Evolution and Maintenance of Database Applications

S.K.GARDIKIOTIS, N.MALEVRIS
Department Of Informatics
Athens University of Economics and Business
76 Patission Str., GR104 34, Athens
GREECE

Abstract: - With current software technology trends, databases lie on the heart of almost every modern software system. The evolving nature of the environment under which a database is developed imposes continuous changes in both its structure and data that may respectively affect the validity of any application interacting with the database. In this paper we present a quantitative evaluation of the evolution of a database application as a result of database schema modifications. To increase the maintainability of such applications we have employed a reverse engineering strategy for transforming any part of the database related code that is written outside the DBMS into modules stored inside the DBMS. In addition, we provide a classification of schema modifications according to their criticality on database code invalidation that will further facilitate the maintenance process of database applications.

Key-Words: - Database testing, Reverse engineering, Software Maintenance

1 Introduction

The vital role of databases in modern software applications raises a demand for introducing new methodologies focused on testing and maintaining such applications. One of the main factors that complicate the maintenance of database applications is the evolutionary environment of the database itself. Thus, in database applications the maintenance process, apart from taking into account effects of software evolution [6], should also include actions towards adaptation to changes driven by database evolution [13].

Evolution in databases can be triggered by a variety of reasons including performance boost, altered user requirements or changes to the domain modeled by the database. Given that such evolutionary changes occur quite frequently at the level of database conceptual schema [10] it is clear that we need to develop regression testing strategies for the adaptation of applications interacting with the evolved schema.

In our study we quantify the evolution of database code driven by changes in the conceptual schema of the underlying database. Initially, reverse engineering is applied to database applications whenever embedded SQL (Structured Query Language) statements are encountered in their source code. This results in the conversion of the statements written in the hosting high level language into valid code implemented in the procedural language provided by the Database Management System (DBMS). The derived piece of code that is stored within the DBMS is usually referred to as stored procedure. The benefits of such an approach have to do with the suppression of the impact of schema changes inside the DBMS preventing from propagating influences to the application code.

Subsequently, the possible types of schema modifications are presented. We can distinguish the types of modifications that a database schema change imposes on a stored procedure between performance and validity oriented modifications: a change on a database schema may affect either the performance of a stored procedure (e.g. a change on a table index) or its validity (e.g. modification of a table field accessed by the stored procedure). In the present paper we will consider the second type of changes and we will classify them according to criticality levels. These levels reflect the maintenance effort required towards regaining database code validity.

2 Database Applications

A database application can be defined as any software application that interacts with one or more databases. In practice, database applications can be classified into two categories [1] depending on whether they reside inside or outside the DBMS.

Definition 1. We shall use the term **internal database code** to refer to the source code of database applications that are built and reside in the DBMS. The implementation language consists of both SQL statements and statements of the procedural languages (PL/SQL) that the DBMS

provides. In most DBMSs, internal database code is known as *stored procedure(s).* Hereafter the terms internal database code, database code and stored procedures will be used interchangeably.

Definition 2. Conversely, **external database code** refers to database applications that are implemented outside the DBMS in a general purpose programming language that also supports embedded SQL statements. The communication with the database is performed via calls to one of the standard application programming interfaces (API) that are available for database connectivity such as OLEDB (Object Linking and Embedding Database), ODBC (Open Database Connectivity), JDBC (Java Database Connectivity) etc.

There are many advantages for writing applications in internal database code. The most important is performance boost: stored procedures/functions run faster than ad-hoc queries because they are precompiled meaning that they are parsed, compiled and optimized once during their first execution. They also offer a significant improvement in software reuse as they can be referenced by many external applications. Another benefit concerns the enhancement of database security by operating as a shield against the exposure of database schema details to developers. Additionally, their usage reduces network traffic and client computer's loading.

Theorem 1. Any part of a database application that references a database component and is written in external database code can be rewritten in internal database code.

