CRUD-DOM:
A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms - an Enhanced Performance Assessment Based on a Case Study

Oscar M. Pereira[1], Rui L. Aguiar[2]
Instituto de Telecomunicações
University of Aveiro
Aveiro, Portugal
{omp[1],ruilaa[2]}@ua.pt

Maribel Yasmina Santos
Algoritmi Research Center
University of Minho
Guimarães, Portugal
maribel@dsi.uminho.pt

***Abstract*—the development of database applications comprises three different tiers: application tier, database tier and finally the middle tier also known as the data access layer. The development of each tier *per-se* entails many challenges. Very often the most difficult challenges to be addressed derive from non-functional requirements, as productivity, usability, performance, reliability, high-availability and transparency. This paper is focused on defining and presenting a model for the data access layer aimed to integrate object-oriented application tiers and relational database tiers. The model addresses situations on which users need to explicitly write down complex static Create, Read, Update and Delete (CRUD) expressions and simultaneously get advantages regarding some non-functional requirements. The model, known as CRUD Data Object Model (CRUD-DOM), tackles the following non-functional requirements: performance, usability and productivity. The main contributions of this paper are threefold: 1) to present the CRUD-DOM model; 2) to carry out an enhanced performance assessment based on a case study; 3) to present a tool, called CRUD Manager (CRUD-M), which provides automatic code generation with complementary support for software test and maintenance. The main outcome of this paper is the evidence that the pair CRUD-DOM and CRUD-M effectively addresses productivity, performance and usability requirements in the aforementioned context.**

Keywords - CRUDDO; CRUD-DOM; database; impedance mismatch;, high-performance computing; measurement; middle tier.

I. INTRODUCTION

In order to bridge the gap between the object-oriented and the relational paradigms, a model, known as CRUD-DOM, has been already presented [1]. This paper is an extended version of [1].

In spite of their individual successes, object-oriented and relational paradigms are simply too different to bridge seamlessly, leading to difficulties informally known as *impedance mismatch* [2]. The diverse foundations of the object-oriented and the relational paradigms are a major hindrance for their integration, being an open challenge for more than 45 years [3]. The challenge derives from the multiplicity of aspects that need to be bridged across both paradigms: imperative languages versus declarative languages; compilation and execution performance versus search performance; classes, algorithms and data structures versus relations and indexes; transactions versus *threads*; null pointers versus null for the absence of value [3], and finally, inheritance versus specialization. The impedance mismatch thus presents several challenges for developers of database applications, where often both paradigms are found. These challenges are especially noticeable in environments where production code is under strict development deadlines, and where (timely) code development efficiency is a major concern. In order to cope with the impedance mismatch issue, several solutions have emerged, such as language extensions (SQLJ [4], LINQ [5]), call level interfaces [6] (JDBC [7], ODBC [8] ADO.NET [9]), object/relational mappings (O/RM) (Hibernate [10], TopLink [11], LINQ [5]) and persistence frameworks (JDO [12], JPA [13]). Language extensions may provide static syntax and type checking but always rely on proprietary standards. Call level interfaces, despite their performance, provide no static syntax or static checking. O/RM have the advantage of treating data as objects but do not take the advantage of the database engine performance and further rely on proprietary standards. Persistent frameworks have the same drawbacks as O/RM. Despite their individual advantages, these solutions have not been developed to effectively address situations where users need to write complex static CRUD (Create, Read, Update, Delete) expressions. Table I presents an example of a not very simple CRUD expression that is not easily supported by any current solution. The increasing of the query complexity increases the weaknesses of current solutions.

TABLE I. A CRUD EXPRESSION

```
Select pt.pt_id, pt.pt_fName, pt.pt_lName
    From pt_pilot pt,cc_circuit cc,cf_classif  cf
    Where pt.pt_id=cf.cfPt_id  and  cf.cf_date=cc.cc_date and
           cf.cf_position not between 1 and 3
     Group by pt.pt_id, pt.pt_fName, pt.pt_lName
     Having count(cf.cf_position) = (Select count(*) From Cc_circuit … )
 Union
 Select top 5 distinct( ….
     from …
     ….
 Order by …
```

Not easily supported means that current solutions push users to deal with additional issues, as a decay of performance, the usage of proprietary language extensions, the usage of proprietary mapping techniques, the absence of support to edit and dynamically test CRUD expressions and some SQL features not easily supported or even not supported at all.

A. Performance, Usability and Productivity

CRUD-DOM addresses mainly three non-functional requirements: performance, usability and productivity. In this section we introduce their basic concepts and also justify and emphasize the relevance of each one.

Performance

Performance is a non-functional software requirement that, among other issues, evaluates how well a system or a component copes with a set of requirements namely for timeless [14]. In this context, two dimensions may be considered: responsiveness and scalability. Responsiveness evaluates conformance with response time requirements evaluating the amount of time to accomplish a task or the number of tasks that can be accomplished in a given period of time. Scalability evaluates the capacity of a system to handle growing demand of power computation while keeping its responsiveness. In this paper we are focused on responsiveness. Scalability will be addressed in future works.

Performance is a pervasive outcome of software systems [15]. Everything affects it, as software design, programming paradigms, programming languages, compilers, operating systems, communication networks, hardware and third party software. As a pervasive quality, performance opens many opportunities to research contributions. Very often, it is one of the most challenging non-functional software requirements in database applications. System architects and system designers are called to decide upon many and difficult options. Each option has an impact on the overall performance. As an example, the middle tier may be built around distinct technologies and solutions, as previously mentioned, being CLI one of them. Despite CLI drawbacks, they cannot be discarded as an important and valid option whenever performance and SQL expressiveness are considered key issues [3]. CLI provide mechanisms to encode Create, Read, Update and Delete (CRUD) expressions inside strings, easily incorporating the power and the expressiveness of SQL. Thus, power and expressiveness are crucial advantages of CLI but this comes with unavoidable and important drawbacks (see detailed discussion in section III).

Usability

Usability is another non-functional software requirement in Software Engineering. It is linked to the software quality design [16]. Several definitions may be found for the usability concept [17-22]. We may accept the definition of Jacob Nielson [17], which is focused on concepts as *Learnability*, *Efficiency*, *Memorability*, *Errorless-pronability* and *Satisfaction*. The application of usability in this work is twofold. The first one is related to the development process of the data access layer. It is addressed by the CRUD-M, which must provide a GUI with improved usability. The second one is related to the usage of the data access layer. The access data layer should provide an improved usability to developers of the application tier.

Productivity

In this work, productivity comprises the factors that may influence costs during the three phases of the software application life-cycle: development, test and maintenance of access layers.

The development phase usually unlocks financial and material resources, and also motivates the involved human resources. On the contrary, the test and maintenance phases are usually neglected and therefore we will pay some additional attention to them.

The costs associated with software testing are very high and may exceed 30% of the total cost of a project [23]. Two of the most relevant sources for such a high cost comprise the attitude assumed by the development team [24] and also by the absence of an adequate infrastructure dedicated to software testing. In the U.S. in 2002 it was estimated a cost between \$22 and \$59.1 billion [25]. In opposite to what is commonly accepted, rather than an act of testing, the software testing should be seen as an overall strategy to be included in the entire life-cycle of a software system: "*the act of designing tests is one of the best bug preventers known*", Beizer in [26].

Software maintenance is well known for its very high costs and delays in its implementation. Despite being the aspect that consumes more resources during the product life-cycle [27], it has usually been neglected. Software maintenance is an inevitable activity resulting from requests for assistance derived from its usage and from its aging. Software maintenance is associated with different sources but it is generally classified into 4 categories, each one with different weights [28]: adaptive - 25% (changes in the environment where software works); perfective – 50% (adaption to new requirements); corrective – 21% (error correction); preventive – 4% (prevention of future errors). These values, although presented in 1980, still continue to be accepted and cited in several publications [27-29]. Some sources of software maintenance may not be easily controlled by the development team, as are the adaptive and perfective sources. But the other two, mainly the corrective one, have their basis and origin in flaws occurred during the development and test of the product.

B. Motivation

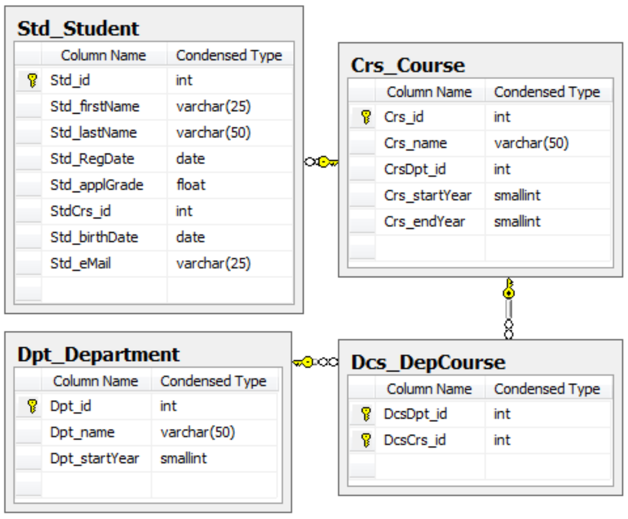
The motivation for this work is anchored in the fact that none of the available current solutions and technologies address effectively and simultaneously all the following features:

- <u>Hand-written CRUD expressions</u> - business logic in database applications very often rely on SQL statements that have to be hand-written. This may be derived from the fact that CRUD expressions are too complex and/or CRUD expressions cannot be inferred from any other data model.

- Decoupled data access layer - in order to completely decouple the three tiers of database applications, the data access layer should be developed as a separate component. This will ensure not only the decouple at its usage level (application level) but also at its development level (organizational/responsibility level).
- Technological independency - technological independency assures that: solutions may run on any environment; users are not compelled to learn new technologies.
- Productivity - productivity should be maximized by exempting users from writing any source code; source code should be automatically generated and tested. Maintenance activities should require minimum user effort.
- Usability: usability should always be presented as a key concern in all aspects: development of the data access layer and also the usage of the data access layer.
- Performance: the performance of the data access layer must always be the main concern.

This work aims to provide a solution that copes with the aforementioned features. For such, we developed a model, known as CRUD Data Object Model (CRUD-DOM) where each CRUD expression is wrapped into a type-safe and type-state object-oriented component, known as CRUD Data Object (CRUD-DO). Furthermore, we developed a tool addressing automatic CRUD-DO generation having as only input the standard SQL statements written by users. This tool is known as CRUD Manager (CRUD-M).

Throughout this paper, by default - unless explicitly referred, all examples are based on Java, SQL Server 2008 and JDBC (CLI) for SQL Server (sqljdbc4). Code snippets may not execute properly since we will only show the relevant code for the points under discussion. For conciseness, Figure 1 presents a partial view of a database schema, which will be used throughout the examples of this paper.



Std_Student

Column Name	Condensed Type
Std_id	int
Std_firstName	varchar(25)
Std_lastName	varchar(50)
Std_RegDate	date
Std_applGrade	float
StdCrs_id	int
Std_birthDate	date
Std_eMail	varchar(25)

Crs_Course

Column Name	Condensed Type
Crs_id	int
Crs_name	varchar(50)
CrsDpt_id	int
Crs_startYear	smallint
Crs_endYear	smallint

Dpt_Department

Column Name	Condensed Type
Dpt_id	int
Dpt_name	varchar(50)
Dpt_startYear	smallint

Dcs_DepCourse

Column Name	Condensed Type
DcsDpt_id	int
DcsCrs_id	int

Figure 1. Partial view of the database schema

This paper is organized as follows. Section II presents related work. Section III highlights the impedance mismatch problem. Section IV describes our proposed model (CRUD-DOM), while section V presents the automatic code generation tool (CRUD-M). Section VI presents performance assessment and finally, Section VII presents the final conclusion.

II. Related Work

This section presents the different approaches for the integration of object-oriented and relational paradigms. As a well-known problem in industry, multiple techniques and solutions have been released to address the impedance mismatch problem. For some solutions we will present a real case but always dealing with a very simple query.

Embedded SQL [30] is a method for writing SQL statements in-line with regular source code of the host language inside source files. The SQL statements provide the database interface while the host language provides the remaining support needed for the application to execute. The files are then pre-processed (pre-compiled) in order to check the correctness of the SQL statements namely against the database schema, host language data type and SQL data type checking, and finally syntax checking of the SQL constructions. SQLJ [4] is an example of an Embedded SQL standard, which provides language extensions for embedding SQL statements in regular Java source files. Some SQLJ disadvantages, which are common to most Embedded SQL technologies: 1) SQLJ relies on an extra standard; 2) SQLJ does not decouple SQL statements from regular source code; 3) SQLJ is not suited for client-server environments; 4) SQLJ does not provide a clean object-oriented interface to the assisted application; 5) SQLJ does not provide assistance regarding the maintenance of SQL statements; 6) SQLJ requires a JVM (Java Virtual Machine) built in the database; 7) In practice, embedded SQL has never been widely adopted by end users. Table II shows an example using SQLJ. Examples of other languages that support embedded SQL are: C, C++, COBOL and Fortran.

Table II. SQLJ example

```
// Java
void getStudent( int id ) throws SQLException {
   String firstName = null;
   String lastName = null;
   #sql {
         Select Std_firstName, Std_lastName
            INTO: firstName, lastName
            From Std_Student
            Where Std_id = :id
   }
   System.out.println("Student's name: " + firstName + " " + lastName);
}
```

Despite the aforementioned general disadvantages, some embedded SQL features may be considered as advantages such as: it is based on single development environment with a strong interconnection between the two paradigms; unlike other solutions, embedded SQL does not need to be executed to check the correctness of the SQL syntax. This task is executed by the pre-compiler.

