DBSoft: A Toolkit for Testing Database Transactions

Zuhoor A. Al-Khanjari
Sultan Qaboos University/Department of Computer Science, Muscat, Oman
Email: zuhoor@squ.edu.om

Youcef Baghdadi, Abdullah Al-Hamdani, and Sara Al-Kindi
Sultan Qaboos University/Department of Computer Science, Muscat, Oman
Email: {ybaghdadi, abd}@squ.edu.om; sara.al.kindy@gmail.com

***Abstract*—Databases (DBs) are used in all enterprise transactions, which require attention not only to the consistency of DB, but also to existence, accuracy and correctness of data required by the transactions. While the Atomicity, Consistency, Isolation, and Durability (ACID) properties of a transaction ensure that DB is consistent after the execution of each transaction, it is not sure that the transactions retrieve the correct data. Indeed, the testing phase of the transactions, in the development process, is often ignored. Therefore, there is a need for testing techniques and tools. This paper proposes an architecture, a design, and an implementation of a tester, we refer to as DBSoft, to test transactions, in terms of required data they need to access. The architecture of DBSoft is a layered one. It is made of five components having separate concerns and serving each other: (C1) a parser to collect information, specifically for the metadata, (C2) an input generator to generate test cases, (C3) an output generator to implement the test cases, (C4) an output validator to validate test cases, and (C5) a report generator to generate test reports. DBSoft aims at avoiding cost effective transaction run-time errors.**

***Index Terms*—Databases, Transactions, Testing Tools, Metadata, XML**

I. INTRODUCTION

Database Management Systems (DBMSs) play a crucial role in storing, accessing, and managing data. Most organizations deal with a certain form of DBMS, as these systems provide, through a uniform interface that is SQL, an easy, efficient access to large amount of data by hiding the low-level details of how the data is physically structured and accessed. All enterprise transactions perform a kind of Create/Retrieve/Update/Delete (CRUD) operations against DBs through SQL.

The handled data, through transactions, is used in day-to-day activities or decision-making, which requires a certain attention not only to the consistency of DB, but also to existence, accuracy and correctness of data required by the transactions.

Yet, when transactions running against the DBs abort due to lack or inaccuracy of data can prove to be costly in terms of time and money to organizations. While the Atomicity, Consistency, Isolation, and Durability (ACID) properties of the transaction ensure that the DB is consistent after execution of each transaction, it is not sure that the transactions match the correct data description or value at run-time. When DBs and transactions are not tested before implementation, errors or bugs may appear at any time during the implementation phase or the exploitation (e.g., data population).

In the development process, guided by the three-level architecture, the step that deals with the mapping external schema/conceptual schema is often ignored. This would ensure that the required data by all transactions map into the conceptual schema and vice-versa. This critical testing-kind step ensures that transactions retrieve the correct data description or values, which avoids any costly run-time errors.

One would think that the amount of research being invested in the field would be vast given the importance of DB systems. Unfortunately, the opposite is true. Testing DBs is not easy [1]. For instance, relational DBs have hundreds of interrelated tables structured in a specific way, and described in a metadata (aka catalog), which makes it complex. In addition to this complexity, the metadata is not static as the DB administrator always alters it when business requirements change. That is, the schema and the states of these tables need to be consistently validated to avoid transactions run-time errors.

Therefore, there is a crucial need for techniques and tools to test the correctness of data against the transaction requirements in terms of data. These techniques and tools should be integrated within the development process, preferably before the implementation, i.e., at the mapping interface between transactions and DB.

doi:10.4304/jetwi.5.3.205-212

We propose an architecture, a design, and an implementation of tester, DBSoft, to test transactions, in terms of required data, against DBs they access. The architecture of DBSoft is a layered one. It is made of five components having separate concerns, and serving each other:

C1. A parser: this component generates XML files from the metadata about the input DB to test.

C2. An input generator: this component creates test cases by using the information generated by the parser.

C3. An output generator: this component runs the DB tests by using the populated test cases and saving the results.

C4. An output validator: this component validates the results produced by the output generator.

C5. A report generator: this component produces reports, graphs, and other information about the executions of the test.

DBSoft aims at avoiding cost effective transaction run-time errors.