Intuitively and practically the proof of the above theorem is quite straigthforward: any SQL statement embedded in a generic-purpose programming language can be mapped into a stored procedure inside the DBMS. Since the API provided by the database vendor supports the SQL statement there is a valid mapping from external onto internal code whereas the reverse does not always hold due to the limitations imposed by the API implementation. In fact the embedded SQL statement will be mapped onto one or more new stored procedure(s) and replaced in the original source code by a call to these stored procedures.

The above theorem has important practical implications to both testing and maintenance methodologies for database applications. Actually, reverse engineering on database applications in means of transforming embedded code into stored procedures will operate as code wrapping that prevents the propagation of the impacts of database changes outside the DBMS. Moreover, stored procedures as part of the conceptual schema of a database are accompanied with a variety of information held in data dictionary that facilitates their maintenance and revalidation.

3 Database Object[1] Dependencies

At the conceptual level a database consists of a number of objects that may depend on each other according to their definitions. More precisely, we can distinguish between two types of database object dependencies namely direct and indirect.

Definition 3. A database object *a* is **directly dependent** on a database object *b* when definition of *a* contains an explicit reference to *b.*

Definition 4. A database object *a* is **indirectly dependent** on a database object *b* when it is directly dependent on a database object *c* that is directly dependent on *b*.

As an example of different dependency types consider a function that refers to a view based on a table. According to *definition 4* the function is indirectly dependent on the base table since it is directly dependent on the view which is directly dependent on the table.

Definition 5. A database object *a* is **dependent** on another database object *b* when *a* is either directly or indirectly dependent on *b*.

Most modern DBMS offer a mechanism for detecting dependencies among their objects, often referred to as “dependency tracking” or “dependency management”. Usually, this mechanism is used to ensure the integrity of the database structure upon specific schema modifications. In terms of implementation the dependency management is invoked by the CASCADE/RESTRICT clauses of DROP SQL statements: the CASCADE clause forces the DBMS to drop both the object of the DROP command and all dependent objects, whereas the RESTRICT clause prevents object removal if there is another dependent object. In fact, many DBMS vendors implicitly implement only the RESTRICT clause in DDL (Data Definition Language) statements in order to minimize the side effects of schema alterations. In cases where CASCADE clause is permitted the tracking mechanism detects the invalidation of database code (stored procedures, functions) but does not supply the administrator or

[1] The term *database object* refers to any conceptual object of the database schema and must not be confused with the terminology of object-oriented or object-relational DBMS.

programmer with adequate information to regain the operability of these objects.

4 Schema Modifications

As schema modification we consider any alteration to anyone of the logical components of the database schema. A logical component of a DBMS is defined as each separate object that user can access after connecting to the database. Depending on the DBMS, there are some differences in the set of logical components the database consists of. Throughout the rest of this paper we will refer to PostgreSQL, an object-relational database management system (ORDBMS) [12] which conforms and extends the SQL-92 and SQL-99 standard definition.

PostgreSQL supports the following set of logical components: tables, views, functions, data types, operators, aggregates, conversions and sequences. Tables can be further analyzed to a number of relevant elements called table related elements: columns, constraints, triggers, indexes and rules.

In the following subsections we present the possible modifications of the most common database components intending to estimate their impact on database code validity. We also describe how these modifications can be performed in terms of DDL (Data Definition Language) commands within PostgreSQL ORDBMS. Since our interest is to quantify schema change impact we assume that the CASCADE clause is used whenever applicable.

4.1 Functions/Procedures

PostgreSQL provides four kinds of database functions: query language functions (functions written in SQL), procedural language functions, internal functions and C-language functions. Other database vendors such as Oracle, Microsoft etc provide similar objects for storing code into the database server. For example in Microsoft SQL Server there are two types of database subroutines namely stored procedures and user-defined functions (UDFs), where stored procedures correspond to query and procedural language functions of PostgreSQL while UDFs correspond to internal (built-in) and C-language functions. Regardless of each DBMS implementation for the objects storing database code the impact of the schema modifications is considered to be the same. In the following subsections we will describe the basic operations performed within the *function* object of PostgreSQL.

4.1.1 Addition

A new function is defined via the 'CREATE FUNCTION' command, which is similar but not fully compatible with SQL standard definition. The creation of a new function does not affect the validity of existing functions.

