationship from the Addresses to the Users table. But this solution uses the same join condition and adds complexity to the code.

```
Define Addresses.User = (userId, Users.userId)
```

Figure 42: Defining a new relationship between the Addresses and Users table

This example shows that there should be a solution for using relationships in both directions, while providing the nested data overview. More research is required to know the consequences. Since relationships can only be used forwards and backwards from the table it has been defined on, the query expressiveness should be limited as well.

6.4.2 Compiling a query to SQL

Compiling the nested query to SQL code is not as straightforward as functions or table declarations. Gathering the projection and selection is not the challenge, but iterating through the nested structure of columns and relationships can be. The "projection" tree has to be explored for relationships to generate the list of joins. The compilation process consists of three steps:

1. Collect columns from projection
2. Collect all filters from selection
3. Analyze for relationships and add missing join conditions
4. Put information into a template

These steps are shown in Figure 43, which represents pseudo code for the implementation of the algorithm.

```
Public String buildQuery ( String tableName, Selection selection, Projection
projection, Environment environment, String localScope ) {

       val sqlProjection = ""
       foreach ( value <- projection) do
              sqlProjection += flattenProjection(value);

       val sqlSelection = ""
       foreach ( relationalOp <- relationalOperations)  do
              sqlSelection += flattenSelection(relationalOp)

       val tables[] = tableName
       foreach (value <- projection) do
              relationship = analyzeRelationships(value, environment)
              tables += relationship.tables
              sqlSelection += relationship.joins

       if (sqlSelection.empty)
              return "SELECT" + sqlProjection + "FROM" + tables
       else
              return "SELECT" + sqlProjection + "FROM" + tables + "WHERE" +
              sqlSelection
}
```

Figure 43: Implementation for building a flattened SQL query

1. Collect columns from projection
The projection can contain two types of expressions (see section 6.5 for complete syntax): the value expression and a column list expression. The value expression can differ from an integer value up to column names as identifier expression. The column list expression is introduced to allow retrieving multiple columns from a relationship. The example in Figure 41 unfolds the "Address" relationship to retrieve multiple columns. The function "flattenProjection" analyzes each list of columns and flattens it to the SQL structure, by making a list of the columns. When a relationship identifier with one or more columns is found, each column gets the unique relationship identifier in front of the column name to refer to the right table, as shown in Figure 44.

SQLento	`Users {`
	`       Address {`
	`              number,`
	`              street`
	`       }`
	`} [ userId > 10 & Address.number > 20 ]`

SQL	`SELECT Address.street, Address.number FROM users u, addresses Address WHERE u.userId = Address.userId AND u.userId > 10 AND Address.number > 20`

Figure 44: Result of flattening the projection

2. Collect filters from selection
The "flattenSelection" function performs almost the same operations on the selection compared to the "flattenProjection", since it flattens the nested data structure to a flat SQL data structure. But the "selection" contains constraints on columns, which may contain a reference to a column or a relationship as well. This allows developers to specify constraints on the related table. The example shown in Figure 44 has two filters to constraint the user ID and the Address relationship's street number, where the "Address.userID" is a relationship. Both filters are flattened to a SQL filter and the identifier for the relationship is replaced by the identifier of the related table. The results are collected and put directly into the SQL query.

3. Analyze relationships and collect joins
The most complicated step is analyzing the projections for any join conditions, by iterating through lists and sublists of values, columns and column lists. The "analyzeRelationships" function keeps track of the list of tables and join conditions by comparing each identifier with the information stored in the environment. If a relationship is found, the join conditions are retrieved and the involved tables are stored in another list. Since join conditions can be chained, each join condition has to be analyzed for matching tables. For example, what if the user table has a relation with a table called "friends" and the friends table has also a relationship with the address table, as shown in Figure 45. This means that the join condition from users to friends and from friends to addresses are linked to each other. By finding the matching tables, each join condition can be converted to a correct format. The function returns after searching through the selection list the tables and join conditions as a tuple.

SQLento	users { friends { address { * } } } }
SQL	SELECT * FROM Users usr, Friend friends, Addressess address WHERE users.userId = friends.userId AND friends.userId = address.userId

Figure 45: Example of chained join conditions

4. Putting the information into the template
After collecting all the projection, selection, tables and joins, everything is inserted in the SQL template to build up the query. This template is returned as string format to generate one total list of SQL code.

6.4.3 Completeness

The nested queries are far from complete since it only supports the projection and selection of relational algebra to specify table expressions. In comparison to SQL the developer can only express the "SELECT", "FROM" and "WHERE of the query. Other statements like "HAVING" or "ORDER BY" are not supported yet. Another limitation are the relationship declarations, as they work as pointers from a table. Those pointers can be used refer forwards and backwards, as long as the query starts with the table that contains the relationship. Relationships in SQL are not bound to a table, which shows that SQLento is limited. More research is required to gain information about the expressiveness limitations.

6.5 Syntax

SQLento's database programming language shares syntax similarities with GPPLs, like how functions and procedures are defined. But it is still mostly based on SQL's specification, since it has been built on top of SQL and the PL/SQL extension. The prototype language developed for this thesis is explained further by using the Backus-Naur Form, which is a notation technique to describe the syntax.

Base program

```
<program>              ::= <transactionStatement>⁺
<transactionStatement> ::= <tableDefinition> | <functionDefinition> |
                           <procedureDefinition> | <relationDefinition>
```

The database program consists of a list of transaction statements. Currently SQLento supports four types of transaction statements: tables, functions, procedures and relationship declarations.