In this paper, we present the architecture and design of the DBSoft, but we limit the implementation to the crucial component of the toolkit that is the parser. This component parses the metadata, generated by the DBMSs during the creation phase of the DB schema, in order to extract the required information in a standard XML format. Then the XML documents will be used in generating test cases and their expected outcome as well as generating a DB test.

The remainder of the paper is organized as follows: Section 2 provides an overview of related work and existing tools. Section 3 discusses DBMS metadata required by the toolkit. Section 4 presents the architecture of DBSoft. Section 5 details the DBSoft parser. Section 6 presents an implementation of the parser component. Section 7 includes concluding remarks and future direction of the work and research.

II. RELATED WORKS AND TOOLS

The closely related work to what is being proposed in this paper is done by Chays *et al.* [1][2][3] with the AGENDA toolkit. It is composed of five components: a parser, a state generator, an input generator, a state validator and an output validator. AGENDA parses a database transaction, generates test cases, and validates the result. The AGENDA parser is the focal step of interest for now. Based on a modified form of PostgreSQL's [4], the internal parser collects relevant information about a DB supplied to it and stores the information in an internal DB called the AGENDA DB.

In [5], a framework is presented to perform efficient regression tests on DB transactions. In [6], issues with the automatic running of DB regression tests are listed. A framework for creating a test database is presented in [7] and [8], while in [9], the framework tests the features of DBMSs and also includes an automatic DB generator called QAGen.

Due to security and confidentially issues tied with "live" DB data, an automatic data generating tool is proposed in [10]. It is called ADG. Automatic Data Generation is also highly useful in terms of its ability to create a desired problem situation within a DB for testing.

A number of different methods in creating the test cases exist, and these are discussed in [11], [12], [13] and [14].

There is no evident related work in the academic community to collecting information from DBMS metadata to be used in DB testing, and it is safe to say that DBSoft toolkit is the first to apply this concept.

The main two differences between our approach and AGENDA are:

1. Instead of developing a new parser, we extract relevant information from DBMS metadata.
2. DBSoft toolkit does not store the information in tables, but represents it as standard XML documents. Indeed, storing the information in an internal DB as done by AGENDA is not efficient in terms of simplicity and specifically extensibility. DBSoft uses XML document to store information due to its nature of being self-descriptive, easily processed and human-readable. However, XML tags and attributes are based on AGENDA DB tables.

One of the aims of the DBSoft toolkit is to enable performing regression tests on DB while they are updated. DBSoft will generate test cases and a test DB state by means of the information stored in XML documents that are produced in the data extraction step. The test cases will be used to run the test DB.

III. DBMS METADATA

There exist many definitions of the concept metadata. The most general is “data about data”. In DB systems, a metadata describes and provides information about all the objects of a DB such as tables, views, indexes, sequences, stored procedures, functions, etc. For example, in Oracle, a table is described by more than fifty data such as name, number of columns, number of rows, etc. In PostgreSQL, objects are described in Information_schema.

The SQL standard defines a uniform interface to access this data, but not all DBMSs implement this feature. Hence, a number of different mechanisms have evolved with accessing metadata over different systems: For instance:

- Oracle has data dictionary and metadata registry.
- PostgreSQL provides system catalogs and Information_Schema.

- SQL Server and MySQL contain system catalogs and the Information_Schema, but the method for querying it differs from that of PostgreSQL.

Additionally, due to the nature of where the information is being collected from, it is safe to say that the integrity of the information collected is increased and hence more trustworthy.

Metadata is the most relevant input to the testing step, as it should describe all the data required by the transactions running against the DB.

IV. DBSOFT ARCHITECTURE

The architecture of DBSoft is a layered one. It is made up of components having separate concerns, but serving each other as shown in Figure 1. This ensures:

- A smooth and independent design and implementation of the components.
- A straightforward wrapping of the components into services for an easy adoption of Service-Oriented Architecture (SOA).
- A replacement of any of the components by other components when some non-functional requirements such as reliability or performance.
- Security of the components.