Object-relational mapping [31, 32] is a programming technique aiming at enforcing relational data models to be closely aligned with the object-oriented paradigm. The relational to object-oriented translation is driven by an explicit mapping (generally in XML) or by schema annotations (inside the source code file). Much of the enforcement is on behalf of getting an object-oriented logic access layer coping with the impedance mismatch [2] issue. Every relational concept must, somehow, have its corresponding concept(s) in the object-oriented paradigm. Very often, mainly in legacy databases, the translation is not straightforward, leading to complex translations, as the case of the relationship and specialization concepts. In these cases, besides the aforementioned hindrance, the relational model lacks essential conceptual information obliging oneself to an extra effort on defining relationship direction, cardinality, etc. Nevertheless, O/RM techniques have been quite successful, either as commercial products (e.g., Oracle TopLink [11], ADO.NET Entity Framework [33], LINQ [5]) or as open source projects (e.g., Hibernate [10]). Albeit this achieved success, well known O/RM drawbacks are unavoidable: 1) each O/RM programming technique relies on proprietary standards introducing new mapping schemas and new SQL-equivalent manipulation languages; 2) O/RM entails an additional effort to map the relational model into the object-oriented model; 3) performance and expressiveness are the two main O/RM penalties; 4) complex CRUD expressions may be supported but they must be hand written and users have no support for their editing and testing. Table III shows a Hibernate example with HQL and Table IV shows an example with Hibernate without HQL (both in Java). Table V shows an example with LINQ in C#. For conciseness, the mapping schema and mapping classes are not explicitly presented.

TABLE III. HIBERNATE EXAMPLE WITH HQL

```
// Java
void getStudent(int id) {
   Session s=HibernateUtil.getSessionFactory().getCurrentSession();
   List list=s.createQuery("from Std_Student").
                 setInteger("Std_id", id);
   Student std=(Student) list.get(0);
   firstName=std.Std_firstName();
   lastName=std.Std_lastName();
   System.out.println("Student's name: " + firstName + " " + lastName);
}
```

TABLE IV. HIBERNATE EXAMPLE WITHOUT HQL

```
// Java
void getStudent(int id) {
   Session s=HibernateUtil.getSessionFactory().getCurrentSession();
   Student std=(Student) s.load(Student.class,id);
   firstName=std.Std_firstName();
   lastName=std.Std_lastName();
   System.out.println("Student's name: " + firstName + " " + lastName);
}
```

TABLE V. LINQ EXAMPLE

```
// C#
void getStudent(int id) {
   Student std=from s in db.StdStudent where Std_id=id select s;
   firstName=std.Std_firstname;
   lastName=std.Std_lastname;
   Console.WriteLine("Student's name: " + firstName + " " + lastName);
}
```

Despite the aforementioned disadvantages, O/RM techniques are very powerful whenever the middle tier implementation relies on a direct object-oriented perspective of the relational model. In this particular context O/RM tools relieve programmers from most of the translation work between the two paradigms. CRUD-DOM is not tailored to tackle these situations. Its target is focused on middle tiers based on more complex CRUD expressions. Anyway, CRUD-M may be extended in other to provide an additional feature to automatically create the source code to execute the 4 basic SQL statements in each table: Select one row (by primary key), Insert one row, Update one row (by primary key) and Delete one row (by primary key).

Safe Query Objects [34] combine object-relational mapping with object-oriented languages to specify queries using strongly-typed objects and methods. They rely on Java Data Objects to provide strongly-typed objects and also to provide data persistence. Safe Query Objects are a promising technique to express queries but share most of the aforementioned drawbacks of O/RM, namely regarding performance and SQL expressiveness.

SQL DOM [35] generates a Dynamic Link Library containing classes that are strongly-typed to a database schema. These classes are used to construct dynamic SQL statements without manipulating any strings. As Safe Query Objects, SQL DOM does not take the full advantage of SQL expressiveness and also exhibits very poor results regarding performance.

Static Checking of Dynamically Generated Queries [36] presents a solution based on static string analysis of Java programs to find out where SQL statements are being constructed. The main idea is to find out all possible combinations of distinct SQL statements and then analyze them regarding their syntax and their type mismatch errors. This approach does not affect system performance but exhibits some drawbacks as: 1) all source code is hand written from string concatenation till JDBC execution context; 2) it does not provide any object-oriented view of the SQL statement execution context.

ADO.NET [9, 37] is part of the base class library included in the Microsoft .Net Framework. It is a set of classes that expose data access services to .NET programmers. The DataSet is the key component implementing a disconnected memory-resident representation of the data source. Some of the most important features are: it is aimed at integrating several and distinct data sources (XML, relational, etc.); it supports several related tables, constraints and relationships between them. The representation of the data source may be as complex as necessary. Therefore, ADO.NET is tailored to meet distinct requirements from those here announced. Table VI depicts the code for an ADO.NET example written in C#.

TABLE VI. ADO.NET EXAMPLE

```
// C#
void getStudent(int id) {
  string sql="select * from Std_student where Sd_id=" + id;
  SqlDataAdapter da=new SqlDataAdapter();
  da.SelectCommand=new SqlCommand(sql,conn);
  DataTable dt=DataTable();
  da.Fill(dt);
  DataRow dr=dt.Rows[0];
  string firstName=dr["Std_firstName"];
  string lastName=dr["Std_lastName"];
  Console.WriteLine("Student's name: " + firstName + " " + lastName);
}
```

Call Level Interfaces (CLI) [6], as JDBC [7] and ODBC [8] are practically an unavoidable option whenever performance and SQL expressiveness are simultaneously considered key issues. CLI provide mechanisms to encode Create, Read, Update and Delete SQL expressions inside strings, easily incorporating the performance and expressiveness of SQL. Thus, performance and expressiveness are crucial advantages of CLI but this comes with unavoidable and important drawbacks, namely there is no easy way to link CRUD expressions and the applications they assist; the act of edit CRUD expressions is tricky and error-prone; CRUD expressions are awkward regarding their maintenance and CRUD expressions are vulnerable to SQL injection attacks. In order to overcome the drawbacks of these techniques, we aim to explore CLI, namely through JDBC. These drawbacks and other issues will be thoroughly addressed in the next section. Table VII shows an example using JDBC.

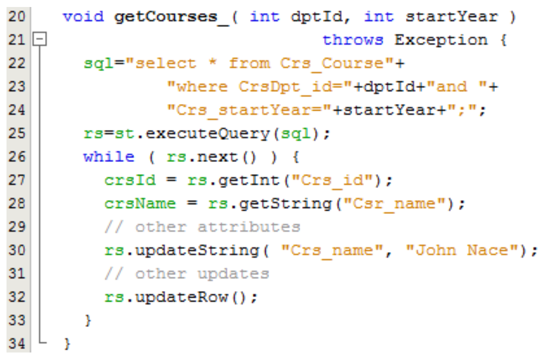
TABLE VII. JDBC EXAMPLE

```
// Java
void getStudent(Connection,conn, int id) throws SQLException {
  sql="select * " +
         "from Std_student " +
         "where std_student=" + id + ");";
  st=conn.createStatement();
  rs=st.executeQuery(sql);
  firstName=rs.getString("Std_firstName");
  lastName=rs.getString("Std_lastName");
  System.out.println("Student's name: " + firstName + " " + lastName);
}
```

III. IMPEDANCE MISMATCH: COMMON JDBC DRAWBACKS

JDBC is a common tool for integrating relational databases with Java programming language. JDBC is also a representative of the typical challenges of CLI. As such, we will explore JDBC as a target tool. Thus, this section aims to emphasize common drawbacks regarding the utilization of JDBC focusing mainly on the *ResultSet* interface. The drawbacks may be split into four categories: 1) the process for editing SQL statements; 2) the process for retrieving data from returned relations; 3) the process of updating databases through *CONCUR_UPDATABLE ResultSet*s; 4) protocols of *ResultSet* interface regarding its usability. Figure 2 presents a simple example, which comprises some of the drawbacks related to categories 1), 2) and 3). This example is used in the following paragraphs to describe JDBC drawbacks:

a) There is no easy way to link CRUD expressions and their results to the application they assist. CLI provide services to ease the integration of object-oriented applications and relational databases but relevant issues are not overcome as string concatenation (Figure 2: lines 22-24) and conversion between relational and object-oriented paradigms (Figure 2: lines 27, 28, 30).



```
20   void getCourses_( int dptId, int startYear )
21                                   throws Exception {
22     sql="select * from Crs_Course"+
23             "where CrsDpt_id="+dptId+"and "+
24             "Crs_startYear="+startYear+";";
25     rs=st.executeQuery(sql);
26     while ( rs.next() ) {
27       crsId = rs.getInt("Crs_id");
28       crsName = rs.getString("Csr_name");
29       // other attributes
30       rs.updateString( "Crs_name", "John Nace");
31       // other updates
32       rs.updateRow();
33     }
34   }
```

Figure 2. Some JDBC drawbacks

b) Editing CRUD expressions and access to their results is tricky and error-prone. CRUD expressions are constructed by concatenating strings and access to their results is achieved by reading attribute by attribute in a row by row basis. Some of the most usual errors are: a) concatenation errors - missing space between lines (Figure 2, lines 22, 23) and missing space before "and" (Figure 2: line 23); b) type mismatch error - argument *startYear* and column *Crs_startYear* (Figure 2: lines 20, 24); c) retrieving data - misspelled column name (Figure 2: line 28);

c) Errors cannot be checked for correctness at compile time, addressed in [36]. None of the previous errors can be caught at compile time demanding great accuracy while editing the source code in order to prevent additional time on testing, debugging and future maintenance.

d) CRUD expressions are awkward regarding their maintenance, addressed in [38]. CRUD expressions (construction and execution) comprise many different entities grouped in three classes: SQL syntax, CLI services and database schema. While SQL syntax and CLI services can be considered stable, database schema is a dynamic entity. Database schema may change for many reasons, as initial error on conceptual model or the emerging of new requirements, which usually happens several times during the development process and even also after application deployment. Any simple change in the database schema may involve a huge work on updating the strings that encode the affected SQL statements.

e) CRUD expressions are vulnerable to SQL injection attacks, addressed in [39]. This issue is not addressed in the current version of CRUD-DOM.

f) *ResultSet* usability, *ResultSet* interface has dozens of states, dealing with different combinations of *ResultSet* instantiations, directions, accesses, updates, etc. The developer is before a huge task to become aware of how to use the *ResultSet* interface. *ResultSet* interface comprises several distinct protocols not organized in interfaces, conveying the idea that everything is possible in anytime. ResultSet interface is composed by more than 200 methods and 10 attributes. Figure 3 presents a partial view of the *ResultSet* interface. Each *ResultSet* state has its own usage protocol gathering a subgroup of all methods of the ResultSet interface. Figure 4 depicts the most relevant protocols for this work: Read, Update, Insert and Delete actions. While Read and Delete protocols do not comprise a start and an end instruction, Update and Insert protocols always have a start instruction (implicitly for Update and explicitly for Insert) and an end instruction. Besides the starting and the ending instructions, the main issue for Update and Insert protocols is that the cursor cannot be moved from the current selected row while the protocol is being executed. If the cursor is moved from the selected row while the protocol is being executed, the protocol will be aborted and previous changes are discarded from the in-memory of the ResultSet. In order to overcome some of these difficulties we will present an approach where each protocol is executed through a dedicated interface improving this way ResultSet usability.

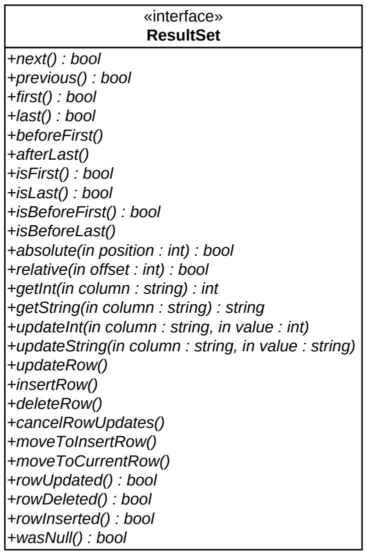

```
«interface»
ResultSet
+next() : bool
+previous() : bool
+first() : bool
+last() : bool
+beforeFirst()
+afterLast()
+isFirst() : bool
+isLast() : bool
+isBeforeFirst() : bool
+isBeforeLast()
+absolute(in position : int) : bool
+relative(in offset : int) : bool
+getInt(in column : string) : int
+getString(in column : string) : string
+updateInt(in column : string, in value : int)
+updateString(in column : string, in value : string)
+updateRow()
+insertRow()
+deleteRow()
+cancelRowUpdates()
+moveToInsertRow()
+moveToCurrentRow()
+rowUpdated() : bool
+rowDeleted() : bool
+rowInserted() : bool
+wasNull() : bool
```

Figure 3. Partial view of the *ResultSet* interface

Some of the aforementioned drawbacks have already been individually addressed as previously cited. In this paper we will present a simple, integrated and unified alternative to overcome all the aforementioned drawbacks, except for the SQL injection attack. The alternative comprises both the CRUD-DOM and the CRUD-M.