Definitions

```
<tableDefinition>      ::= <identifier> “ = ” <tableDeclaration>
<relationDefinition>   ::= “define” <memberExpression> “{” <innerJoin> “}”
<functionDefinition>   ::= <accessModifier> “function” <returnTypeDeclaration>
                           <identifier> “(” <parameter>* “) {” <expression> “}”
```

```
<procedureDefinition>  ::= <accessModifier> “procedure” <returnTypeDeclaration>
                           <identifier> “(”<parameter>* “)” “{” <statement>+ “}”
```

Definitions allow developers to specify the content of the program, which are the tables, relationships, functions and procedures. The "tableDefinition" requires table a declaration, by describing its columns and its properties. The "relationDefinition" is a newly introduced feature to describe relationships between tables as a subfield of another table. Further details about this feature can be found in section 6.3. The "functionDefinition" and "procedureDefinition" allows users to specify subroutines which may look equal at first sight. There are a few important differences:

- Functions do not support the "void" return type, captured by the compiler
- Functions identify themselves by the "function" keyword, while procedures use the "procedure" keyword in SQLento's language
- The body of a function can only contain expressions to return information, while procedures can contain statements and expressions

These design choices prevent functions from causing side-effects and maintains the clear difference between a function and a procedure. The keyword is used to make developers aware of the type of subroutine.

Declarations

```
<tableDeclaration>     ::= “ { ” <columnDeclaration>* “ | ” < columnDeclaration>* “ }”
<columnDeclaration>    ::= <columnTypeDeclaration> <identifier> “,”?
<parameterDeclaration> ::= <typeDeclaration> <identifier> “,”?
```

The "tableDeclaration" is used to specify the columns by using two lists. The first list contains zero or more columns with the primary key constraint and the second list contains zero or more columns without this constraint. At syntax level it is possible to declare a table without columns, but the compiler doesn't accept zero columns in total. But developers should be able to declare tables without primary key columns or tables containing only primary keys. The compiler checks if at least one column is present in the whole table declaration to prevent creating tables without columns at all. The "columnDeclaration" declaration type allows specifying the columns of a table by providing a column type declaration and an identifier. Last but not least is the "parameterDeclaration" for describing parameters in user-defined functions and procedures.

Type Declarations

```
<typeDeclaration>      ::= ( “String” | “Int” | “Float” | “Double” | “Date” |
                           “Boolean” | “a”.. “z”+)
<columnTypeDeclaration>::= <typeDeclaration> (“ ( ” <intLiteral> “ ) ”)?
<returnTypeDeclaration>::= “void” | “ [ ” <tableTypeDeclaration> “ ] ” |
                           <typeDeclaration>
<tableTypeDeclaration> ::= <columnDeclaration>+
```

SQLento supports four variations of the type declaration. The "typeDeclaration" variant encapsulates the most commonly used data types. The "columnTypeDeclaration" is used to specify columns and the maximum value length in a table declaration. Currently it is not possible to add more column specifications like the "not null" constraint. The return type declaration is used to specify the return type values and therefore introduces the void return type and the table type. At parser level it is ensured that functions do not support the "void" return type. The last declaration, "tableTypeDeclaration", is used when information has to be returned in a table format, which occurs often when a query returns multiple rows and columns of information. Developers have to provide the column types and names to return information in this table format.

Expressions

```
<relationalOperatorExpr>      ::= (<literalValueExpression> ( “==” | “!=” | “>=” |
                              “<=” | “>” | “<” ) <expression>)
<existsSubquery>              ::= <selectExpression> (“.exists” | “.EXISTS”)
<functionCallExpression>      ::= <identifier> “(” <literalValueExpression>* “)”
<operatorExpression>          ::= <literalValueExpression> ( “+” | “-” | “*” | “/” )
                              <expression>
<selectExpression >           ::= <identifier> “ {” <projection> “} ”
                              “[” <selection> “]”)?
<valueExpr>                   ::= <valueExpression>
```

Expressions can be used within the body of a function or certain statements like conditionals. The "relationalOperatorExpression" allows to evaluate the equality of values. It requires an literal value expression, an operator and an expression. Currently only the right-hand side supports expressions to prevent infinite loops in the compiler. The "subQuery" expression allows to check whether a value from a query is returned or not. It contains a "selectExpression" to specify the query it is applied on. The "functionCallExpression" allows developers to specify a function call by providing the identifier and a set of arguments for each parameter. The "operatorExpression" allows to write mathematical expressions like multiplications or subtractions. Still the number of mathematical operators supported is limited to adding, subtracting, multiplying and dividing. The "selectExpression" is one of the most important expressions because it specifies how to create queries. It is only possible to provide the selection and projection in relational algebra format. Ordering, limitations and other ways to influence the query results are not supported yet. Details about queries are discussed further in section 6.4. The last expression is the "valueExpression", which allows evaluating literal values or variable references.

Literal Value Expressions

```
<valueExpression>       ::= <stringValue> | <intValue> | <floatValue> | <doubleValue> |
                        <dateValue> | <booleanValue> | <identifierExpression> |
                        <memberExpression> | <columnListExpression>
<memberExpression>      ::= <identifierExpression> “.” <identifierExpression>
<identifierExpression> ::= <stringValue>
<stringExpression>      ::= “a”.. “z”+
<floatExpression>       ::= “-”? (0..9)+ “.” (0..9)* “f”
<doubleExpression>      ::= “-”? (0..9)+ “ . ” (0..9)+
<intExpression>         ::= “-”? (0..9)+
<dateValue>             ::= (0..9)+ “ - ” (0..9)+ “ - ” (0..9)[4]
```

```
<booleanValue>          ::= (“true” | “TRUE”|“false” | “FALSE”)
<asteriskExpression>    ::= “*”
```

The "valueExpressions" contain the most used value types and value references used in SQLento. It does not support the complete set of data types, like user defined types.