4.1.2 Deletion

PostgreSQL provides the 'DROP FUNCTION' command for function removal, which is not compatible with the corresponding command defined in SQL standard. Upon deletion, all the functions depending on the deleted function will be invalidated.

4.1.3 Renaming

The 'ALTER FUNCTION' command in PostgreSQL is used only to rename a function. It is not compatible with the SQL standard which provides the same command with different functionality. Renaming a function invalidates database code referencing it.

4.1.4 Modification

A function modification includes changes in the code or input/output arguments. To change function's code PostgreSQL uses the 'REPLACE FUNCTION' command, which does not conform to the SQL standard. Changes in function's arguments or returning type require recreation of the function and invalidate any dependent function.

4.2 Tables

4.2.1 Addition

The definition of a new table is performed via the 'CREATE TALBE' SQL command. Obviously, this type of schema change does not affect the code written for the old version of schema unless it is combined with other modifications.

4.2.2 Deletion

Table deletion is performed via the 'DROP TABLE' command and invalidates dependent database code.

4.2.3 Renaming

PostgreSQL provides the 'RENAME' clause of the 'ALTER TABLE' command to rename a table with the following syntax: *ALTER TABLE <table name> RENAME TO <new table>*. This clause is an extension to the SQL standard. Table renaming invalidates directly dependent database code. If database code depends on the table through a view (indirect dependency) the renaming will not cause invalidation.

4.2.4 Modification

Modifying the structure of a table is discussed in section 4.3 where all table related elements are presented analytically.

4.3 Table Related Elements

4.3.1 Columns

4.3.1.1 Addition

Columns can be added to an existing table using the

ADD COLUMN clause of the 'ALTER TABLE' command. This clause is compatible with SQL standard with the exception that it does not support defaults and not-null constraints. Column addition invalidates code that depends on the modified table.

4.3.1.2 Deletion

Column deletion uses the same 'ALTER TABLE' command but with the DROP COLUMN parameter specified. In PostgreSQL deletion of a column does not physically remove the column, but simply makes it invisible to SQL operations. Physical removal takes place over time when existing rows will be updated. This clause is not compatible with the SQL standard definition which does not permit zero-column tables. Column deletion invalidates any code segment that is dependent on the deleted column.

4.3.1.3 Renaming

Column renaming is carried out with the RENAME clause of the 'ALTER TABLE' command and causes the invalidation of database code directly dependent on the renamed column. If database code accesses the column through a view (indirect dependency) the column renaming will not cause invalidation.

4.3.1.4 Modification

A column modification includes: data type change, data length change, precision change, default values change. Each DBMS imposes some limitations upon field modification in order to protect existing data. For instance to perform a data type change the old data type is required to be implicitly converted to the new data type. Apart from data type change, all the other column modifications are carried out via the ALTER COLUMN form of the 'ALTER TABLE' command which is in full conformance with the SQL standard. In PostgreSQL data type change is performed by updating the system table pg_attribute that stores information about column. The modification of a table column will invalidate database code that is dependent on the modified column.

4.3.2 Constraints

PostgreSQL and most DBMSs support the following types of constraints: PRIMARY KEY constraints, FOREIGN KEY constraints, UNIQUE constraints and CHECK constraints. Constraints are defined on table columns usually during the creation of the table via the 'CREATE TABLE' command. Additionally, constraints can be defined after a table has been created using the appropriate form of the 'ALTER TABLE' command. Although constraints enforce data integrity they do not deal with changes on database structure and therefore they have no direct impact on the validity of the database related code. It is however possible to cause run time errors. As an example, consider a definition of a UNIQUE KEY constraint on a table column. Upon insertion if the database code does not check for uniqueness in column values there will be an execution error.

4.3.3 Triggers

Triggers are a special type of database code that take effect upon data modification. A trigger is defined over a table and is invoked in response to an INSERT, UPDATE or DELETE SQL statement. Triggers are similar to constraints in terms of enforcing data integrity and business rules. They are however more powerful than constraints since they can reference any other object in the database. The way that triggers are defined implies that they do not produce any dependency to other database objects except from the table they are based on. Consequently, the addition, deletion or modification of a trigger does not affect the validity of code of other database modules.