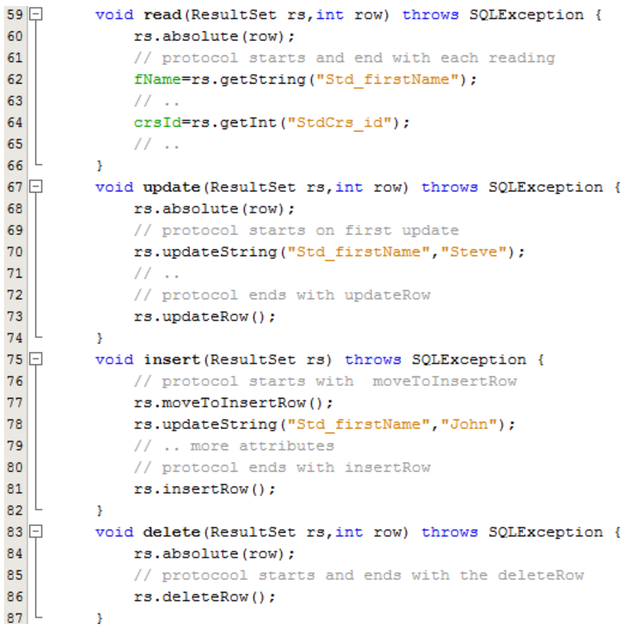

```
void read(ResultSet rs,int row) throws SQLException {
    rs.absolute(row);
    // protocol starts and end with each reading
    fName=rs.getString("Std_firstName");
    // ..
    crsId=rs.getInt("StdCrs_id");
    // ..
}
void update(ResultSet rs,int row) throws SQLException {
    rs.absolute(row);
    // protocol starts on first update
    rs.updateString("Std_firstName","Steve");
    // ..
    // protocol ends with updateRow
    rs.updateRow();
}
void insert(ResultSet rs) throws SQLException {
    // protocol starts with  moveToInsertRow
    rs.moveToInsertRow();
    rs.updateString("Std_firstName","John");
    // .. more attributes
    // protocol ends with insertRow
    rs.insertRow();
}
void delete(ResultSet rs,int row) throws SQLException {
    rs.absolute(row);
    // protocool starts and ends with the deleteRow
    rs.deleteRow();
}
```

Figure 4. Read, Update, Insert and Delete protocols

IV. CRUD-DOM

CRUD-DOM is our abstract model aimed at bridging the gap between relational databases and object-oriented applications. The CRUD-DOM goals are manifold, which were described in section I.B Motivation. Before we delve into the CRUD-DOM issue, we will present a concise overview of Statement/ResultSet interfaces and CRUD expressions.

A. Statement and ResultSet

The Statement interface [40] is used to execute SQL statements and to return the possible results they produce (only for Select statements). The returned results are managed by the ResultSet interface [41]. Loosely speaking, ResultSet interface provides two orthogonal functionalities: *scrollability* and *updatability*. Scrollability defines the ability to scroll over the rows retrieved from the database. There are two options: *forward only* – in this case cursors may only move forward one row at a time; *scrollable* – cursors may move in any direction and jump several rows at a time. Updatability defines the capacity to change the in-memory data managed by the ResultSet interface and therefore the content of the host database. There are two main possibilities: *read only* – the content of the ResultSet is read only and, therefore, no changes are allowed; *updatable* – changes may be performed over the in-memory data, as Insert, Update and Delete. These functionalities are defined at instantiation time of the parent Statement or PreparedStatement [42] object. The combination of these two functionalities influences the performance of many actions that are executed. This analysis will be carried out in section VI.

B. CRUD Expressions

CRUD expressions are the basic entities from which CRUD-DOM specification must evolve. Therefore, before proceeding with the CRUD-DOM specification, it is advisable to briefly survey CRUD expressions in order to be aware of the JDBC context in which they are used. CRUD expressions comprise the four basic SQL statements for accessing information in databases: *Select*, *Insert*, *Update* and *Delete*. While *Insert*, *Update* and *Delete* statements are used to alter the state of databases, *Select* statements allow the implementation of several views of the database. Hence, CRUD expressions may be grouped into two categories: "query CRUD expressions" (Q-CRUD) whenever involving a *Select* statement; and "execute CRUD expressions" (E-CRUD) whenever involving an *Insert*, *Update* or *Delete* statement. The corresponding CRUD-DOs share some source code but relevant differences must be emphasized. The most relevant difference is that Q-CRUD expressions return one or more relations from the database therefore requiring specific processing, as seen in Figure 2 (lines 26-28). Additionally, in some circumstances and also for certain Q-CRUD expressions it is possible to instantiate updatable *ResultSet*s. Updatable *ResultSet*s provide embedded protocols to update, to delete and to insert data in databases. Figure 2 (lines 30-32) concisely presents a case for the update situation. Other examples are presented in Figure 4 for Update, Insert and Delete actions. Thus, two types of CRUD expressions may be defined. Q-CRUD expressions executed on updatable *ResultSet* are named as Active Q-CRUD expressions (AQ-CRUD). Q-CRUD expressions executed on non-updatable *ResultSet* are named as Passive Q-CRUD expressions (PQ-CRUD).

C. CRUD-DOM Objectives

As previously mentioned CRUD-DOM addresses three main objectives: high performance, high usability and high productivity. In this section we will describe the most relevant features to be included and that are dependent on CRUD-DOM architecture: performance and usability (access layer usability). The remaining features as usability (CRUD Manager usability) and productivity depend on CRUD Manager.

Performance

To comply with the performance objective, the following features were established:

- <u>Pool of CRUD-DOs</u>: CRUD-DOs rely on statically crated CRUD expressions. CRUD-DOs exist inside the access layer and are supposed to be reused over and over again. Therefore, a pool of CRUD-DOs should minimize CRUD-DO instantiation time.
- <u>Prepared statements</u>: for the reasons pointed in the previous feature (reuse of CRUD-DOs), it is advisable to use prepared statements (PreparedStatement [42]) instead of Statements (Statement [40]).

Usability

To comply with the usability, the following features were established:

- <u>Type-state [43] oriented interfaces</u>: For each main ResultSet protocol (Read, Update, Insert and Delete) CRUD-DOM makes available a type-state oriented interface.
- <u>Semantic interfaces</u>: all interfaces defined by CRUD-DOM aimed to deal with query parameters and attributes of the returned relations are always semantically oriented. This means that the names of their methods and their arguments are always derived from the associate queries.
- <u>Factory</u>: from users' perspective, all CRUD-DOs are created and managed through a factory.

D. CRUD-DOM Details

We will present CRUD-DOM architecture by enumerating and describing the fundamental features for each type of CRUD expression: E-CRUD, PQ-CRUD and AQ-CRUD. Afterwards, we will present class diagrams for each type of CRUD expression. For all presented examples we assume that:

- "CruddoName" is the name for all types of CRUD expressions used as examples.
- Q-CRUD expression is "*select co1A*, *colB from table where colA>param*" where *colA* is *integer* and *colB* is *String*.
- E-CRUD is any *delete*, *update* or *insert* SQL statement with one parameter (*param*) of type integer.

All CRUD-DOs share the following features:

- Every CRUD-DO has a unique name.
- Every CRUD-DO is built around one class, known as the invocation class, and among other things, the class is responsible for the execution of the CRUD expression.
- The name of the invocation class is the same as the one given to the CRUD-DO.
- The invocation class has only one constructor with no arguments. Its visibility is protected.
- The invocation class has one method with the following signature *void config(Connection conn)*. This method is responsible for setting the connection to be used during the query execution.
- The invocation class has one method named *execute*, which is responsible for the execution of the CRUD expression. This method returns no value and has as many arguments as the number of the CRUD expression parameters. The name, type and order of the arguments depend on the name, type and order of CRUD expression parameters. For our example, *execute* has one parameter named as *param* and its type is *int*.

All CRUD-DOs derived from E-CRUD expressions share the following feature:

- The invocation class has a method with the following signature: *int getAffectedRows()*; this method returns the number of rows affected by the execution of the E-CRUD expression.

Figure 5 presents the class diagram for the E-CRUD expression example.

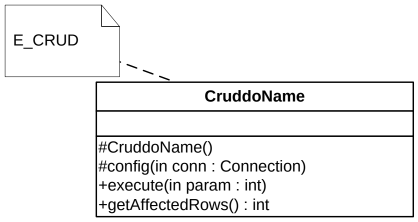


Figure 5. Class diagram for E-CRUD expressions

All CRUD-DOs derived from Q-CRUD expressions share the following feature:

- The invocation class must implement all the scrollable methods in accordance to its instantiation criterion.
- If the ResultSet is scrollable, provide a method with the following signature: *int getRowCount()*; this method returns the number of rows retrieved by the Select statement;
- Q-CRUD expressions have no concrete instances. They are super types for PQ-CRUD and AQ-CRUD expressions.

All CRUD-DOs derived from PQ-CRUD expressions share the following features:

- Extend features of Q-CRUD expressions;
- The invocation class has one method with the following signature: *CruddoName_readTuple beginRead();*
- *CruddoName_readTuple* class, known as the access class, implements one method, generally known as access method, for each attribute of the returned relation. Each access method has the following signature *javaDataType gAttributeName()* where *JavaDataType* is the correspondent java data type for SQL data type and the method's name is built by concatenating the name of the attribute (first letter converted to uppercase) with the prefix *g*.

Figure 6 presents the class diagram for the PQ-CRUD example.

All CRUD-DOs derived from AQ-CRUD expressions share the following features:

- Extend features of Q-CRUD expressions;
- The invocation class may provide any subset of the following four features: *readable*, *updatable*, *insertable* and *deletable*; whenever provided, the *readable* feature may also be included in the remaining features to improve their usability;
- If CRUD-DO is *readable* it implements one method with the following signature: *CruddoName_readTuple beginRead()*;
- If CRUD-DO is *updatable* it implements one method with the following signature: *CruddoName_updateTuple beginUpdate( )*;
- If CRUD-DO is *insertable* it implements one method with the following signature: *CruddoName_insertTuple beginInsert( )*;
- If CRUD-DO is *deletable* it implements one method with the following signature: *void delete()*;
- *CruddoName_readTuple* class: previously explained for PQ-CRUD;
- *CruddoName_updateTuple* and *CruddoName_insertTuple* classes provide functionalities easily perceived from *CruddoName_readTuple* class: access methods have *s* as prefix instead of *g*;.
- The *delete* method, deletes the current row from the *ResultSet*.

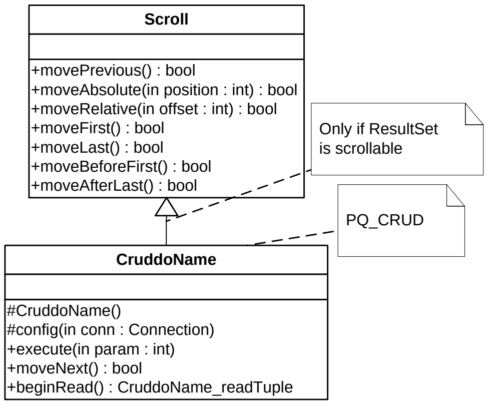
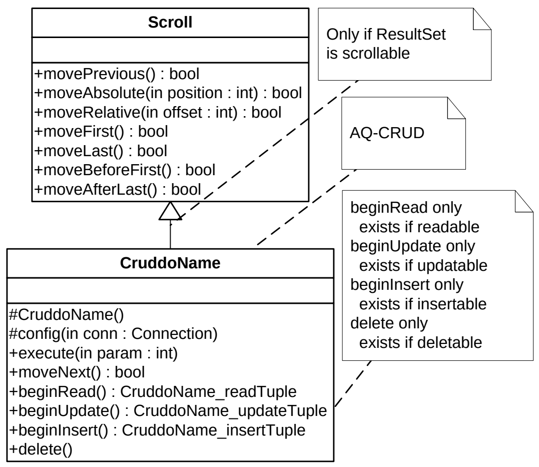

Scroll
+movePrevious() : bool
+moveAbsolute(in position : int) : bool
+moveRelative(in offset : int) : bool
+moveFirst() : bool
+moveLast() : bool
+moveBeforeFirst() : bool
+moveAfterLast() : bool
Only if ResultSet is scrollable
PQ_CRUD
CruddoName
#CruddoName()
#config(in conn : Connection)
+execute(in param : int)
+moveNext() : bool
+beginRead() : CruddoName_readTuple

Figure 6. Class diagram for PQ-CRUD expressions

Figure 7, Figure 8, Figure 9 and Figure 10 present the class diagrams for AQ-CRUD expressions.

Class diagrams have been presented for each type of CRUD expression. To completely understand the class diagrams it is necessary to have an understanding of how the *ResultSet* interface is implemented. Original *ResultSet* method names have been renamed and some new ones have been included. Renamed methods are easily identified: *next->moveNext*, *previous->movePrevious*, etc. Only a subgroup of all methods has been presented in order to avoid overcrowding the class diagrams.

There is a factory responsible for creating the correct instances and also for managing the pool of CRUD-DO instances. Figure 11 depicts the factory source code for managing CruddoName. After their utilization, CRUD-DOs may be released for future reutilization (Figure 11, line 25) maintaining this way an active pool of CRUD-DOs. Before creating a new instance, the factory checks if there is any instance available inside the pool (Figure 11, line 31).