Statements

```
<ifStatement>           ::= “if” “(” <expression> “)” <statement> (“else” <statement>)?
<blockStatement>        ::= “ { ” <statement>+ “ } ”
<returnStatement>       ::= “return” <expression>
<callStatement>         ::= <identifier> “(” (<literalValueExpression> “,”?)* “)”
<tableStatement>        ::= <insertStatement> | <updateStatement> | <removeStatement>
```

Statements are used in procedures since they do not evaluate to a value and can cause side-effects. The "ifStatement" allows to influence the execution flow by expecting a boolean expression and a statement within the body. The "blockStatement" is used when multiple statements are used in the body of the "ifStatement". But the "ifStatement" does not support an "else if" statement with an expression. The "returnStatement" allows to return an expression in user-defined procedures and expects the "return" keyword and an expression. Last but not least is the "tableStatement" to express DML statements on tables. This can differ from inserting new rows, updating rows to removing rows from tables.

Table Statements

```
<insertStatement>       ::= <identifier> “ += { ” <stringValue>+ “}”
<updateStatement>       ::= <identifier> (“ [ ” <selection> “ ] ” )?  “ = { ”
                            <assignStatement>+ “ } ”
<removeStatement>       ::= <identifier> (“ [ ” < selection > “ ] ” )? “.drop”
```

Table statements are part of the DML because they provide ways to adjust the table's state. Developers can add information by using "insertStatement", which requires to know the table and the values of the insert procedure. Currently it is not possible to add multiple rows at once. Updating information in tables can be achieved when the information of the "updateStatement" has been provided. It requires the table name, the conditions and the assignment of values to columns. The conditions are used to describe when a row should be updated and uses the selection of the relational algebra format. Removing rows from a table is achieved by using the "removeStatement". It requires to specify under which criteria the row has to be removed, also by using a selection, and requires to put ".drop" at the end. For both the "updateStatement" and the "removeStatement" it is not required to provide the conditions for updating and removing the information.

Relational Algebra

```
<projection>            ::= (literalValueExpression “ , ”?)+
<selection>             ::= (<relationalOperatorExpr > “ , ”?)+
<innerJoin>             ::= <identifier> “,”<memberExpr>
```

Queries, table statements and relation definitions use relational algebra for their specification. Relational algebra can be mapped to SQL since the relational data model is based on this mathematical logic. For example, selections in relational algebra are used to obtain results based on criteria. SQL allows to specify these filters in the WHERE clause. SQLento supports currently three relational algebra components: projections, selections and inner joins. The projection contains a list of columns and values to be returned from the query. The selection contains the information for filtering results based on criteria, specified in "relationalOperatorExpressions". Relationship definitions use the inner join to specify the relationship between the table's own column and another table's column. The inner join makes sure that only results are obtained when there exist a match between the related columns. The inner join is used for the relationship declaration.

6.6 Translation

The semantics of SQLento's language is given by the compiler prototype, since the mapping process to SQL adds meaning to each part of the language. This compiler consists of a several steps: parsing SQLento code to an abstract syntax tree, analyzing the abstract syntax tree to build up the environment, checking the abstract syntax tree for problems and mapping the abstract syntax tree to SQL code. In the last step the information of the abstract syntax tree is put into a "template" of SQL code. This chapter describes the semantics of SQLento's code in a short way, by leaving out the implementation details of this compilation process and by describing the results.

Used definitions:

- `compile ( env:Environment, tr: List[TransactionStatement] ) → String`
- `compileTransactionSt ( env:Environment, tr:TransactionStatement ) → String`
- `compileDeclaration ( decl:Declaration ) → String`
- `compileExpression ( env:Environment, expr:Expression, localSc:LocalScope ) → String`
  `compileStatement ( env:Environment, st:Expression, localSc:LocalScope) → String`

Each of these compile functions map SQLento code to SQL code by analyzing the command and putting the SQL information into a string. Each category of the BNF grammar, e.g. statements, expressions, declarations, and etc, have their own compilation function to separate each concern. The "compile" function is the entry point for the compilation process, which requires the environment and the "to be processed" transaction statement. The "environment" is a function which maps SQLento information to SQL, like giving parameters a different symbol and naming convention for SQL. This information is useful for the mapping process to check whether an identifier is referring to a local accessible parameter or a table in the global accessible environment. The local scope tells the compiler in which part of the environment should be looked.

Transaction statements

- `compileTransactionSt ( env, tableDefinition ( identifier, tableDeclaration ) ) = "CREATE TABLE" + identifier +  "(" + compileDeclaration ( tableDeclaration ) + ")";`
- `compileTransactionSt ( env, relationDefinition ( identifier, innerJoin ) ) = compileRelationalOperation ( env, innerJoin, identifier )`
- `compileTransactionSt (env, functionDefinition (accessModifier, returnType, identifier, list[Parameter], expression))) = "CREATE OR REPLACE FUNCTION" + identifier + "(" + compileDeclaration (list[Parameter.type]) + ")" + "RETURNS" + compileDeclaration (returnType) + "AS $$ BEGIN" + compileExpression (env, expression, identifier) + "END; $$ LANGUAGE plpgsql;"`
- `compileTransactionSt (env, procedureDefinition (accessModifier, returnType, identifier, list[Parameter], list[Statement])) = "CREATE OR REPLACE FUNCTION" + identifier + "(" + compileDeclaration (list[Parameter.type]) + ")" + "RETURNS" + compileDeclaration (returnType) + "AS $$ BEGIN" + compileStatement (env, list[Statement], identifier) + "END; $$ LANGUAGE plpgsql;"`

The information in each type of transaction statement is added in a template of SQL code. The expressions, declarations and statements are analyzed further to generate specific SQL code. For example, each parameter of the procedure is analyzed further by using the "compileDeclaration" function call. For most of the definitions the mapping is straight forward, except for the "relationDefinition". This one maps straight away to a foreign key constraint. The "relationDefinition" is one of the key features introduced by this database language, which is explained further in section 6.3.