4.3.4 Indexes

Indexes are database objects that are used to enhance database performance upon data retrieval. An index can refer to one or more columns of a table and is automatically updated whenever a data change on these columns occurs. Although indexes have a great impact on database performance they do not raise any validation issue concerning database code.

4.4 Views

A view is a virtual table defined by a SQL query. Actually views act as a filter on the underlying tables encapsulating the details of their structure. This yields a protection mechanism against table alterations (consistent interfaces). As an example we have already described the case of table and column renaming (see 4.2.3 and 4.3.1.3). The SQL standard definition permits only definition and deletion of views ('CREATE VIEW' and 'DROP VIEW' commands), which have the same effect with the table corresponding commands.

5 Classification

In this section we summarize the classification of schema modifications presented above according to their criticality on the stored procedures validity. There are namely three categories of schema changes:

Definition 6. A **Compilation Critical (CC)** schema change is a change that requires stored procedure recompilation. Upon such a modification (e.g. data type change of a table field) all the code accessing the altered database object is not compiled successfully and needs to be updated.

Definition 7. An **Execution Critical (EC)** schema change is a change that causes an error during the execution of a stored procedure. Upon such a

modification (e.g. null values are not permitted in a table field) the code accessing the altered database object is compiled successfully although it is not execution safe and therefore needs to be updated.

Definition 8. A **Non-critical (NC)** schema change is a change that does not affect the validity of the code accessing the altered database object.

In order to support the classification described in the previous section a prototype was developed. It collects information from the data dictionary, constructs the dependency graph of database objects and detects, by using this graph, the impact of a schema modification on database code validity. The prototype was used on a small database application implemented under PostgreSQL 7.3.4. The results are presented in Table 1.

Database Object		Object Number	Dependent Functions	Modification Type	Affected Functions	Impact Type
Functions		74	23	Addition	0	NC
				Deletion	23	CC
				Renaming	23	CC
				Modification	23	CC
Tables		15	71	Addition	0	NC
				Deletion	60	CC
				Renaming	60	CC
Table Related Components	Columns	118	71	Type change	71	CC
				Size change	71	CC
				Renaming	60	CC
				Addition	71	CC
				Deletion	71	CC
				Allow null	60	EC
	Constraints	27	71	Addition	71	EC
				Deletion	71	EC
	Triggers	4	0	Addition	0	NC
				Deletion	0	NC
				Modification	0	NC
	Indexes	23	0	Addition	0	NC
				Deletion	0	NC
Views		7	11	Addition	0	NC
				Deletion	11	CC
				Modification	11	CC

Table 1: Classification in PostgreSQL

Database Objects contains each one of the objects defined in the database. *Object Number* refers to the total number of the corresponding database object (e.g. there are 74 functions). *Dependent Functions* refers to the number of database code modules depending on the corresponding database object (e.g. there are 60 functions that depend on one or more tables). The maximum value of this metric is equal to the total number (*Object Number)* of functions. *Modification Type* corresponds to the type of schema modification according to the analysis presented in section 4. *Affected Functions* contains the maximum number of functions that will be affected by the schema modification. This value is always less or equal to the number of *Dependent Functions*. Finally, *Impact Type* refers to the classification of the schema modification according to definitions 6-8.

6 Related Work

Our search in database research literature revealed little work for testing and maintaining database applications against schema modifications. Early work by Sjøberg [10] reveals and quantifies schema evolution in databases. In [7] Li identifies application compatibility as an issue of schema-evolution and addresses the problem of *schema-change transparency* further analyzed to *downward* and *upward* application compatibility. In fact, research proposals and current implementations treat schema-change transparency as part of schema-level rather than application-level evolution. Approaches in this direction include schema versioning [9], schema mutation and schema modification.