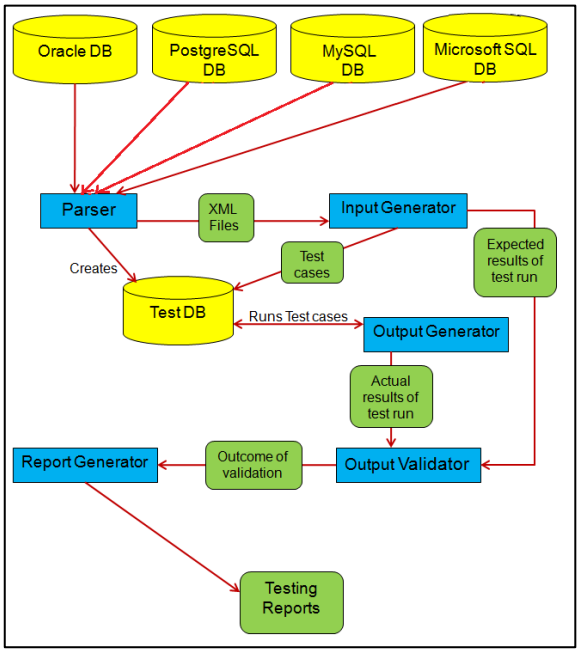


Figure 1. DBSoft Architecture

The components are specified as follows:

C1. A parser: this component is involved in collecting information from the metadata about the DB to test. It generates XML files consisting of the collected information for the DB test.

C2. An input generator: this component creates test cases by using the information in the XML files generated by the parser component. It populates the DB test with test cases and creates an XML file of the expected results of each test case.

C3. An output generator: this component runs the DB tests by using the populated test cases and saving the results.

C4. An output validator: this component validates the results produced by the output generator by comparing them with the expected results from the input generator.

C5. A report generator: this component produces reports, graphs, and other information about the executions of the test.

The aforementioned architecture guides us towards a testing process that consists of five steps. Each component translates into a step in the testing process.

1. *Step 1*: information collection
2. *Step 2*: test case generation
3. *Step 3*: test case implementation
4. *Step 4*: test case validation
5. *Step 5*: report generation

The stepping-stone into the DBSoft testing tool is the parser. In this work, much of attention is given to the parser, as it constitutes the corner stone of the system. Through the correct design and implementation of the parser, we would ensure that the rest of the components will work properly, thus meeting the requirements of DBSoft as a powerful tool for testing DBs.

V. DBSOFT PARSER

Using the parser, a DB could be parsed; and relevant information would be extracted into XML documents.

In order to be able to create an efficient and real-world usable tool for DB testing, the first consideration is that DBs need to be transformed into a uniform object, as there is a number of DBMS in existence today with different metadata, catalogs, and physical storage mechanisms.

The creation of standard XML documents, outlining the details of the physical organization and structure of DB schema, will also aid in:

(i) the creation of efficient test cases, and a test DB for testing.

(ii) the generation of test data for DB population.

(iii) the validation of the results of the test runs.

A. DBSoft parser functioning

The following are the sub-steps taken by the DBSoft parser:

1. User inputs location of DB

The DBSoft parser collects the relevant information from the metadata in the DBMS. The program needs to be pointed to the location of the DB, to be tested, in order to access the metadata. The user will provide the correct information on the location of the DB. This includes the host, the port number, the DB name, the username, and the password.

2. DBSoft parser creates connection

Once the location of the database is input correctly, the DBSoft parser opens a connection to the DB.

3. Information collected from the metadata

The DBSoft parser will begin collecting the relevant information from the metadata. This is done by querying the metadata by means of relevant commands through the JDBC API as shown in Figure 2. Since the mechanism of accessing metadata differs from one DBMS to another, the commands issued will be according to the DBMS in question.

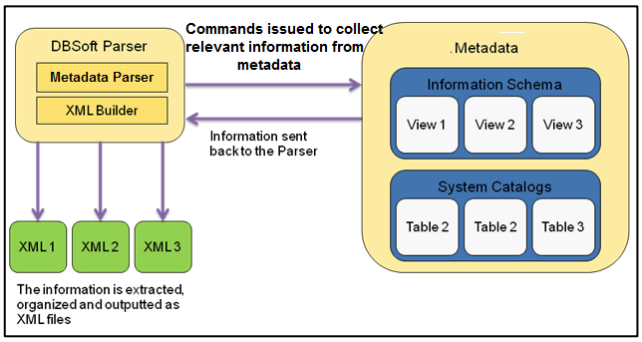


Figure 2. DBSoft Parser Component: Collecting Information from Metadata

4. Information extracted into XML document

The information that has been collected in the parsing step will be organized and extracted into XML documents, such as:

- Tables (containing information on tables, number of attributes, type, etc).
- Attributes (containing information on attributes, tables they are contained in, type, etc).
- Boundary values (containing information on boundary values, range, type, etc).
- Table relationships (relationships between tables in the database and their associated attributes).