Declarations

- `compileDeclaration ( tableDeclaration ( list[PrimaryKeys], List[Columns] ) ) = compileDeclaration ( list[PrimaryKeys] ) + compileDeclaration ( List[Columns] ) + "," + "PRIMARY KEY" + "("compileDeclaration ( list[PrimaryKeys] ) + ")"`
- `compileDeclaration ( columnDeclaration (identifier, typeDeclaration) ) = identifier + compileDeclaration ( typeDeclaration )`
- `compileDeclaration ( parameterDeclaration ( typeDeclaration, identifier ) ) = compileDeclaration ( typeDeclaration )`
- `compileDeclaration ( tableTypeDeclaration ( list[columnDeclaration] ) ) = "table" + "(" + compileDeclaration ( list[columnDeclaration] ) + ")"`
- `compileDeclaration ( typeDeclaration ( type, length ) ) = type + "(" + length + ")"`

Mapping declarations result into a specification of tables, columns, value types and parameters. These are used for table definitions, function definitions, procedure definitions and relation definitions. All information can directly be placed into a template of SQL code.

Expressions

- `compileExpression ( env, relationalOperatorExpr ( leftExpr, operator, rightExpr ), localScope ) = compileExpression ( env, leftExpr, localScope ) + operator + compileExpression ( env, rightExpr, localScope )`

- `compileExpression ( env, existsSubquery ( queryExpression ), localScope )`
  `= EXISTS + "(" compileExpression ( env, queryExpression, localScope ) + ")"`
- `compileExpression ( env, functionCallExpr ( identifier, list[valueExpression] ) )`
  `= identifier + "(" compileExpression ( env, list[valueExpression], localScope ) + ")"`
- `compileExpression ( env, operatorExpression(left, operator, right ), localScope )`
  `= compileExpression ( leftExpr ) + operator + compileExpression ( rightExpr )`
- `compileExpression ( env, memberExpression ( parent, child ), localScope )`
  `= compileExpression ( parent ) + "." + compileExpression ( child )`
- `compileExpression (env, valueExpression, localScope )`
  `= value`
- `compileExpression ( env, queryExpression, localScope )`
  `= buildQuery ( env, queryExpression, localScope )`

Each expression is mapped to an equivalent in SQL where most can be directly mapped. There are two exceptions though, which are the "valueExpression" and the "queryExpression". The values have been simplified to make the documentation more readable. The queries are specified in detail in section 6.4, since they differ in syntax and semantics compared to SQL. The "valueExpression" is for almost every case trivial since the supported data types are based on SQL's specification only with a different naming convention. For example, a string in SQLento is equal to the text data type in SQL. But for one case the semantics has to be explained: identifiers. When the "valueExpression" contains an identifier to refer to a parameter, then the mapping process has to be adjusted. Parameters in SQLento's language have a name to be identifiable, while SQL uses the dollar sign and a number. For example, in SQL the reference to the first parameter would end up like "$1". Therefore each identifier should be looked at, by using the environment and the local scope to check if it is a parameter. If so, adjust the name to the dollar sign and the parameter number.

Statements

- `compileStatement ( env, ifStatement ( expression, statement, statement2 ), localScope )`
  `= "if" + compileExpression ( expression ) + compileStatement ( statement ) + "else" + compileStatement ( statement2 ) + "endif"`
- `compileStatement ( env, blockStatement ( list[statement] ), localScope )`
  `= compileStatement ( list[statement] )`
- `compileStatement ( env, CallStatement ( identifier, list[valueExpression] ), localScope )`
  `= identifier + "(" compileExpression ( env, list[valueExpression], localScope ) + ")"`
- `compileStatement ( env, returnStatement ( expression ), localScope )`
  `= "return" + compileExpression ( env, expression, localScope )`
- `compileStatement ( env, assignStatement ( identifier, expression ) , localScope )`
  `= identifier + "=" +  compileExpression ( env, expression, localScope )`
- `compileStatement ( env, insertStatement ( identifier, list[valueExpression] ) , localScope )`
  `= "INSERT INTO" + identifier + "VALUES" +  "(" compileExpression ( env, list[valueExpression], localScope ) + ")"`
- `compileStatement ( env, updateStatement ( identifier, selection( list[RelationalOperatorExpr] ), list[ValueExpression] ), localScope )`

```
  = "UPDATE" + identifier + "SET" +  compileStatement ( env, selection(
  list[RelationalOperatorExpr] ), localScope ) + "WHERE" + compileRelationOp (
  env, selection, localScope )
▪ compileStatement ( env, deleteStatement ( identifier, selection (
  list[RelationalOperatorExpr] ) ), localScope )
  = "DELETE FROM" + identifier + "WHERE" +  compileStatement ( env,
  list[ValueExpression], localScope ) + "WHERE" + compileRelationOp ( env,
  selection, localScope )
```

SQLento supports a set of the most commonly used statements, like control flow statements and DML statements. Each of these statements just require placing the information on the correct position in the SQL template, since they do not differ in semantics from the SQL specification except for the completeness of the language.