Scroll
+movePrevious() : bool
+moveAbsolute(in position : int) : bool
+moveRelative(in offset : int) : bool
+moveFirst() : bool
+moveLast() : bool
+moveBeforeFirst() : bool
+moveAfterLast() : bool
Only if ResultSet is scrollable
AQ-CRUD
CruddoName
#CruddoName()
#config(in conn : Connection)
+execute(in param : int)
+moveNext() : bool
+beginRead() : CruddoName_readTuple
+beginUpdate() : CruddoName_updateTuple
+beginInsert() : CruddoName_insertTuple
+delete()
beginRead only exists if readable
beginUpdate only exists if updatable
beginInsert only exists if insertable
delete only exists if deletable

Figure 7. Class diagram for AQ-CRUD expressions

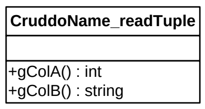


Figure 8. Readable class diagram for Q-CRUD expressions

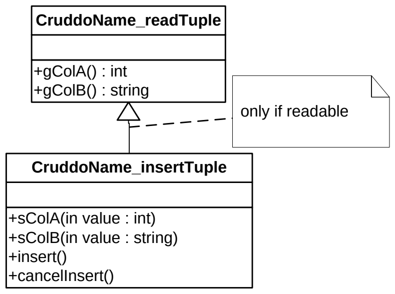


Figure 9. Insertable class diagram for AQ-CRUD expressions

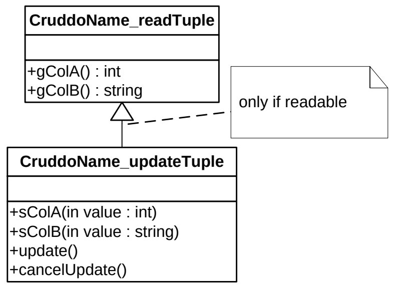


Figure 10. Updatable class diagram for AQ-CRUD expressions

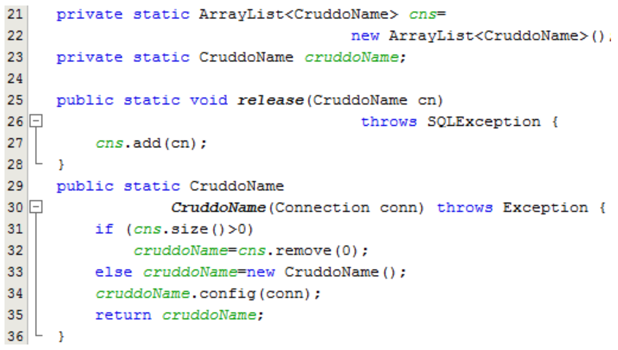

```
21  private static ArrayList<CruddoName> cns=
22                           new ArrayList<CruddoName>();
23  private static CruddoName cruddoName;
24
25  public static void release(CruddoName cn)
26                                throws SQLException {
27      cns.add(cn);
28  }
29  public static CruddoName
30            CruddoName(Connection conn) throws Exception {
31      if (cns.size()>0)
32         cruddoName=cns.remove(0);
33      else cruddoName=new CruddoName();
34      cruddoName.config(conn);
35      return cruddoName;
36  }
```

Figure 11. Factory: pool management

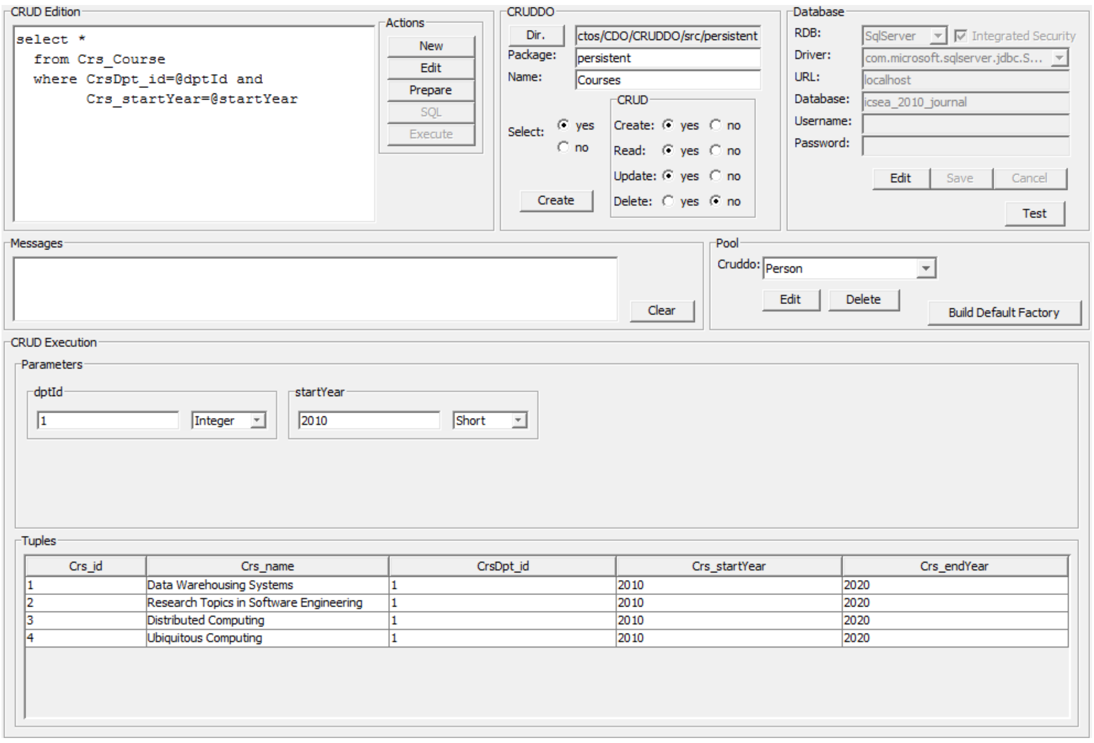
V. CRUD MANAGER

CRUD-M addresses productivity objectives (automatic code generation, semi-automatic test and also maintenance) and usability objectives. No special programming skills should be required to use CRUD-M and learning time should be minimal. CRUD-M usage is centered in a GUI component presented in Figure 12. Figure 12 shows a concrete example for an AQ-CRUD expression, called *Courses*, which was created as readable, updatable and insertable but not deletable. Figure 13 shows the usage of CRUD-DO *Courses* from the application tier point of view. As one can see, the integration is seamless regarding impedance mismatch. Additionally, an approach for the implementation of *ResultSet* as a typestate [44] component is provided improving this way CRUD-DO usability. This may be verified, as an example, by the definition of the *Courses_readTuple* interface (Figure 13, lines 68, 69), which provides a coherent protocol for retrieving data from the *ResultSet*.

CRUD-M encompasses five main blocks as depicted in Figure 14. User launches CRUD-M and defines which database is going to be used. Then, “Schema Reader” reads the schema of the database. From now on, users may edit and/or maintain CRUD expressions. “CRUD Editor” provides a context where CRUD expressions may be edited. “CRUD Execution Unit” may help “CRUD Editor” in some specific tasks as defining SQL parameters and executing statements against the database. After executing successfully an SQL statement against the database, users are allowed to create CRUD-DO, which will be accomplished by “CRUD-DO Generator”. “CRUD Maintenance” parses CRUD-DO and retrieves the underlying CRUD expression to be reedited by “CRUD Editor”. A more detailed description for each bock follows:

Schema Reader: this component reads the schema of the database, which is mainly used to automatically suggest the Java data types for the parameters of CRUD expressions.

CRUD Editor: CRUD Editor is a text editor where CRUD expressions may be written from scratch. Parameters defined in runtime must be identified through a unique name preceded by a ‘@’ character. These names will be used for



```
select *
  from Crs_Course
  where CrsDpt_id=@dptId and
        Crs_startYear=@startYear
```

Package: persistent
Dir.: ctos/CDO/CRUDDO/src/persistent
Name: Courses
Select: [✓ yes] no
CRUD
Create: [✓ yes] no
Read: [✓ yes] no
Update: [✓ yes] no
Delete: yes [✓ no]

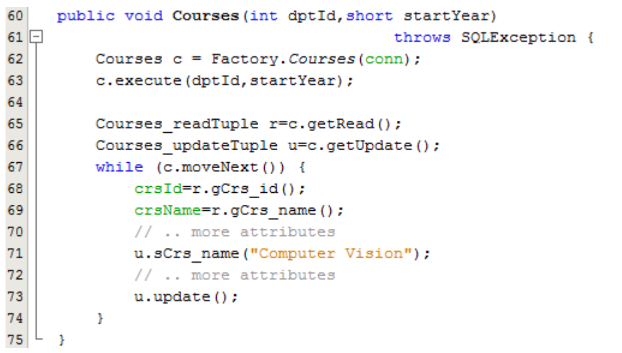
RDB: SqlServer
Integrated Security ✓
Driver: com.microsoft.sqlserver.jdbc.S...
URL: localhost
Database: icsea_2010_journal
Username:
Password:

Cruddo: Person

dptId: 1 Integer
startYear: 2010 Short

Crs_id	Crs_name	CrsDpt_id	Crs_startYear	Crs_endYear
1	Data Warehousing Systems	1	2010	2020
2	Research Topics in Software Engineering	1	2010	2020
3	Distributed Computing	1	2010	2020
4	Ubiquitous Computing	1	2010	2020

Figure 12. CRUD-M GUI



```
public void Courses(int dptId,short startYear)
                                throws SQLException {
    Courses c = Factory.Courses(conn);
    c.execute(dptId,startYear);

    Courses_readTuple r=c.getRead();
    Courses_updateTuple u=c.getUpdate();
    while (c.moveNext()) {
        crsId=r.gCrs_id();
        crsName=r.gCrs_name();
        // .. more attributes
        u.sCrs_name("Computer Vision");
        // .. more attributes
        u.update();
    }
}
```

Figure 13. *Courses* from the application point of view

the arguments of the *execute* method of the invocation classes. In our example we have defined two parameters: *dptId* and *startYear*.

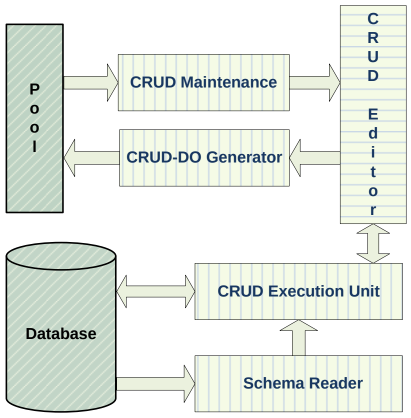


Figure 14. Block diagram of the CRUD-M

CRUD Execution Unit: CRUD Execution Unit is responsible for three tasks: 1) providing, whenever necessary, input data components for SQL parameters. Each input component is identified by the name of the associated parameter and has a default Java Data Type derived from the database schema. Users may select another Java Data Type becoming responsible for their decision; 2) executing the edited CRUD expression against the database proving this way an expedite and integrated tool for evaluating the correctness of CRUD expressions and also for testing the outcome of CRUD expressions. Developers are relieved to write source code to test and debug their CRUD expressions; 3) formatting a table in runtime to present the content of returned relations, whenever the underlying CRUD is a Q-CRUD expression. This visualization allows developers to have an immediate visual feedback about the retuned data and easily evaluate the outcome of Q-CRUD expressions execution. In our example, the returned relation has 4 rows and 5 attributes.

CRUD-DO Generator: CRUD-DO Generator creates automatically all the necessary source code for the underlying CRUD expressions. For all types of CRUD expressions, users must input some additional information, as: CRUD-DO's name, package's name, type of CRUD expression, pool directory for CRUD-DOs, etc. Some additional information is required if the CRUD expression is of type AQ-CRUD, as which of the following functionalities should be implemented: *readable*, *insertable*, *updatable* and *deletable*.

CRUD Maintenance: this component keeps track of all existing CRUD-DOs in the pool directory. Any CRUD-DO in the pool directory may be selected for editing or to be deleted. If it is selected for editing, the underlying CRUD expression is retrieved from the invocation class and presented in the CRUD editor. From now on, the CRUD expression may be retested or reedited to update the current CRUD-DO or even to create a new one.

VI. Performance Assessment

As mentioned in section I, two dimensions may be considered for performance: responsiveness and scalability. Hereafter, performance should be understood as the responsiveness dimension. The first version of CRUD-DOM was presented in [1]. CRUD-DOM performance was evaluated by measuring the responsiveness for a particular situation: a fixed block of code (one for each protocol) was repeatedly executed for a specific number of times. In this new approach, we will get a more dynamic view about how CRUD-DOM and JDBC behave. This will be achieved by stressing them under several conditions. Details are explained in the next sub-sections.

A. *The valuation testbed*

All measurements share the same platform: PC - Dell Latitude E5500; CPU - Intel Duo Core P8600 @2.40GHz; RAM - 4.00 GB; OS - Windows Vista Enterprise Service Pack 2 (32bits); Java SE 6; JDBC(sqljdbc4) and SQL Server 2008 version 10.0.1600.22. In order to promote an ideal environment the following actions were taken: the running threads were given the highest priority and all non-essential processes/services were canceled. Transactions were not used and *auto-Commit* was used in all connections.

A new database was created in conformance with the schema presented in Figure 1. In order to avoid some overhead added by SQL Server, some default properties were changed as, Auto Update Statistics = false and Recovery Model=Simple.

The performance assessment addresses two goals: the first one, known as standard JDBC assessment (S-JDBC), is to understand the behavior of the standard Statement and ResultSet interfaces; the second one, based on a component relying on CRUD-DOM (C-CRUD), aims to assess C-CRUD and compare it with S-JDBC. S-JDBC and C-CRUD are from now on generally known as entities and formally represented by the letter *E*.

Part of the results of both assessments is influenced by the Microsoft TDS protocol and also by the implemented mechanisms on both sides (JDBC and SQL Server) to support it. Some key notes are provided to help on the understanding of the collected results:

- selected data through forward-only and read-only ResulSets are always transferred to the client side in a single batch. Sql Server does not implement any mechanism to supervise or control client behavior. On the other side, for other types of ResultSets, Sql Server transfers data in blocks and keeps track of clients' operations. This is achieved by a cursor and a dataset that keeps all the selected data and also keeps track on which row clients are pointing to. This means that it is expected that forward-only and read-only ResultSets should get better performance results than the other types of ResultSets.
- forward-only ResultSets require a simpler mechanism to scroll over the selected data. This means that JDBC and Sql Server have optimized algorithms and therefore improved performance for forward-only ResultSets.
- Read-only ResultSets do not create, explicitly, any concurrency constraint on the database and, therefore, their implementations are more effective on both sides.
- Scrollable and updatable ResultSets are expected to have the worst performance. They are the sum the most complex implementations of TDS: not forward-only and nor read-only.

The size of blocks to be retrieved from the Sql Server may be controlled by setting the block fetch size. Thus, in order to impose a similar environment to all the collected measures, the fetch size has been set to guaranty that all rows are retrieved from Sql Server in a single block.

Sql Server supports more ResultSet types than those defined in the standard JDBC. A more detailed description about Microsoft implementation of JDBC may be found here [45].

In [1], the context in which the assessment took place was characterized by: 1) the type of Statement {Forward-only Read-only (FR), Forward-only Updatable (FU), Scrollable Read-only (SR) and Scrollable Updatable (SU)}; 2) the type of operation {Read (R), Update (U), Insert (I) and Delete (D)} and finally 3) by defining a normalized metric based on the number of cycles that was possible to compute in a second. In spite of its simplicity and validity, we have adopted a new strategy that provides a better evaluation for both entities.