Figure 3 illustrates a standard structure for XML documents to represent relational databases maintained in different DBMS. For each database system, an XML tree is generated by extracting information from its DBMS metadata. The database is composed of a set of table elements. Each table is composed of three main elements: *name* (the table name), *attributelist* (list of attributes in the table with their types and constraints), and *constraints* (other constraints in the table such as the primary key, foreign keys, unique attributes). Information about each table can be extracted from the tables in the DBMS metadata catalog.

The *attributelist* element for each table is composed of a list of attribute elements to represent information about each attribute such as name, type, default value, maximum and/or minimum values. The information about each attribute element in the XML tree can be extracted from the attributes and the boundary values in the metadata catalog. The constraint element is composed of a list of constraints, including primary, foreign keys, and other constraints on the database. In the implementation section, we illustrate how an XML document is generated using the proposed XML tree structure for an existing database system as shown in Figure 7.

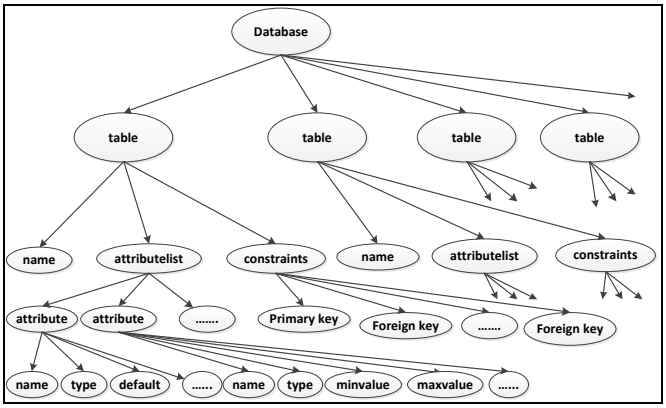


Figure 3. XML Tree Structure for a Relational Database

B. Transactions of DBSoft Parser

The XML files produced by the DBSoft parser can be employed in several testing transactions. The aim for the DBSoft toolkit is to cover most if not all DB testing methods.

Analysis can be made in regards to whether the transaction program is behaving as specified, i.e., checking correctness, in addition to reflecting upon whether the DB schema correctly models the organization of real-world data.

Test cases can be produced by using the information in the XML files, such as creating specific queries that will make a DB transaction 'break'. Data tailored for specific problem areas found within the DB can be generated, and in turn will be populated within a test DB that has been built using the information in the XML documents.

Regressions tests can be performed on databases when they are updated, ensuring that everything still work as specified. Validity of the results will be made using the information extracted in the parsing step (constraints, types, etc) in addition to looking into other automatic means.

VI. IMPLEMENTATION

To implement DBSoft, we have used PostgreSQL [15] DBMS and the same approach can be used to extract metadata from other DBMSs. It is an object-relational DBMS, originally developed at UC Berkeley. It is open-source. In this work, we will be dealing with the

PostgreSQL's (version 8.4.5) metadata, namely its system catalogs and Information_Schema.

System catalogs in PostgreSQL are regular tables that store the schema or metadata on a parallel DB. There are about 60 system catalog tables, each catalog deals with a specific type of metadata.

The Information_Schema is a set of views that provides information about objects defined in the current DB. It is portable and more stable, as it is defined with SQL standard, contrary to the system catalogs mentioned above which are specific to PostgreSQL. There are about 51 views.

The information stored within each form of metadata is nearly the same. However, there exist some differences between the two with the amount of information stored. Hence, they complement each other in terms of the information they contain. Thus, we can extract information from both metadata rather than concentrating on one only.