Relational operations

```
▪ compileRelationalOp ( env, projection ( list[ValueExpression] ), localScope )
  = compileExpression ( env, projection, localScope )
▪ compileRelationalOp ( env, selection ( list[RelationalOperatorExpr]),
  localScope )
  = compileExpression ( env, selection, localScope )
▪ compileRelationalOp (env, innerJoin (Identidier, MemberExpression) ,
  Identifier)
  = ALTER TABLE foreignKey.table ADD CONSTRAINT identifier
  FOREIGN KEY foreignKey.column REFERENCES primaryKey.table ( primaryKey.column
  );
```

The relational operations can be mapped to the SQL equivalent. The projection matches with the SELECT statement of SQL, because both declare which columns of information to retrieve. The selection contains the list of filters or constraints, comparable to the "WHERE" statement in SQL. The inner join is an exceptional case since it is mapped directly to a foreign key constraint. The join conditions for reuse in queries and statements have been stored in the environment to be of use (see section 6.3).

6.7 Advantages and disadvantages

SQLento provides the following advantages:

- Relationships can be declared once.
- Relationships used as a nested data structure in queries.
- SQLento programs can be compiled to SQL.
- The simplified syntax is more comparable to GPPL languages.

The disadvantages are:

- Relationship declarations can result in a complex network of nested tables.
- Joins can't be modified within the query.
- Suboptimal queries can be generated if the same relationship structure is defined with different names.
- Queries are limited by the relationship declarations.

The introduced features does make database programming easier, since the syntax has been shortened, relationships can be declared once and the solution can be compiled to SQL. But one of the main trade-offs is the reduced flexibility, since developers can't adjust join conditions directly in a query, if required. It is also possible to create complex networks of nested (joined) tables and therefore create suboptimal queries. It may be possible that the same path is used but with different relationship identifiers.

6.8 Conclusion

SQLento provides a new database language to express transaction statements, definitions, declarations, expressions and statements. It is build on top of SQL and uses Oracle's SQL specification for the mapping process, to provide equal semantics with a different syntax. The syntax of SQLento is comparable to GPPL by using similar ways to write subroutines and brackets. One of the key features of SQLento's language is providing a solution to prevent duplicate join conditions. Relationships are declared after the table definitions and can be used in queries. Currently it is not possible to use relationships for insert, update and remove statements. Queries in SQLento use the relationships to provide a nested overview of the relationships between tables, which can be used as a bi-directional pointer as long as the parent table is used first. Using relations as object pointers does influence the query capability, because the relationship is bound to one table instead of both related tables. How this influences the queries in expressiveness is still unknown and is recommended as future work. There may exist solutions to solve this concern. In overall SQLento has a solution to declare join conditions once, to provide a nested overview of data in queries and to shorten the syntax. But queries are limited by the relationship declarations, joins can't be modified in queries and complex networks of nested tables can result in suboptimal queries.

In sense of completeness, there is much work to do. For example, only one concern has been solved by the prototype language and the nested queries brought the data model goal closer to the Object oriented data model. This leaves many concerns open to solve and the language is by far not SQL complete. For example, the queries do not support ordering yet. From this result we can conclude that SQLento is still a prototype or proof-of-concept, with the potential to grow further to a complete solution.

7 Conclusion

Our goal was to make database programming easier by providing a new database programming language. Since we encountered mismatches between the GPPL and database programming, we decided to gather a list of concerns from different sources, like papers and websites. Each of the found concerns are not very crucial to solve and the existing work to solve them are only partial solutions. Therefore we aimed to design a solution to solve all concern. This goal has been partially achieved, with some slight changes. We noticed that some of the concerns are easier to solve by providing an architecture, instead of solving them by providing a new database programming language. Therefore we had to adjust our plan to design an architecture first. One of the key features is the independence of the architecture from the general purpose programming language and the DBMS. This allows to generate a web service interface based on the code written in the new database language. This web service interface can be generated for any GPPL to make sure communication is possible. For this reason SQLento's database code does not have to be mixed up with the GPPL. This makes the solution more portable. The architecture also provides a solution to map the object oriented data model to the relational data model to prevent data model mismatches, but this is left optional to let developers decide about his solution. Using this architecture has its trade-offs. The latency may be a concern because an extra layer is added between the GPPL and the DBMS, but the performance impact is still unknown and requires research. The solution also adds complexity since it provides a new database programming language and has to be installed as a middleware solution.

SQLento is one of the main contributions, which is the new database programming language. Developers can write function, procedure, table and relation definitions in SQLento, which can contain a limited set of statements and expressions. It introduces concepts such as a nested data structure, relationship declarations and syntax simplifications. The actual prototype language is a subset of the concepts and introduces the relationship declarations, syntax simplifications and nested queries. Programs written in SQLento are mapped to executable SQL code and contains mostly syntactical differences, except for two features. We introduce a new way to declare relationships and a way to write queries. Relationships are declared after defining the tables and result in a foreign key constraint. The information about the relationship is stored into the memory to be used in queries and statements. When a query contains the reference to the relationship, the join condition is inserted automatically. The nested queries provide a way to write the list of columns and relationships as a nested structure. For example, if there exists a relationship between the users and the addresses tables, it is possible to access the addresses from the users as a field. The disadvantage of this solution is that relationships are declared from a table to another table. Therefore it is possible to use relationships to reason forwards and backwards, as long as the query starts with the table containing the relationship. This solution limits the expressiveness of queries because this problem does not occur in SQL. This concern is added to the future work, since there was no time left to get this problem fixed. Another disadvantage of SQLento's solution is that complex networks of nested tables can be created, which may result in suboptimal queries. It is currently possible to add duplicate join conditions with a different name and therefore create suboptimal queries. This could make a complex network of nested tables more difficult to manage.