An interesting approach is presented in [5] where Daou et al. employ regression testing algorithms to launch validation tests against all the affected modules of a database application. Another approach of testing is introduced in [8] where testing takes into account the validity of the database state after an operation takes place. From the same point of view, [3] and [4] propose a framework that generates valid input data for a database application and tests it against a populated database according to their impact on database state. Chan et al. in [1] and [2] aim to apply white-box testing techniques in database applications by transforming every SQL statement of the application to some general-purpose programming language using relational algebra expressions. In [14] Zhang et al. treat database testing as a constraint satisfaction problem where the constraints are those imposed by the WHERE clause of the SQL statement and those implemented as part of the DBMS (primary-foreign key, null values etc) whereas the objective is a property or properties that the result of the query must have in order to apply the testing method. Finally, Tan et al. [11] propose an approach for elicitating inclusion dependencies by directly analyzing the source code of a database transaction without referring to the database state. Based on this program analysis they produce a set of program path patterns to fulfill the above goal.

7 Conclusion & Future Work

During the maintenance of database applications, one of the main issues is the invalidation of database code due to schema changes. Most of previous approaches support application compatibility through schema evolution and versioning whereas

our intention is to test and revalidate the source code of the application itself. Towards this direction we have employed a reverse engineering strategy for transforming any part of the source code that refers to the database and is written outside the DBMS into modules stored inside the DBMS. Apart from performance benefits, the derived stored procedures as database objects increase the maintainability and relieve the maintainer from the burden of supporting code distributed and written in different programming languages. In addition, we have classified each schema modification according to its criticality on database code invalidation. To test the classification results a prototype was built that constructs a dependency graph of database objects and detects the impact of a schema modification on database code validity. This was run against a small application whose database related code was implemented in the programming language offered by PostgreSQL ORDBMS.

Currently, we are working towards extending our work by applying traditional structural testing techniques on database code. Based on the results of this paper and our future research we aim to derive a framework for automating the maintenance of database applications.

References:

[1] M.Y.Chan and S.C.Cheung, Applying white box testing to database applications, Technical Report HKUST-CS99-01, January 1999.

[2] M.Y.Chan and S.C.Cheung, Testing database applications with SQL semantics, *Proc.of the 2nd Int.Symp. on Cooperative Database Systems for Advanced Applications*, 364-375, March 1999.

[3] D.Chays, S.Dan, P.G.Frankl, F.I.Vokolos and E.J.Weyker, A Framework for testing database applications, *Proc.Int Symp. on Software Testing and Analysis*, 147-157, August 2000.

[4] D.Chays, Y.Deng, P.G.Frankl, S.Dan, F.I.Vokolos and E.J.Weyker, AGENDA: A test generator for relational database applications, *Technical Report*, August 2002.

[5] B.Daou, R.A.Haraty, N.Mansour, Regression Testing of Database Applications, *ACM SAC* 2001.

[6] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, W.M. Turski, Metrics and Laws of Software Evolution – the nineties view, *Proc.of the 4th International Symposium on Software Metrics*, IEEE Computer, 1997, pp.20-32.

[7] X.Li, A Survey of Schema Evolution in Object-Oriented Databases, *IEEE International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 31)*, 1999, pp362-371.

[8] M.Robert and F.J.Maryanski, Automated Test, Plan Generator for Database Application Systems, ACM 1991.

[9] J.F. Roddick, A survey of schema versioning issues for database systems, *Information and Software Technology*, 37(7), 1995, pp. 383-393.

[10] D.Sjøberg, Quantifying schema evolution. *Information and Software Technology*, 35(1), 1993.

[11] Hee Beng Kuan Tan, Yuan Zhao, Automated Elicitation of inclusion dependencies from the source code for database transactions, *Journal of Software Maintenance and Evolution: Research and Practice*, 2003, 15:pp379-392.

[12] J.Worsley, J.Drake. *Practical PostgreSQL*, O'Reilly & Associates, January 2002.

[13] C.Zaniolo, S.Ceri, C.Faloutsos, R.Snodgrass, V.S. Subrahmanian, R.Zigari. *Advanced Database Systems*, Morgan Kaufmann Publishers, 1997.

[14] J.Zhang, C.Xu and S.C.Cheung, Automatic generation of database instances for white-box testing, *IEEE 25th Annual International Computer Software and Applications Conference (COMPSAC)*, 2001.