PostgreSQL metadata storages can be accessed using an Application Programming Interface (API) such as Java Database Connectivity API (JDBC) [15]. JDBC is a middleware that consists of a set of classes that enables transactions developed with Java to interact with DBs, whereas the relevant information can be extracted from the metadata sources using SQL commands. This makes both the system catalogs and the Information_Schema, a trove of information fit to be employed in parsing a PostgreSQL DB. Figure 4 shows an example of query that gets the names of all the tables in a DB by means of the Information_Schema. The last part of the command ensures that the names of the tables contained within the system catalogs and the Information_Schema will be retrieved as well.

```
SELECT table_name
  FROM information_schema.tables
 WHERE table_type = 'BASE TABLE'
       AND table_schema NOT IN
              ('pg_catalog',
    'information_schema');
```

Figure 4. Querying the Schema to get Table Information

A. Parser front-end

The UI of the DBSoft toolkit gives users the ability to parse an input DB, output the DBSoft schema, and to recreate the information collected from the parsing stage as the SUT DB as shown in Figure 5 and Snapshot 1.

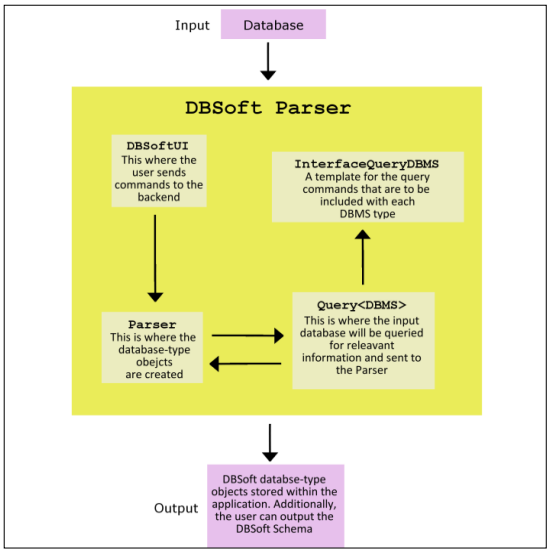
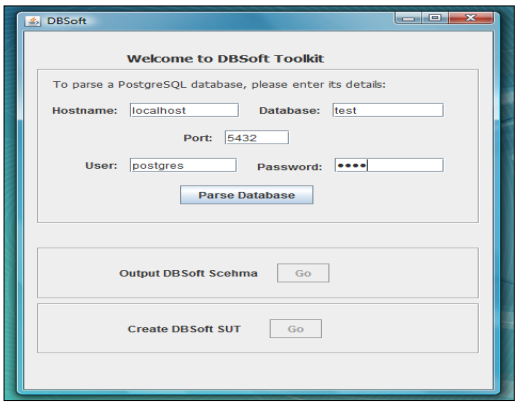


Figure 5. The Internal Build of the Parser

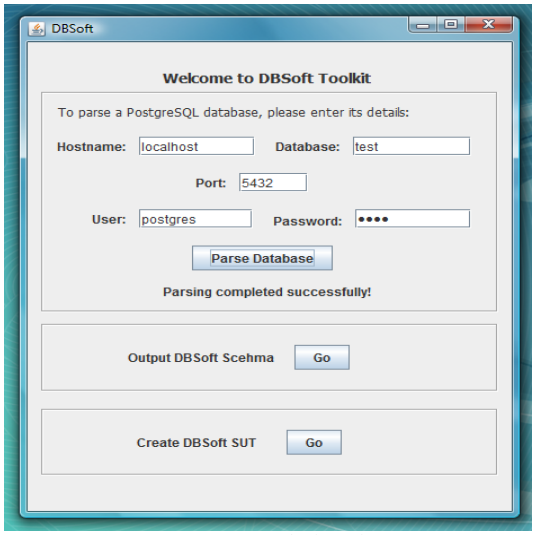


Snapshot 1. The User Interface of DBSoft

B. Schema Creation

At the end of parsing the input database, by means of both SUTBuilder and ParsePostgreSQL, and storing the relevant information in the DBSoft database-type objects, the DBSoft schema is finally created as shown in the Snapshot 2.