Finally, all contributions have been achieved. But there is still much work to do because SQLento's language is not SQL complete, queries are limited, consequences of this solution are still not fully explored and there are more concerns to solve. The next section describes a few possible topics to continue on.

7.1 Future work

In this section we discuss which directions to continue on from this research.

Exploration of concerns: The list of concerns provided in section 3.2 are just a subset of the possible improvements, because the impedance mismatch hasn't been covered fully. There may be even more problems caused by this mismatch than we have noticed. Also the database management systems have more concerns than specified, like the difference in SQL support.

Exploration of solutions: The evaluation of existing work in section 4 is still limited. There exist more solutions to solve the found concerns, which can be compared. For example, SQL++ [15] has not been checked yet. SQL++ is a middleware solution for merging SQL with JSON. More papers could also be checked to gain knowledge about concepts and related work, like the article called "Representing relationships as first-class citizens in an Object-oriented programming language" [54].

Architecture: There are a few topics which could be explored further:

- In what kind of alternative ways can a GPPL connect with the architecture?
- What are its back-up capabilities?
- What kind of compilation methods are available to take care of differences in database languages?
- What is the performance impact of this solution?

Since the list of concerns and solutions are not complete yet, the concept may require to be adjusted if new concepts are introduced.

SQLento: The prototype of SQLento requires a lot of work. The language is not SQL complete, it introduces two features to solve the duplicate join condition concern and queries are currently being limited by the relation declarations. Each of these directions do require attention to make SQLento and the concept more complete, since it targets to be SQL complete and to solve all concerns specified in section 3.2.

8 Appendix

8.1 Code example 1

SQLento

```
users = { int id | string name }
friends = { int x, int y }

define users.userFriends (id, friends.x)
define friends.friendsUsers (y, users.id)

public function [string name, int x, int y, string friendsName]
getFriends() {
       users {
              name,
              userFriends {
                     x,
                     y,
                     friendsUsers {
                            name
                     }
              }
       }[id == 1]
}
```

SQL

```
CREATE TABLE users ( id integer,name text, PRIMARY KEY (id));
CREATE TABLE friends ( x integer, y integer);
ALTER TABLE friends ADD CONSTRAINT friendsTousers FOREIGN KEY (x)
REFERENCES users(id);

ALTER TABLE friends ADD CONSTRAINT friendsTousers FOREIGN KEY (y)
REFERENCES users(id);

CREATE OR REPLACE FUNCTION getFriends() RETURNS  table (name text, x
integer, y integer, friendsName text) AS $$
BEGIN
RETURN   query
SELECT self.name,userFriends.x,userFriends.y,friendsUsers.name FROM
users friendsUsers,friends userFriends,users self WHERE self.id = 1
AND self.id = userFriends.x AND userFriends.y = friendsUsers.id;
END;
$$ LANGUAGE plpgsql;
```

8.2 Code example 2

SQLento

```
users = { int uid | string name, int age}
userAddr = {int uid, int aid }
addresses = { int aid | string street, int streetNumber }

define users.addrRelation (uid, userAddr.uid)
define userAddr.addr (aid, addresses.aid)


public function [int id,string name,int age] getAllUserDetails() {
       users{*}
}

public procedure void insertUser(int id, string name, int age) {
       if (userExists(name,age) == false)
              users += { id, name, age }
}

public function boolean userExists(string searchedName, int
searchedAge) {
        users{1}[name == searchedName & age == searchedAge].exists
}

public function [string name,string address] getUserAddresses(int sid)
{
        users{
                name,
                addrRelation {
                        addr {
                                streetNumber
                        }
                }
        } [uid == sid]
}

public procedure void updateUsername(int sId, string newName) {
        users[id == sId] = { name = newName }
}

public procedure void removeUser(int sId) {
        users[id == sId].drop
}
```

SQL

```
CREATE TABLE users (uid integer,name text, age integer, PRIMARY KEY
(uid));
CREATE TABLE userAddr ( uid integer, aid integer);
CREATE TABLE addresses ( aid integer,street text, streetNumber
integer, PRIMARY KEY (aid));

ALTER TABLE userAddr ADD CONSTRAINT userAddrTousers FOREIGN KEY (uid)
REFERENCES users(uid);

ALTER TABLE userAddr ADD CONSTRAINT userAddrToaddresses FOREIGN KEY
(aid) REFERENCES addresses(aid);

CREATE OR REPLACE FUNCTION getAllUserDetails() RETURNS  table (id
integer, name text, age integer) AS $$
BEGIN
RETURN query
SELECT * FROM users self;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION insertUser(integer, text, integer) RETURNS
void AS $$
BEGIN
IF userExists($2, $3 ) = false THEN
INSERT INTO users VALUES ($1, $2, $3);
END IF;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION userExists(text, integer) RETURNS  boolean
AS $$
BEGIN
RETURN  EXISTS(
SELECT 1 FROM users self WHERE self.name = $1 AND self.age = $2);
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION getUserAddresses(integer) RETURNS  table
(name text, address text) AS $$
BEGIN
RETURN   query
SELECT self.name,addr.streetNumber FROM addresses addr,userAddr
addrRelation,users self WHERE self.uid = $1 AND self.uid =
addrRelation.uid AND addrRelation.aid = addr.aid;
```

```
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION updateUsername(integer, text) RETURNS  void
AS $$
BEGIN
UPDATE users SET name = $2 WHERE id = $1;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION removeUser(integer) RETURNS  void AS $$
BEGIN
DELETE FROM users WHERE id = $1;
END;
$$ LANGUAGE plpgsql;
```

9 Bibliography

[1] A. Formica, H. D. Groger and M. Missikoff, “An efficient method for checking object-oriented database schema correctness”, *ACM Transactions on Database Systems (TODS),* 1998.