The environment in which the assessment here presented took place is characterized by: CRUD expression, scenarios, contexts, units and data. These items are explained in the next paragraphs.

CRUD expressions: All measurements derive from the AQ-CRUD expression "*Select * from Std_student*".

Scenarios (S): Four scenarios were defined for each operation to be evaluated: Read (S_{re}), Update (S_{up}, S_{cu}), Insert (S_{in}, S_{ci}) and Delete (S_{de}). The S_{re} consists in select a certain number of tuples from the database and then read all attributes of all tuples. The update scenario comprises two variants: a) S_{up} consists in selecting a certain number of tuples from the database and then update all attributes (except the primary key – Std_id) of all tuples without committing the changes to the database; b) S_{cu} consists in selecting a certain number of tuples from the database and then update all attributes (except the primary key – Std_id) of all tuples and commit the changes to the database. The insert scenario comprises two variants: a) the S_{in} consists in selecting zero rows from the database and then insert all attributes of a certain number of rows into the ResultSet without committing them to the database; b) the S_{ci} consists in selecting zero rows from the database and then insert all attributes of a certain number of rows into the ResultSet committing them to the database. The S_{de} scenario consists in select a certain number of tuples from the database and then to delete all tuples one by one. Table VIII concisely describes all scenarios. These scenarios are only one possibility among an infinity of others. Thus, it was decided to only assess S-JDBC in these scenarios because the most relevant assessment is carried out for the individual units (see Units).

TABLE VIII. FORMAL DESCRIPTION OF ALL SCENARIOS

S	Description
S_{re}	Delete all tuples from the table Std_Student Insert *n* tuples into the table Std_Student Start clock Select all *(n)* tuples from the table Std_Student For each tuple Read all attributes Stop clock
S_{up}	Delete all tuples from the table Std_Student Insert *n* tuples into the table Std_Student Start clock Select all(*n*) tuples from the table Std_Student For each tuple Update all attributes except the pk // without committing them Stop clock
S_{cu}	Delete all tuples from the table Std_Student Insert *n* tuples into the table Std_Student Start clock Select all(*n*) tuples from the table Std_Student For each tuple Update all attributes except the pk Commit changes Stop clock
S_{in}	Delete all tuples from the table Std_Student Start clock Select all (*zero*) tuples from the table Std_Student While insert more tuples Insert all attributes // without committing them Stop clock
S_{ci}	Delete all tuples from the table Std_Student Start clock Select all (*zero*) tuples from the table Std_Student While insert more tuples Insert all attributes Commit new tuple Stop clock
S_{de}	Delete all tuples from the table Std_Student Insert *n* tuples into the table Std_Student Start clock Select *n* tuples from the table Std_Student For each tuple Delete tuple Stop clock

Unit (U): A unit is a task whose execution time is relevant to understand the behavior of any of the four scenarios. The following units were defined: time to execute the select statement (U_{se}), time to read all returned tuples (U_{re}), time to update all returned tuples but not to commit them to the database (U_{up}), time to insert tuples into the ResultSet but not to commit them (U_{in}), time to update tuples and commit them to the database (U_{uc}), time to insert tuples and commit them to the database (U_{ic}) and finally time to delete tuples from the database (U_{de}). Table IX concisely describes all units. Each scenario may be seen as an aggregation of individual units.

Now let's present the composition for each scenario in terms of units: $S_{re}=U_{se}+U_{re}$, $S_{up}=U_{se}+U_{up}$, $S_{cu}=U_{se}+U_{uc}$ $S_{in}=U_{se}+U_{in}$, $S_{ci}=U_{se}+U_{ic}$ and $S_{de}=U_{se}+U_{de}$.

TABLE IX. FORMAL DESCRIPTION OF ALL UNITS

C	Description
U_{se}	Delete all tuples from the table Std_Student Insert *n* into the database table Std_Student Start clock Select all *(n)* tuples from the table Std_Student Stop clock
U_{re}	Delete all tuples from the table Std_Student Insert *n* into the table Std_Student Select all *(n)* tuples from the table Std_Student Start clock For each tuple Read all attributes Stop clock
U_{up}	Delete all tuples from the table Std_Student Insert *n* tuples into the table Std_Student Select all(*n*) tuples from the table Std_Student Start clock For each tuple Update all attributes without commit (except the pk) Stop clock
U_{cu}	Delete all tuples from the table Std_Student Insert *n* tuples into the table Std_Student Select all(*n*) tuples from the table Std_Student Start clock For each tuple Update all attributes (except the pk) Commit changes Stop clock
U_{in}	Delete all tuples from the table Std_Student Select all (*zero*) tuples from the table Std_Student Start clock While insert more tuples Insert all attributes without commit Stop clock
U_{ci}	Delete all tuples from the table Std_Student Select all (*zero*) tuples from the table Std_Student Start clock While insert more tuples Insert all attributes Commit new tuple Stop clock
U_{de}	Delete all tuples from the table Std_Student Insert *n* tuples into the table Std_Student Select all (*n*) tuples from the table Std_Student Start clock For each tuple Delete tuple Stop clock

Context (C): Four contexts were defined for the Statement interface: forward-only and read-only (C_{fr}), forward-only and updatable (C_{fu}), scrollable and read-only (C_{sr}) and finally scrollable and updatable (C_{su}). All contexts were used to explicitly assess S-JDBC. C-CRUD was only assessed in the C_{su}. The justification for this option is that S-JDBC and C-CRUD architectures do not depend on the running context. The collected differences between S-JDBC and C-CRUD in one context should be equivalent in all the other contexts. This means that if for C_{su} the difference between C-CRUD and S-JDBC is Δt then it will remain Δt for the other contexts. Therefore, the behavior of C-CRUD for the remaining contexts may be inferred from the behavior of S-JDBC in those contexts and from the collected differences between S-JDBC and C-CRUD in C_{su}. This assertion has been confirmed by several collected measurements in the other remaining contexts. Table X describes all contexts.

TABLE X. DESCRIPTION OF ALL CONTEXTS

C	Description
C_{fr}	Forward-only and read-only
C_{fu}	Forward-only and updatable
C_{sr}	Scrollable and read-only
C_{su}	Scrollable and updatable

Data: In order to promote a dynamic view about the behavior of each entity, it was decided not to measure the number of cycles that is possible to be computed in a second but to measure the required time to execute each scenario/unit for a set of numbers of rows. The chosen set of numbers of rows is: 5, 10, 15, 25, 50, 75, 100, 150, 250, 350 and 500 rows. This approach gives a dynamic perspective about the behaviors of all entities and is applied to all scenarios, contexts and units. A simple formalization of both entities may be expressed as:

$$S(\alpha,\eta) \quad (1)$$

$$U(\alpha,\eta) \quad (2)$$

Table XI describes each symbol of equations (1) and (2).

TABLE XI. MEANING OF EQUATIONS (1) AND (2)

	Description	Domain
S	Any subset of all scenarios. All scenarios is represented by S_{all}.	$S \in \{S_{all},S_{re},S_{up},S_{cu}, S_{in},S_{ci},S_{de}\}$
U	Any subset of all units (valid for the defined scenario and context). All units is represented by U_{all}.	$U \in \{U_{all},U_{se},U_{sc},U_{re},U_{up},U_{cu},U_{in}, U_{ci},U_{de}\}$
α	Any subset of all contexts. All contexts are represented by c_{all}.	$\alpha \in \{c_{all},c_{fr},c_{fu},c_{sr},c_{su}\}$
η	Any subset of the set of rows. All set is represented by n_{all}.	$\eta \in \{n_{all},n_5,n_{10},n_{15},n_{25},n_{50},n_{75}, n_{100},n_{150},n_{250},n_{350},n_{500}\}$.

Example, $S_{de}(c_{fu,su},n_{all})$ means: scenario delete, contexts forward-only updatable and scrollable updatable and the complete set of rows.

A slot is defined as the minimum granularity for which it is necessary to collected measures. Examples: $S_{re}(c_{fr},n_5)$, $S_{re}(c_{fr},n_{l0})$ and $U_{up}(c_{fu,su},n_{250})$. The distribution and the total number of different slots are presented in Table XII. The number of slots for S-JDBC for all scenarios is computed by multiplying the number of scenarios by the number of contexts by the number of sets of rows. The other values follow the same reasoning to be computed. The total number of slots for both entities is 649.

TABLE XII. NUMBER OF SLOTS

	S-JDBC	C-CRUD	Total
Scenarios	6x4x11=264	0	264
Units	7x4x11=308	7x1x11=77	385
Total	572	77	649

The measures used in all the following graphics for each slot were computed, as:

- At least 500 raw measures were collected.
- The 25 best raw measures were discarded.
- Measure=average of the 50 best remaining raw measures.

Thus, at least 649x500=324,500 raw measures were collected for this current assessment.

B. S- JDBC assessment

S-JDBC assessment comprises both the units and the scenarios. The assessment of units allows us to analyze and isolate the impact of each context by unit. The assessment of scenarios also allows us to analyze the impact by context but the simulation in closer to real situations because the starting point is always triggered by a *select* statement. Just to remind, AQ-CRUD expressions always comprise a Select statement.

S-JDBC assessment is carried out without any special architecture, avoiding this way any additional overhead. This will be confirmed in the following paragraphs.

1) Assessment of units

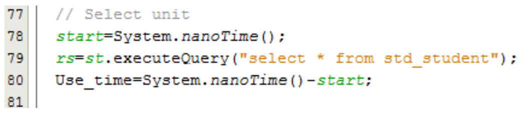
In section IV.A it was mentioned that each context (combination of functionalities) may influence the performance of each operation. In this section we will analyze the impact of the chosen contexts in each unit.

Figure 15, Figure 18, Figure 21, Figure 23, Figure 25, Figure 27 and Figure 30 depict the source code for each unit. Each unit is individually controlled in order to collect accurate measures for its execution time. These figures show that: the source code is exempt of any architecture and the source code is in line with the general description of all units, see Table IX. These units have some modifications when compared to the equivalent ones presented in [1] derived from the changes introduced in the current test-bed.

Figure 16, Figure 17, Figure 19, Figure 20, Figure 22, Figure 24, Figure 26, Figure 28, Figure 29, Figure 31 and Figure 32 show the performance of all units. The column bars represent the required time to execute the unit (T_{sj}) and the associated vertical axis is the left one. The dashed lines represent the required mean time to process one row (R_{sj}) and the associated vertical axis is the right one. R_{sj} is computed dividing T_{sj} by η.

U_{se} – Select unit

Figure 15 depicts the main source code for the $U_{se}(c_{all},n_{all})$. This unit is focused on measuring the required time to select each set of number of rows.



```
// Select unit
start=System.nanoTime();
rs=st.executeQuery("select * from std_student");
Use_time=System.nanoTime()-start;
```

Figure 15 . S-JDBC: source code for the $U_{se}(c_{all},n_{all})$.

Figure 16 presents the general behavior of the $U_{se}(c_{all},n_{all})$. The dashed lines are very close conveying the need for a more detailed graphic. Figure 17 presents a more detailed view of Figure 16 emphasizing the behavior of each context.

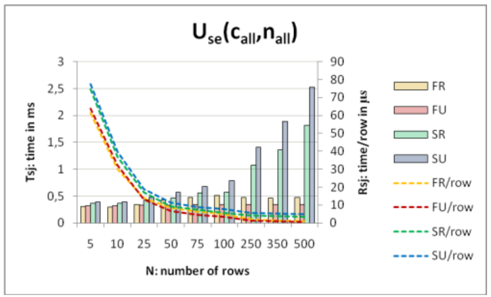


Figure 16. S-JDBC: behavior of $U_{se}(c_{all},n_{all})$

From Figure 16 and Figure 17 we may conclude that:

- T_{sj} of $C_{fr,fu}$ is weakly dependent on η.
- T_{sj} of $C_{sr,su}$ increases with η.
- R_{sj} decreases for all contexts when η increases; as an example, for C_{su}, T_{sj} varies from 77µs till 5.2µs.
- C_{fu}, C_{fr}, C_{sr} and C_{su} are ordered from the best to the worst R_{sj} score.

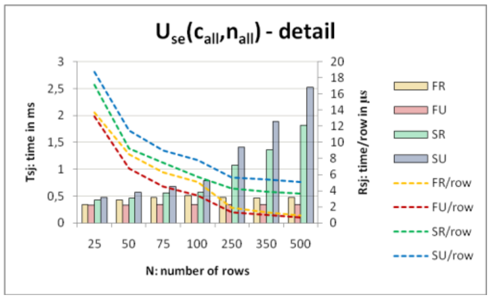


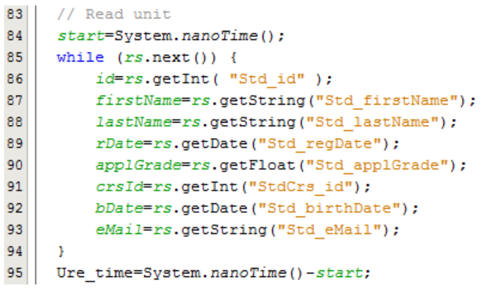
Figure 17. S-JDBC: detail of $U_{se}(c_{all},n_{all})$.

Main point: scrollable ResultSets should be avoided whenever possible, mainly when the number of rows is above 100.

U_{re} – Read unit

Figure 18 depicts the main source code for the $U_{re}(c_{all},n_{all})$. This unit is focused on measuring the required time to read all rows returned by the select statement.

Figure 19 presents the general behavior of the $U_{re}(c_{all},n_{all})$. The dashed lines for $C_{fu,sr,su}$ are very close conveying the need for a more detailed graphic. Figure 20 presents a more detailed view of Figure 19 emphasizing the behavior of the 4 contexts.



```
// Read unit
start=System.nanoTime();
while (rs.next()) {
    id=rs.getInt( "Std_id" );
    firstName=rs.getString("Std_firstName");
    lastName=rs.getString("Std_lastName");
    rDate=rs.getDate("Std_regDate");
    applGrade=rs.getFloat("Std_applGrade");
    crsId=rs.getInt("StdCrs_id");
    bDate=rs.getDate("Std_birthDate");
    eMail=rs.getString("Std_eMail");
}
Ure_time=System.nanoTime()-start;
```

Figure 18. S-JDBC: source code for the $U_{re}(c_{all},n_{all})$.