As mentioned before, this is based on tables in the [5] AGENDA DB. The main difference however is that the AGENDA DB is an actual DB that will be used in replicating the original input DB i.e., it will be queried for the information collected. This is contrary to the DBSoft schema that is an internal representation of a schema within the DBSoft Java application.



Snapshot 2. End of Parsing Message

The users have an option of outputting the DBSoft schema as an actual schema, but this will not make a difference to the replication of the DB, and the creation of test cases, as the DBSoft DB-type objects will be used in this point.

C. XML document Generation

To evaluate the proposed approach, we have used a simple PostgreSQL database application with two tables (Department and Employee) as shown in Figure 6.

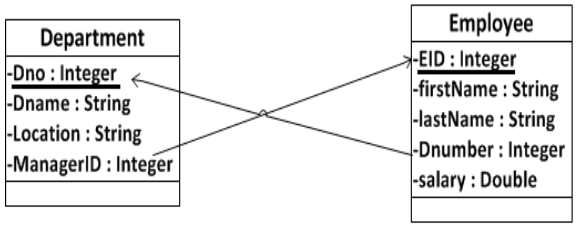


Figure 6. UML Diagram for Company Database Schema

```
<?xml version="1.0" ?>
<!-- An XML file generated for a test database named
     myTestDatabase  composed of two tables-->
<database >
<dbname>SQU Test Database</dbname>
<!--Department table definition -->
<table>
<name>Department</name>
<attributelist>
        <attribute>
                <name>Dno</name>
                <type>integer</type>
                <default> 1</default>
                 <minvalue>1</minvalue>
                <maxvalue>99</maxvalue>
        </attribute>
        <attribute>
                <name>Name</name>
                <type>String</type>
                <maxlength>30</maxlength>
        </attribute>
        <attribute>
                <name>location</name>
                 <type>String</type>
```

The DBSoft was used to extract the metadata for the database in Figure 6 from PostgreSQL database, and the corresponding XML file was generated as shown in Figure 7. The XML file is composed of a single database element <database> that contains one or more <table> elements (each corresponds to a relational database). Each <table> element is composed of <name> (relational table name), <attributelist> (list of table attributes) and <constraints> (list of constraints in the table such as the primary keys <primarykey> and foreign keys <foreignkey>).

Each <attributeList> element contains one or more <attribute> elements that corresponds to all attributes in a given relational database table. The <attribute> element is composed of serious XML element including attribute name <name>, attribute type <type>, default value <default>, maximum value <maxvalue>, minimum value <minvalue> and maximum field length <maxlength> elements.

The XML file can be used to generate test cases for the database using XML tags such as <types>, <maxlength>, <maxvalue>, <minvalue>, <default>,.., etc. Also, the XML file can be used to validate SQL queries such as checking the validity for the table names, attribute names, and constant ranges.

```
<!-- Employee table definition -->
<table>
<name>Employee</name>
<attributelist>
        <attribute>
                <name>EID</name>
                <type>integer</type>
                <minvalue>10000</minvalue>
                <maxvalue>99999</maxvalue>
        </attribute>
        <attribute>
                <name>firstName</name>
                 <type>String</type>
                <maxlength>25</maxlength>
        </attribute>
        <attribute>
                <name>lastName</name>
                 <type>String</type>
                <maxlength>50</maxlength>
        </attribute>
        <attribute>
                <name>Dnumber</name>
                 <type>integer</type>
        </attribute>
```

```
                <maxlength>20</maxlength>
                <default>Muscat, Oman </default>
        </attribute>
        <attribute>
                <name> ManagerID</name>
                <type>integer</type>
        </attribute>
</attributelist>
<constraints>
        <primarykey> Dno </primarykey>
        <unique>Name</unique>
        <foreignkey>
                <attribute> ManagerID <attribute>
                <reftable> Employee </reftable>
                <refattribute>EID<refattribute>
        <foreignkey>
</constraints>
</table>
        <attribute>
                <name>salary</name>
                <type>double</type>
                <minvalue>200</minvalue>
                <maxvalue>3000</maxvalue>
        </attribute>
</attributelist>
<constraints>
        <primarykey>Dnumber</primarykey>
        <foreignkey>
                <attribute> ManagerID <attribute>
                <reftable>Department</reftable>
                <refattribute>Dno<refattribute>
        <foreignkey>
</constraints>
</table>
</database>
```

Figure 7. XML File for a Company Database

VII. CONCLUSION AND FUTURE DIRECTION

Nowadays, the need for an automated tool in testing DB transactions is crucial and critical. DBs support large organization activities if not all, and hence would need to behave correctly to avoid errors and bugs.