[2] A. Maatuk, M. A. Ali and N. Rossiter, “Re-engineering relational databases: the way forward”, *ISWSA '11,* 2011.

[3] Apache, “HBase Acid Semantics”, 27-04-2015. Available: http://hbase.apache.org/acid-semantics.html. Accessed: 26-05-2015.

[4] Apache, “The DBCP component”, Apache, 24-02-2015. Available: http://commons.apache.org/proper/commons-dbcp. Accessed: 16-09-2015.

[5] BaseX Team, “BaseX”, 28-03-2015. Available: http://basex.org/. Accessed: 19-05-2015.

[6] D. G. Murray, M. Isard and Y. Yu, “Steno: Automatic Optimization of Declarative Queries”, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation,* 2011.

[7] D. Greekery, “JOOQ and Hibernate, a discussion”, 21-04-2012. Available: http://blog.jooq.org/2012/04/21/jooq-and-hibernate-a-discussion. Accessed: 19-05-2015.

[8] D. Thakur, “What is Object-Relational Database Systems? Advantages and Disadvantages of ORDBMSS.”. Available: http://ecomputernotes.com/database-system/adv-database/object-relational-database-systems.

[9] Data Greekery, “JOOQ”, 2015. Available: http://www.jooq.org/#a=usp-all. Accessed: 18-05-2015.

[10] DB-Engines, “DB-Engines ranking”. Available: http://db-engines.com/en/ranking. Accessed: 08-05-2015.

[11] Django Software Foundation, “Documentation”, 2015. Available: https://docs.djangoproject.com/en/1.9/faq/models/#does-django-support-nosql-databases. Accessed: 08-12-2015.

[12] E. F. Codd, “A relational model of data for large shared data banks”, *IBM Research Laboratory,* 1970.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, “Bigtable: A Distributed Storage System for Structured Data”, *OSDI 2006,* 2006.

[14] F. Hajdu, “How to report an error from a SQL Server user-defined function”, Stackoverflow, 05-01-2010. Available: http://stackoverflow.com/questions/2005453/why-is-try-catch-block-not-allowed-inside-udfs. Accessed: 05-11-2015.

[15] Forward, “SQL++,” 29-04-2015. Available: http://forward.ucsd.edu/sqlpp.html. Accessed: 27-11-2015.

[16] IBM Knowledge center, “Example of a simple JDBC application”. Available: https://www-01.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/com.ibm.db2z11.doc.java/src/tpc/imjcc_cjvjdbas.dita. Accessed: 13-11-2015.

[17] J. Annevelink, “Database programming languages: a functional approach”, *Proceedings of the 1991 ACM SIGMOD international conference on Management of data,* 1991.

[18] J. Fong, “Converting relational to object-oriented databases”, *ACM SIGMOD,* 1997.

[19] J. Gil and K. Lenz, “Eliminating impedance mismatch in C++”, *VLDB '07 Proceedings of the 33rd international conference on Very large data bases,* 2007.

[20] K. Zeidenstein, “DB2's object-relational highlights: User-defined structured types and object views in DB2”, IBM, 11-10-2001. Available: http://www.ibm.com/developerworks/data/library/techarticle/zeidenstein/0108zeidenstein.html. Accessed: 12-11-2015.

[21] L. Wevers, “A persistent functional language for concurrent transaction processing”, Final thesis, University of Twente, 24-08-2012.

[22] M. Angelov, “Making A Cool Login System With PHP, MySQL & jQuery”, TutorialZine, 17-10-2009. Available: http://tutorialzine.com/2009/10/cool-login-system-php-jquery. Accessed: 06-10-2015.

[23] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich and S. Zdonik, “The object-oriented database system manifesto”, Morgan Kaufmann Publishers Inc., 1992.

[24] M. Kifer, A. Bernstein and P. M. Lewis, “Object-Relational Databases”, in *Database systems: An Application Oriented Approach*, Pearson, 2005, pp. 14.1 - 14.3.

[25] M. Stonebraker, “SQL databases v. NoSQL databases”, *Communications of the ACM,* no. Volume 53 Issue 4, 2010.

[26] M. Wang, “Implementation of object-relational DBMSs in a relational database course”, *SIGCSE '01 Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education,* 2001.

[27] Microsoft, “ACID properties,” Wrox Press, 1998. Available: https://msdn.microsoft.com/en-us/library/aa480356.aspx. Accessed: 26-04-2015.

[28] Microsoft, “Create Custom Templates, SQL server 2014”. Available: https://msdn.microsoft.com/en-us/library/ms166841.aspx. Accessed: 19-05-2015.

[29] Microsoft, “CREATE Statements (Transact-SQL)”. Available: https://msdn.microsoft.com/en-us/library/cc879262.aspx. Accessed: 20-07-2015.