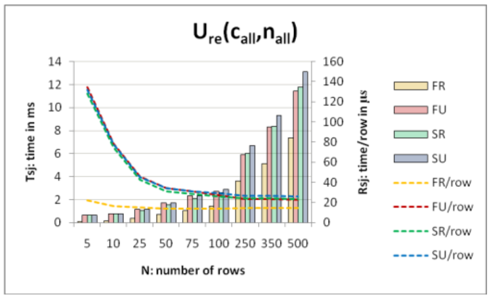


Figure 19. S-JDBC: behavior of $U_{re}(c_{all},n_{all})$.

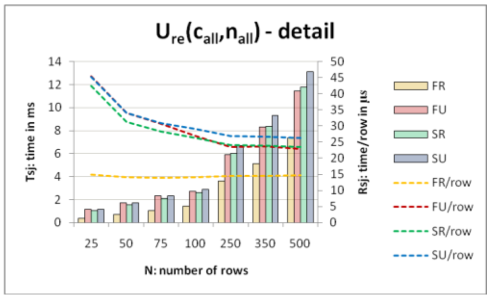


Figure 20. S-JDBC: detail of $U_{re}(c_{all},n_{all})$.

From Figure 19 and Figure 20 we may conclude that:

- T_{sj} of C_{all} increase with η; the C_{fr} is the most independent one.
- R_{sj} decreases for $C_{fu,sr,su}$ when η increases; as an example, for C_{su} the ratio varies from 135µs till

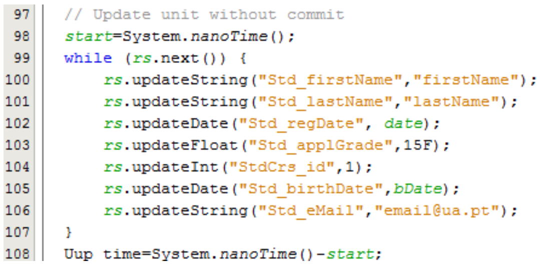
26µs. C_{fr} is independent of η revealing a constant performance.

- C_{fr} has by far the best scores for all η; $C_{fu,sr,su}$ present similar scores for all η.

Main point: C_{fr} is always the best option.

U_{up} – update unit without commit

Figure 21 depicts the main source code for the $U_{up}(c_{fu,su}, n_{all})$. This unit is focused on measuring the required time to update all rows returned by the select statement but without committing those changes to the database.



```
// Update unit without commit
start=System.nanoTime();
while (rs.next()) {
    rs.updateString("Std_firstName","firstName");
    rs.updateString("Std_lastName","lastName");
    rs.updateDate("Std_regDate", date);
    rs.updateFloat("Std_applGrade",15F);
    rs.updateInt("StdCrs_id",1);
    rs.updateDate("Std_birthDate",bDate);
    rs.updateString("Std_eMail","email@ua.pt");
}
Uup time=System.nanoTime()-start;
```

Figure 21. S-JDBC: source code for the $U_{up}(c_{fu,su}, n_{all})$.

Figure 22 presents the general behavior of the $U_{up}(c_{fu,su}, n_{all})$. The dashed lines for $C_{fu,su}$ are overlapped showing that the behaviors are the same for both contexts.

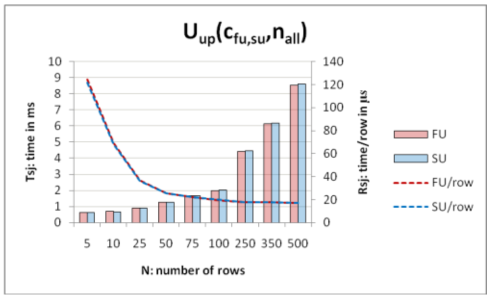


Figure 22. S-JDBC: behavior of $U_{up}(c_{fu,su}, n_{all})$.

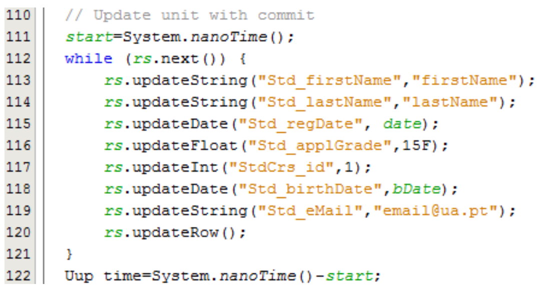
From Figure 22 we may conclude that:

- T_{sj} of $C_{fu,su}$ increases with η;
- The behavior is the same for both contexts.
- R_{sj} decreases for $C_{fu,su}$ when η increases; it ranges from 127µs till 18µs.
- For η>50, R_{sj} tends to be constant

Main point: there is no difference between C_{fu} and C_{su}.

U_{cu} – update unit with commit

Figure 23 depicts the main source code for the $U_{cu}(c_{fu,su}, n_{all})$. This unit is focused on measuring the required time to update all rows returned by the select statement and also for committing those changes to the database.



```
// Update unit with commit
start=System.nanoTime();
while (rs.next()) {
    rs.updateString("Std_firstName","firstName");
    rs.updateString("Std_lastName","lastName");
    rs.updateDate("Std_regDate", date);
    rs.updateFloat("Std_applGrade",15F);
    rs.updateInt("StdCrs_id",1);
    rs.updateDate("Std_birthDate",bDate);
    rs.updateString("Std_eMail","email@ua.pt");
    rs.updateRow();
}
Uup time=System.nanoTime()-start;
```

Figure 23. S-JDBC: source code for the $U_{cu}(c_{fu,su}, n_{all})$.

Figure 24 presents the general behavior of the $U_{cu}(c_{fu,su}, n_{all})$. The dashed lines for $C_{fu,su}$ are overlapped when η>=10, showing that the behaviors are practically the same for both contexts.

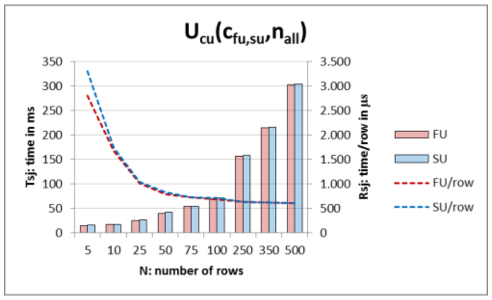


Figure 24. S-JDBC: behavior of $U_{cu}(c_{fu,su}, n_{all})$.

From Figure 24 we may conclude that:

- T_{sj} of $C_{fu,su}$ increases with η;
- For η>=10, the behavior is the same for both contexts.
- R_{sj} decreases for $C_{fu,su}$ when η increases; it ranges from about 3,000µs till 600µs.
- For η>=50, T_{sj} tends to be constant.

From Figure 22 and Figure 24 we conclude that committing the changes to the database causes a significant increase in T_{sj} and R_{sj} in about 25 times for all η. This means that any improvement in CRUD-DOM will very probably convey a minor effect in $C_{fu,su}$.

Main point: for η>=10, there is no difference between C_{fu} and C_{su}.

U_{in} – insert unit without commit

Figure 25 depicts the main source code for the $U_{in}(c_{fu,su}, n_{all})$. This unit is focused on measuring the required time to insert η tuples into the ResultSet but without committing them to the database.

Figure 26 presents the general behavior of the $U_{in}(c_{fu,su}, n_{all})$. The dashed lines for $C_{fu,su}$ are always overlapped showing that the behaviors are the same for both contexts.

```
// Insert unit without commit
start=System.nanoTime();
while ( n-- > 0) {
    rs.moveToInsertRow();
    rs.updateString("Std_firstName","firstName");
    rs.updateString("Std_lastName","lastName");
    rs.updateDate("Std_regDate", date);
    rs.updateFloat("Std_applGrade",15F);
    rs.updateInt("StdCrs_id",1);
    rs.updateDate("Std_birthDate",bDate);
    rs.updateString("Std_eMail","email@ua.pt");
}
Uin_time=System.nanoTime()-start;
```

Figure 25. S-JDBC: source code for the $U_{in}(c_{fu,su},n_{all})$.

From Figure 26 we may conclude that:

- T_{sj} of $C_{fu,su}$ increases with η.
- The behavior is the same for both contexts.
- R_{sj} decreases for $C_{fu,su}$ when η increases; it ranges from 128µs till 19µs.
- For η >=50, T_{sj} tends to be constant.

Main point: there is no difference between C_{fu} and C_{su}.

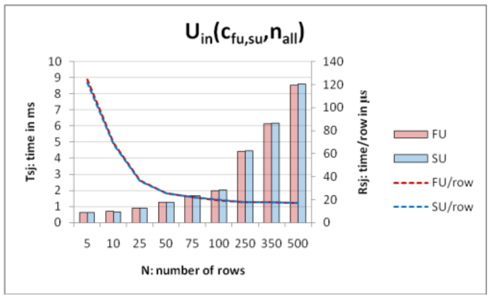


Figure 26. S-JDBC: behavior of $U_{in}(c_{fu,su},n_{all})$.

U_{ci} – insert unit with commit

Figure 27 depicts the main source code for the $U_{ci}(c_{fu,su},n_{all})$. This unit is focused on measuring the required time to insert η into the ResultSet and to commit them to the database.

```
// Insert unit with commit
start=System.nanoTime();
while (rs.next()) {
    rs.moveToInsertRow();
    rs.updateString("Std_firstName","firstName");
    rs.updateString("Std_lastName","lastName");
    rs.updateDate("Std_regDate", date);
    rs.updateFloat("Std_applGrade",15F);
    rs.updateInt("StdCrs_id",1);
    rs.updateDate("Std_birthDate",bDate);
    rs.updateString("Std_eMail","email@ua.pt");
    rs.insertRow();
}
Uci_time=System.nanoTime()-start;
```

Figure 27. S-JDBC: source code for the $U_{ci}(c_{fu,su},n_{all})$.

Figure 28 presents the general behavior of the $U_{ci}(c_{fu,su},n_{all})$. The dashed lines for $C_{fu,su}$ are very close conveying the need for a more detailed graphic. Figure 29 presents a more detailed view of Figure 28 emphasizing the differences between the behaviors of the 2 contexts.

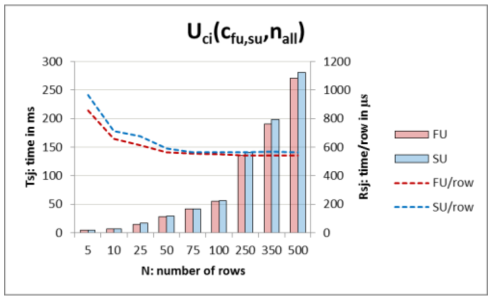


Figure 28. S-JDBC: behavior of $U_{ci}(c_{fu,su},n_{all})$.

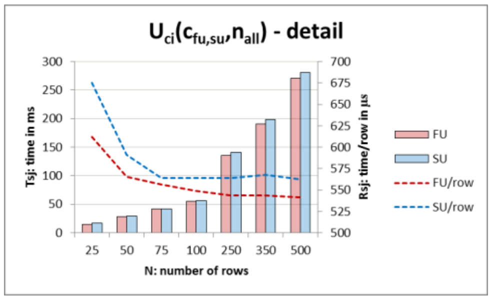


Figure 29. S-JDBC: detail of $U_{ci}(c_{fu,su},n_{all})$.

From Figure 28 and Figure 29 we may conclude that:

- T_{sj} of $C_{fu,su}$ increases with η.
- The behavior is very similar to both contexts.
- R_{sj} is weakly dependent on η for values of η >=75.
- C_{fu} gets better scores for all η.

From Figure 28 and Figure 29 we conclude that committing the new tuples to the database causes an increase in T_{sj} that ranges from 8 times for η =5 till 25 times for N=500.

Main point: scrollable ResultSets should always be avoided whenever possible.

U_{de} – delete unit

Figure 30 depicts the main source code for the $U_{de}(c_{fu,su},n_{all})$. This unit is focused on measuring the required time to delete all tuples returned by the select statement.

```
// Delete Unit
start=System.nanoTime();
while (rs.next())
    rs.deleteRow();
Ude_time=System.nanoTime()-start;
```

Figure 30. S-JDBC: source code for the $U_{de}(c_{fu,su},n_{all})$.

Figure 31 presents the general behavior of the $U_{de}(c_{fu,su},n_{all})$. The dashed lines for $C_{fu,su}$ are very close conveying the need for a more detailed graphic. Figure 32 presents a more detailed view of Figure 31 emphasizing the differences between the behaviors of the 2 contexts. From Figure 31 and Figure 32 we may conclude that:

- T_{sj} of $C_{fu,su}$ increase with η.

- The behavior is very similar to both contexts.
- For η>=50, R_{sj} tends be independent from η.
- C_{fu} gets better scores for all η.

Main point: if possible, scrollable ResultSets should always be avoided mainly when the number of rows is low.

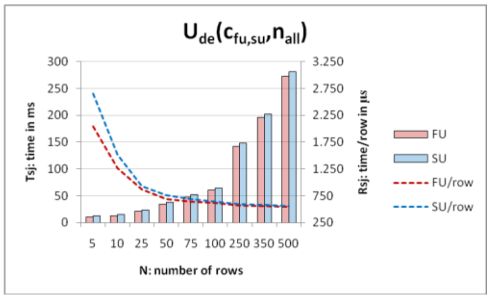


Figure 31. S-JDBC: behavior of $U_{de}(c_{fu,su},n_{all})$.