DBSoft toolkit is proposed in this paper. It is an efficient, practical tool for DB testing. This is realized through the following milestones:

- Implementing the testing process has been realized with the DBSoft parser that collects the required information about the DB to be tested, specifically its metadata.
- The creation of test cases helps in enhancing the DBSoft tester to use a standard method of entering DB information, i.e., XML documents produced by the DBSoft parser. A range of different test cases will be generated and employed in checking the correctness of the DB.
- Another milestone is to enable the DBSoft tester to use test cases for regression testing on DBs, since it generates test cases that can be stored and run several times.

We expect DBSoft to be a startup for DB testing community towards the realization of a standard DBMSs testing process. Although, the complete process has not been presented in the paper, a stepping stone into it has been with the standardizing transaction of the DBSoft parser.

Further development first concerns with the development of all the components. Then, we foresee a DB testing method.

REFERENCES

[1] D., Chays, Y., Deng, P. G., Frankl, S., Dan, F. I., Vokolos and E. J., Weyuker, An AGENDA for Testing Relational Database Applications. Journal of Software Testing, Verification and Reliability, 14(1), PP.17–44, March 2004.

[2] D., Chays, Y., Deng, P. G., Frankl, S., Dan, F. I., Vokolos, and E. J., Weyuker, Demonstration of AGENDA Tool Set for Testing Relational Database, ICSE '03 Proceedings of the 25th International Conference on Software Engineering, PP. 802-803, Published by IEEE Computer Society, May 2003.

[3] Y., Deng, P., Frankl, and D., Chays, Testing database transactions with AGENDA, Proceedings of the 27th international conference on Software engineering (ICSE 05), ACM Press, PP. 88-96, May 2005.

[4] PostgreSQL. The PostgreSQL Global Development Group, [Online] http://www.postgresql.org/ [accessed: 12 July 2012].

[5] F., Haftmann, D., Kossmann, and E., Lo., A Framework for Efficient Regression Tests on Database Applications, The VLDB Journal — The International Journal on Very Large Data Bases, 16(1), PP.145-164, January 2007.

[6] F., Haftmann, D., Kossmann, and A., Kreutz, Efficient Regression Tests for Database Applications. Conference on Innovative Data Systems Research (CIDR), pp. 95-106, 2005.

[7] E., Lo, C., Binnig, D., Kossmann, M. T., Ozsu, and W.-K., Hon, A Framework for Testing DBMS Features. VLDB Journal, 19(2), pp.203–230, April 2010.

[8] N., Bruno and S., Chaudhuri, Flexible database generators, Proceedings of the 31st international conference on Very large data bases, 2005.

[9] C., Binnig, D., Kossmann, E., Lo, and M.T., Özsu., QAGen: generating query-aware test databases, Proceedings of the 2007 ACM SIGMOD international conference on Management of data, January 11-14, 2007.

[10] N. R., Lyons, An automatic data generating system for data base simulation and testing, ACM SIGMIS Database, 8(4), PP.10-13, 1977.

[11] T.Y., Chen, P.L., Poon, and T. H., Tse., A Choice Relation Framework for Supporting Category-Partition Test Case Generation, IEEE Transactions on Software Engineering, 29(7), PP.577-593, July 2003.

[12] H., Bati, L., Giakoumakis, S., Herbert, and A., Surna, A genetic approach for random testing of database systems, Proceedings of the 33rd international conference on Very large data bases, 2007.

[13] C., Mishra, N., Koudas, and C., Zuzarte, Generating targeted queries for database testing, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, PP. 499–510, 2008.

[14] J., Tuya, M.J.S., Cabal, and C.de la, Riva, Mutating database queries, Information and Software Technology, 49(4), PP.398-417, 2007.

[15] PostgreSQL JDBC Driver, The PostgreSQL Global Development Group, Last published: 6 February 2012 [Online] http://jdbc.postgresql.org/ [accessed: 20 August 2012].