[30] Microsoft, “How to: Create and Execute an SQL Statement that Returns Rows”. Available: https://msdn.microsoft.com/en-us/library/fksx3b4f.aspx. Accessed: 02-12-2015.

[31] Microsoft, “LINQ (Language-Integrated Query)”. Available: https://msdn.microsoft.com/en-us/library/bb397926.aspx. Accessed: 19-05-2015.

[32] MongoDB, “NOSQL Database Explained”, 2015. Available: http://www.mongodb.com/nosql-explained. Accessed: 19-05-2015.

[33] MySQL, “CREATE TABLE Syntax”. Available: http://dev.mysql.com/doc/refman/5.7/en/create-table.html. Accessed: 22-08-2015.

[34] Oracle, “CREATE TRIGGER”. Available: http://docs.oracle.com/database/121/LNPLS/create_trigger.htm#LNPLS01374. Accessed: 30-05-2015.

[35] Oracle, “Introduction to Oracle XML DB”. Available: https://docs.oracle.com/database/121/ADXDB/xdb01int.htm#ADXDB0100. Accessed: 19-04-2015.

[36] Oracle, “Oracle Database 12c PL/SQL”. Available: http://www.oracle.com/technetwork/database/features/plsql/index.html. Accessed: 29-06-2015.

[37] Oracle, “SQL Commands”, 2002. Available: http://docs.oracle.com/html/A95915_01/sqcmd.htm. Accessed 01-06-2006.

[38] Oracle, “SQL”. Available: http://docs.oracle.com/cd/E11882_01/server.112/e40540/sqllangu.htm#CNCPT1732. Accessed: 01-06-2015.

[39] Oracle, “Types of SQL Statements”, 01-07-2013. Available: http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_1001.htm. Accessed: 29-05-2015.

[40] Oracle, “Using Prepared Statements”. Available: http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html. Accessed: 12-06-2015.

[41] PostgresOnline, “The Anatomy of a PostgreSQL - Part 1”, 23-11-2007. Available: http://www.postgresonline.com/journal/archives/2-The-Anatomy-of-a-PostgreSQL---Part-1.html. Accessed: 20-05-2015.

[42] PostgreSQL, “40.6. Control Structures,” 2015. Available: http://www.postgresql.org/docs/9.4/static/plpgsql-control-structures.html. Accessed: 05-06-2015.

[43] PostgreSQL, “Chapter 40. PL/pgSQL - SQL Procedural Language”. Available: http://www.postgresql.org/docs/9.4/static/plpgsql-overview.html. Accessed: 20-07-2015.

[44] R. S. Devarakonda, “Object-Relational Database Systems - The Road Ahead”, *Crossroads Volume 7 Issue 3,* pp. 15-18, 2001.

[45] Realm, “Realm”, AY Combinator Company. Available: https://realm.io/docs/java/latest/. Accessed: 20-05-2015.

[46] RedHat, “Hibernate”. Available: http://hibernate.org/. Accessed: 05-11-2015.

[47] S. Mitchell, “Managing Transactions in SQL Server Stored Procedures”, 03-08-2005. Available: http://www.4guysfromrolla.com/webtech/080305-1.shtml. Accessed: 11-11-2015.

[48] S. Prabhu, “100 Tech Tips, #92 Sample Java program to test connection using IBM DB2 JCC Universal Type 4 driver”, IBM Developer Works, 11-05-2012. Available: https://www.ibm.com/developerworks/community/blogs/IMSupport/entry/100_tech_tips_92_sample_java_program_to_test_connection_using_ibm_db2_jcc_universal_driver4?lang=en. Accessed: 12-11-2015.

[49] SQLines, “Stored Procedures and Functions in PostgreSQL”, 2010. Available: http://www.sqlines.com/postgresql/stored_procedures_functions. Accessed: 23-08-2015.

[50] SQLite, “SQLite docs”. Available: http://www.sqlite.org/docs.html. Accessed: 30-05-2015.

[51] Stackoverflow, “Linq to SQL Data context doesnt commit if multiple inserts are called on table with auto increment”, 2009. Available: http://stackoverflow.com/questions/1171111/linq-to-sql-data-context-doesnt-commit-if-multiple-inserts-are-called-on-table-w. Accessed: 14-12-2015.

[52] T. Grust, M. Mayr, J. Rittinger and T. Schreiber, “Ferry: Database-supported program execution”, Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, *SIGMOD,* 2 July 2009.

[53] TechFaq, “What is a database”, 17-10-2014. Available: http://www.tech-faq.com/what-is-a-database.html. Accessed: 16-04-2015.

[54] Tomassetti, “Representing relationships as first-class citizens in an Object-oriented programming language”, 14-10-2015. Available: http://tomassetti.me/representing-relationships-as-first-class-citizens-in-an-object-oriented-programming-language/. Accessed: 27-11-2015.

[55] Tutorialspoint, “Hibernate - Query Language”. Available: http://www.tutorialspoint.com/hibernate/hibernate_query_language.htm. Accessed: 03-12-2015.

[56] Tutorialspoint, “SQL - SELF JOINS,”. Available: http://www.tutorialspoint.com/sql/sql-self-joins.htm. Accessed: 16-11-2015.

[57] Ú. Erlingsson, “Object-Oriented Query Languages”. Available: http://www.cs.cornell.edu/home/ulfar/oodbms/ooqlang.html. Accessed: 23-07-2015.

[58] W. Kießling and G. Köstler, “Preference SQL: design, implementation, experiences”, *VLDB '02,* 2002.