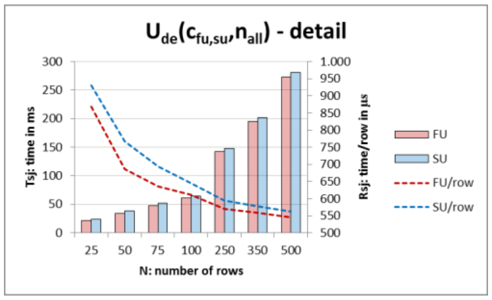


Figure 32. S-JDBC: detail of $U_{de}(c_{fu,su},n_{all})$.

Summary

Despite some particularities, as a summary of all units, we may say that:

- $C_{fr,fu}$ have better scores than $C_{sr,su}$.
- Most of the times, C_{sr} have better scores than C_{su}.
- T_{sj} increases with η except for $C_{fr,fu}$ in U_{se}.
- R_{sj} decays when η increases except for C_{fr} in U_{re}.
- R_{sj} decays rapidly from η=5 till η=50 or η>75.
- R_{sj} tends to be constant for η>=50 or η>=75

The collected measures come in line with the knowledge about the TDS protocol [46] and its implementation on the client side and on the server side. Scrollable and updatable ResultSets always use a cursor and a dataset inside the Sql Server. The cursor management and the selected row in the client side are always synchronized leading this way a decrease in the overall performance. This characteristic will also have impact in the next assessment.

2) *Assessment of scenarios*

In spite of being an important issue, the scenarios have been introduced only to simulate situations closer to a hypothetical situation. Therefore, we only briefly present some results for the assessment of the six scenarios. Figure 33, Figure 34, Figure 35, Figure 36, Figure 37 and Figure 38 present the individual behavior for each scenario.

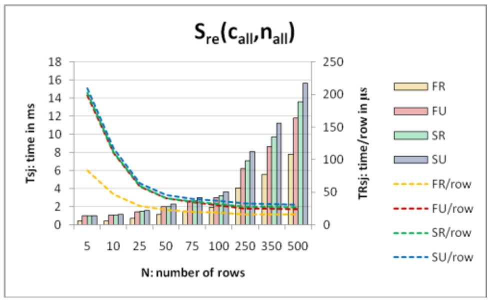


Figure 33. S-JDBC: behavior of $S_{re}(c_{all},n_{all})$

The main idea to be emphasized is that the global behavior of each scenario follows the global behavior of the correspondent unit. The measures for each η and each context are now increased by adding the correspondent collected value for U_{se}. The weight of U_{se} is almost unnoticeable for $S_{cu,ci,de}$. This derives from the fact that these contexts have very high T_{sj}. Anyway, the weight of U_{se} is not

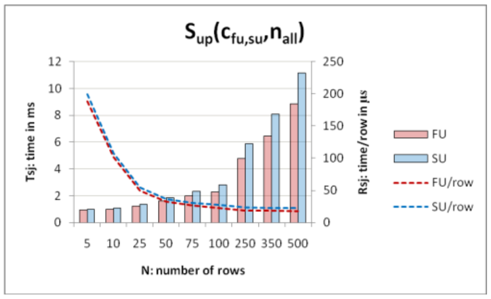


Figure 34. S-JDBC: behavior of $S_{up}(c_{fu,su},n_{all})$

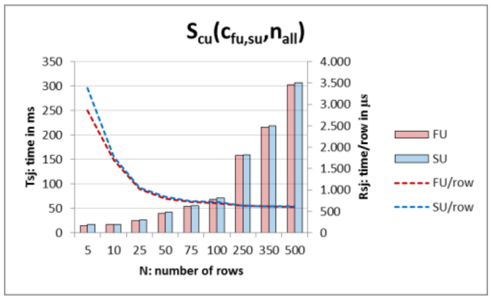


Figure 35. S-JDBC: behavior of $S_{cu}(c_{fu,su},n_{all})$

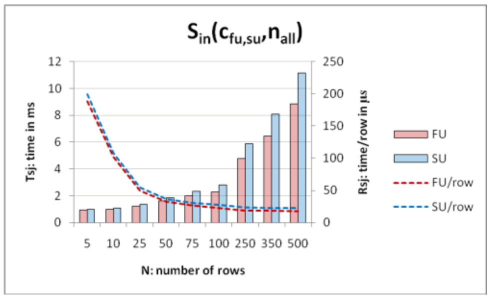


Figure 36. S-JDBC: behavior of $S_{in}(c_{fu,su},n_{all})$

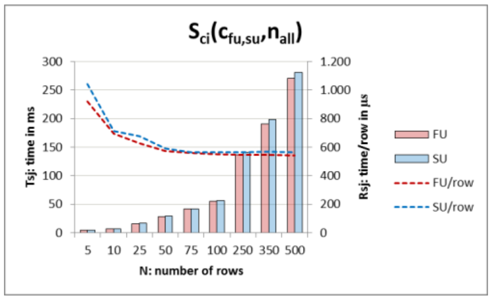


Figure 37. S-JDBC: behavior of $S_{ci}(c_{fu,su},n_{all})$

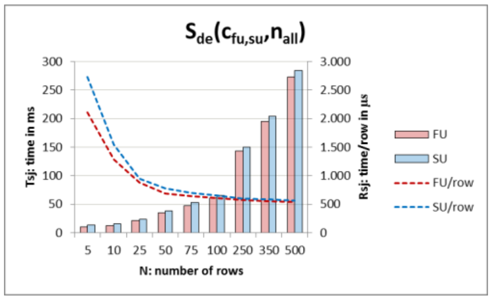


Figure 38. S-JDBC: behavior of $S_{de}(c_{fu,su},n_{all})$

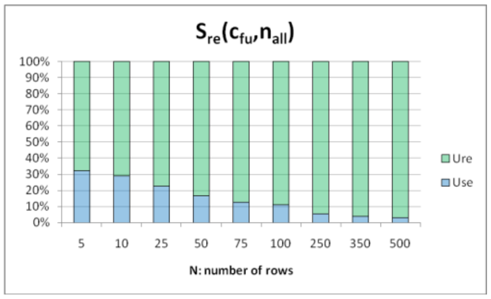


Figure 39. S-JDBC: weight of each unit in the $S_{re}(c_{fu},n_{all})$

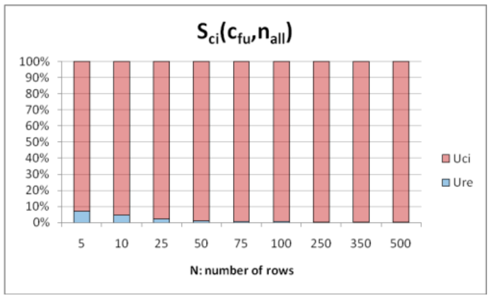


Figure 40. S-JDBC: weight of each unit in the $S_{ci}(c_{fu},n_{all})$

constant neither for each context nor for each η. Two examples for C_{fu} are shown in Figure 39 and Figure 40. In these graphics each column represents the relative weight of each unit for the total measured value. They show the relative weight of each unit in the $S_{re}(c_{fu},n_{all})$ and $S_{ci}(c_{fu},n_{all})$, respectively. As expected, U_{se} has a higher weight in S_{re} than in S_{ci} for all η.

C. *C-CRUD assessment*

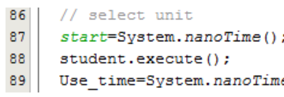
C-CRUD assessment, as mentioned before, will only comprise units. Scenarios will not be addressed because the defined scenarios are only one among infinity of possibilities. Moreover, each scenario conveys a similar behavior as the correspondent main units (others than U_{re}) as has been shown for S-JDBC.

C-CRUD assessment will be presented through graphics that show the differences between S-JDBC and C-CRUD. In all graphics, the bars represent the time required to execute a unit (T_{cc}) and the dashed lines represent the % of the difference between S-JDBC and C-CRUD (V_{cc}) = (C-CRUD)-(S-JDBC)/(C-JDBC). The axis for the bars is the left one and the axis for the dashed lines is the right one.

The main source code for the implementation of C-CRUD basically differs from the depicted code for S-JDBC on the usage of the type-state interfaces. The main structure is equal on both entities. Anyway, we will always present the source code in order to provide a better context for the understanding of how each unit was assessed. The CRUD-DO's name is *Student*.

U_{se} – select unit

Figure 41 depicts the main source code for the $U_{se}(c_{all},n_{all})$. No differences were detected between S-JDBC and C-CRUD and therefore there is no need to present the correspondent graphic. $U_{se}(c_{all},n_{all})$ behavior for S-JDBC is presented in Figure 16 and Figure 17.

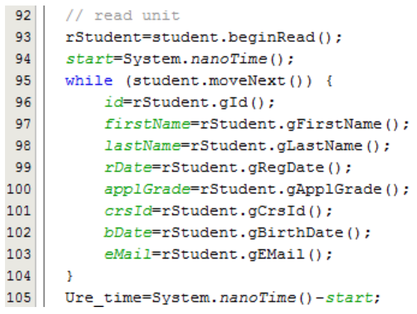


```
// select unit
start=System.nanoTime();
student.execute();
Use_time=System.nanoTime()-start;
```

Figure 41. C-CRUD: source code for the $U_{se}(c_{all},n_{all})$.

U_{re} – read unit

Figure 42 depicts the main source code for the $U_{re}(c_{all},n_{all})$. Figure 43 presents the general behavior of the $U_{re}(c_{all},n_{all})$.



```
// read unit
rStudent=student.beginRead();
start=System.nanoTime();
while (student.moveNext()) {
    id=rStudent.gId();
    firstName=rStudent.gFirstName();
    lastName=rStudent.gLastName();
    rDate=rStudent.gRegDate();
    applGrade=rStudent.gApplGrade();
    crsId=rStudent.gCrsId();
    bDate=rStudent.gBirthDate();
    eMail=rStudent.gEMail();
}
Ure_time=System.nanoTime()-start;
```

Figure 42. C-CRUD: source code for the $U_{re}(c_{all},n_{all})$.

From this figure we may conclude that:

- V_{cc} decreases for all contexts when η increases; C_{fr} is the most independent one.

- The variation of V_{cc} along η is very similar to all contexts
- The behavior of C_{fr} has the largest difference to S-JDBC. This derives from the fact that $U_{re}(C_{fr})$ in S-JDBC has by far the best scores leading to the situation where any C-CRUD overhead implies a stronger impact.
- $C_{fu,sr.su}$ have very similar differences to S-JDBC.
- Vcc for C_{fr} range from about 2.9% till 2.6%
- V_{cc} for the other contexts range from about 2.6% till 2.15%

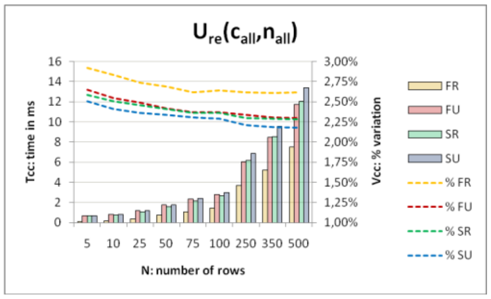
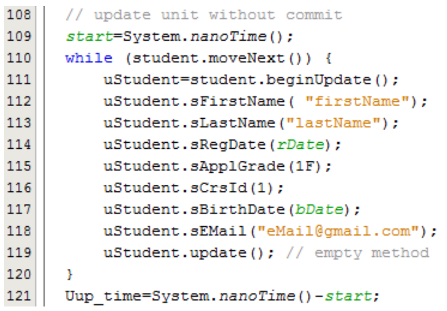


Figure 43. C-CRUD: behavior of $U_{re}(c_{all},n_{all})$.

U_{up} – update unit without commit

Figure 44 depicts the main source code for the $U_{up}(c_{fu,su},n_{all})$. Figure 45 presents the general behavior of the $U_{up}(c_{fu,su},n_{all})$.



```
// update unit without commit
start=System.nanoTime();
while (student.moveNext()) {
    uStudent=student.beginUpdate();
    uStudent.sFirstName( "firstName");
    uStudent.sLastName("lastName");
    uStudent.sRegDate(rDate);
    uStudent.sApplGrade(1F);
    uStudent.sCrsId(1);
    uStudent.sBirthDate(bDate);
    uStudent.sEMail("eMail@gmail.com");
    uStudent.update(); // empty method
}
Uup_time=System.nanoTime()-start;
```

Figure 44. C-CRUD: source code for the $U_{up}(c_{fu,su},n_{all})$.

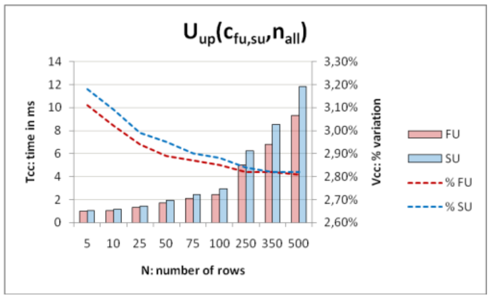


Figure 45. C-CRUD: behavior of $U_{up}(c_{fu,su},n_{all})$.

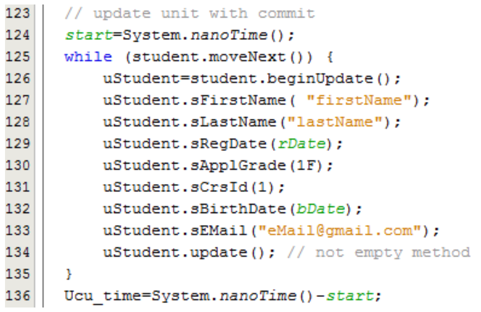
From this figure we may conclude that:

- V_{cc} decreases for all contexts when η increases.
- C_{su} has the largest difference to S-JDBC but they converge from η=5 till overlap for η>350.
- For C_{su}, V_{cc} ranges from about 3.2% till 2.8%.
- For C_{fu}, V_{cc} ranges from about 3,1% till 2.8%

U_{cu} – update unit with commit

Figure 46 depicts the source code for the $U_{cu}(c_{fu,su},n_{all})$. Figure 47 presents the general behavior of the $U_{cu}(c_{fu,su},n_{all})$. From this figure we may conclude that:

- The maximum variation of V_{cc} is 0.01% in each context.
- V_{cc} for C_{su} is always higher than for C_{fu}.
- V_{cc} ranges from 0.01% till 0.03%. The low impact of C-CRUD derives from the relative very low overhead introduced by C-CRUD. The *commit* operation is very slow weakening this way the relative weight of C-CRUD overhead.



```
// update unit with commit
start=System.nanoTime();
while (student.moveNext()) {
    uStudent=student.beginUpdate();
    uStudent.sFirstName( "firstName");
    uStudent.sLastName("lastName");
    uStudent.sRegDate(rDate);
    uStudent.sApplGrade(1F);
    uStudent.sCrsId(1);
    uStudent.sBirthDate(bDate);
    uStudent.sEMail("eMail@gmail.com");
    uStudent.update(); // not empty method
}
Ucu_time=System.nanoTime()-start;
```

Figure 46. C-CRUD: source code for the $U_{cu}(c_{fu,su},n_{all})$.

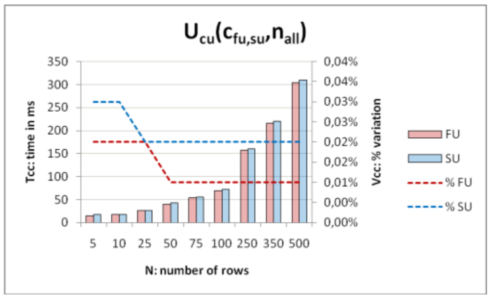


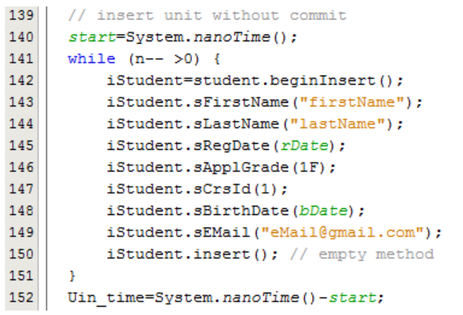
Figure 47. C-CRUD: behavior of the $U_{cu}(c_{fu,su},n_{all})$.

U_{in} – insert unit without commit

Figure 48 depicts the main source code for the $U_{in}(c_{fu,su},n_{all})$. The method *insert()* is an empty method avoiding this way committing new tuples to the database.

Figure 49 presents the general behavior of the $U_{in}(c_{fu,su},n_{all})$. From this figure we may conclude that:

- V_{cc} decreases for all contexts when η increases.
- C_{su} has the largest difference to S-JDBC but its difference to C_{fu} is minimal and converges to zero for η=500.
- For C_{su}, V_{cc} ranges from about 3.2% till <2.8%
- For C_{fu}, V_{cc} ranges from about 3,18% till <2.8%



```
// insert unit without commit
start=System.nanoTime();
while (n-- >0) {
    iStudent=student.beginInsert();
    iStudent.sFirstName("firstName");
    iStudent.sLastName("lastName");
    iStudent.sRegDate(rDate);
    iStudent.sApplGrade(1F);
    iStudent.sCrsId(1);
    iStudent.sBirthDate(bDate);
    iStudent.sEMail("eMail@gmail.com");
    iStudent.insert(); // empty method
}
Uin_time=System.nanoTime()-start;
```

Figure 48. C-CRUD: source code for the $U_{in}(c_{fu,su}, n_{all}$

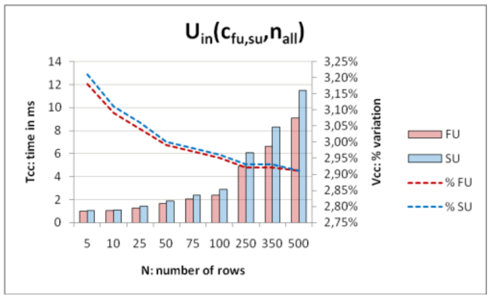
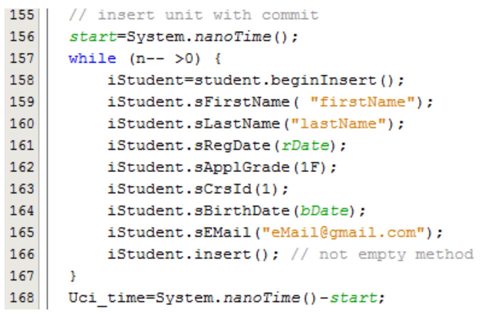


Figure 49. C-CRUD: behavior of the $U_{in}(c_{fu,su}, n_{all})$.

U_{ci} – insert unit with commit

Figure 50 depicts the main source code for the $U_{ci}(c_{fu,su}, n_{all})$. In opposite to $U_{in}(c_{fu,su}, n_{all})$ the method *insert()* commits new tuples to the database.



```
// insert unit with commit
start=System.nanoTime();
while (n-- >0) {
    iStudent=student.beginInsert();
    iStudent.sFirstName( "firstName");
    iStudent.sLastName("lastName");
    iStudent.sRegDate(rDate);
    iStudent.sApplGrade(1F);
    iStudent.sCrsId(1);
    iStudent.sBirthDate(bDate);
    iStudent.sEMail("eMail@gmail.com");
    iStudent.insert(); // not empty method
}
Uci_time=System.nanoTime()-start;
```

Figure 50. C-CRUD: source code for the $U_{ci}(c_{fu,su}, n_{all})$.

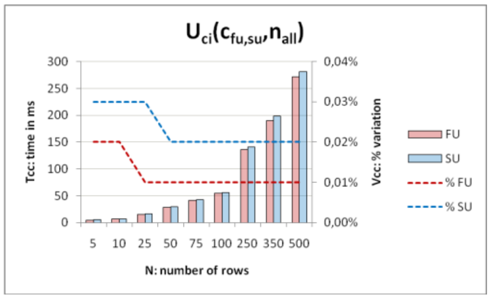


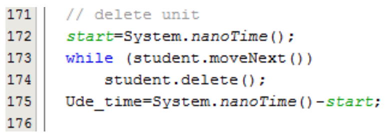
Figure 51. C-CRUD: behavior of $U_{ci}(c_{fu,su}, n_{all})$.

Figure 51 presents the general behavior of the $U_{ci}(c_{fu,su}, n_{all})$. From this figure we may conclude that:

- The maximum variation of V_{cc} along η for each context is at most 0.01%.
- V_{cc} for C_{su} is about twice the value of C_{fu}. Anyway, the involved absolute values are very small.
- V_{cc} ranges from 0.01% till 0.03%. The low impact of C-CRUD derives from the relative very low overhead introduced by C-CRUD. The *commit* operation is very slow weakening this way the relative weight of C-CRUD overhead.

U_{de} – Unit delete

Figure 52 depicts the main source code for $U_{de}(c_{fu,su}, n_{all})$.



```
// delete unit
start=System.nanoTime();
while (student.moveNext())
    student.delete();
Ude_time=System.nanoTime()-start;
```

Figure 52. C-CRUD: source code for the $U_{de}(c_{fu,su}, n_{all})$.

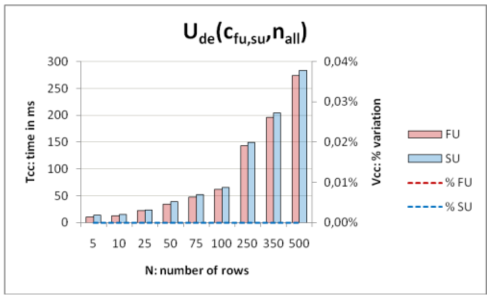


Figure 53. C-CRUD: behavior of $U_{de}(c_{fu,su}, n_{all})$.

Figure 53 presents the general behavior of the $U_{de}(c_{fu,su}, n_{all})$. From this figure we may conclude that V_{cc} is so small for both contexts that it is not possible to represent them in the graphic. This derives from the fact that the *delete* operation is too slow and also from the fact that S-JDBC and C-CRUD implementations are very similar.

Summary

Despite some particularities, as a summary of all units, we may say that:

- Between units, the weight of V_{cc} decreases when η increases.
- For slower units ($U_{cu,ci,de}$) the C-CRUD overhead is lower than 0.03%.
- For faster units ($U_{re,up,in}$) the C-CRUD overhead ranges from 3.2% till 2.4%.

VII. CONCLUSION

The solution here presented proved to be effective for bridging the gap between the object oriented and the relational paradigms in the context where programmers have no alternative but write the required CRUD expressions to implement the middle tier. This may occur in situations where CRUD expressions cannot be derived from any data model and mainly in applications where CRUD expressions are complex or very complex. The span of its effectiveness relies on two main dimensions: the model itself and the CRUD-M.

The model itself: CRUD-DOM addresses the following issues:

- CRUD-DOM encapsulates CRUD expressions of any complexity and exposes an object-oriented interface to the assisted application translating this way the row/table oriented paradigm into the object-oriented paradigm; the encapsulation hides all the complexity for the communication between the two paradigms tackling this way the impedance mismatch issue for the present context, which is focused on static customized CRUD expressions;
- interfaces are strongly-typed and type-state oriented providing this way an improved usability and productivity;
- it is amenable to the development addressing automatic code generation improving this way programmers productivity;
- CRUD-DOM totally relies on JDBC and copes with requirements as SQL expressiveness and system performance;
- it does not rely on any complementary or proprietary technology; the version here presented is based on Java but CRUD-DOM may be implemented in any other object-oriented programming language;
- it promotes the development of intermediate access layers this way decoupling applications and databases tiers and, therefore, leveraging this way the separation of concerns.

CRUD-M: CRUD-M addresses the following issues:

- from user defined SQL statements CRUD-M automatically creates all the necessary source code to implement the correspondent CRUDDOs, promoting this way programmers productivity;
- CRUD-M provides the programmers an automatic mechanism to test SQL statements promoting this way their productivity;
- CRUD-M allows programmers to easily update existing CRUDDO promoting this way their productivity.

So, the collaboration and interdependence between CRUD-DOM and CRUD-M is a key issue to achieve the three announced goals: 1) programmers' productivity – less time to develop, test and maintain middle tiers; 2) middle tier performance is kept at a level very similar to the standard JDBC API and, 3) usability is significantly improved when compared with the standard JDBC.

Regarding CRUD-DOM performance, despite the limited range of tests, the obtained results show that in most database applications the induced overhead may be considered as perfectly acceptable. Anyway, for very demanding database applications some additional attention should be given to CRUD-DOM, mainly for faster units, in order to minimize its overhead. Improving the performance of the slower units is beyond the programmer's scope. Most of the time is spent on updating the state of the database. Thus, it is expected, for these slower units, that any improvement in the source code should have a negligible impact on performance.

The automatic source code development tool, CRUD-M, designed as proof of concept, proved to be an efficient tool addressing all features of CRUD-DOM in an integrated way. Programmers are only required to input customized SQL statements of any complexity. CRUD-M relieves programmers from writing and testing any source code addressing this way the productivity requirement. Additionally, it provides an interactive GUI where programmers are guided step by step, since the editing of CRUD expressions till the creation of CRUD-DO addressing this way the usability requirement.

Some small differences in the final results between this assessment and [1] derives from the fact that the environments in which they took place are slightly different. Anyway, the fundamental conclusions and the collected results are basically identical. CRUD-DOM induces an overhead that for most of the database applications may be considered as not significant. Anyway, some more attention is needed to minimize the CRUD-DOM overhead in order to address very demanding database applications.

A new version of CRUD-DOM is being prepared. This new version will support several mechanisms of concurrency promoting this way CRUD-DOM performance in new directions. Additional new features will be also included in order to support current JDBC features. Among them:

- to provide support for the execution of SQL statements in batch mode;
- to provide support to execute stored procedures;
- to provide support to allow programmers to choose at runtime between *statements* and *preparedStatements*;

It is expected that CRUD-DOM and CRUD-M may be used in database applications where the middle tier is not a direct object-oriented perspective of relational models as happens with O/RM tools. CRUD-DOM and CRUD-M impact may be significant in database applications where CRUD expressions are very complex. Without the support of CRUD-DOM and CRUD-M, complex CRUD expressions are not easy to write, test, maintain and wrapped in a structure identical to CRUD-DOM.

REFERENCES

[1] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "CRUD-DOM: A Model for Bridging the Gap Between the Object-Oriented

and the Relational Paradigms," in *ICSEA 2010 - International Conference on Software Engineering and Applications*, Nice, France, 2010, pp. 114-122.

[2] M. David, "Representing database programs as objects," in *Advances in Database Programming Languages*, F. Bancilhon and P. Buneman, Eds., ed N.Y.: ACM, 1990, pp. 377-386.

[3] ODBMS.ORG. (2011 May). *Integrating programming languages and databases: what is the problem?* Available: http://www.odbms.org/experts.aspx#article10

[4] *Part 1: SQL Routines using the Java (TM) Programming Language*, 1999.

[5] Microsoft Corporation. (2011 May). *The LINQ Project*. Available: http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx

[6] ISO. (2011 May). *ISO/IEC 9075-3:2003*. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134

[7] Oracle. (2011 May). *JDBC Overview*. Available: http://www.oracle.com/technetwork/java/overview-141217.html

[8] Microsoft. (2011 May). *Microsoft Open Database Connectivity*. Available: http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx

[9] Microsoft Corporation. (2011 May). *Overview of ADO.NET*. Available: http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx

[10] B. Christian and K. Gavin, *Hibernate in Action*: Manning Publications Co., 2004.

[11] Oracle. (2011 May). *Oracle TopLink*. Available: http://www.oracle.com/technetwork/middleware/toplink/overview/index.html

[12] Oracle. (2011 May). *Java Data Objects (JDO)*. Available: http://www.oracle.com/technetwork/java/index-jsp-135919.html

[13] D. Yang, *Java Persistence with JPA*: Outskirts Press, 2010.

[14] C. U. Smith and L. G. Williams, *Performance Solutions: a Practical Guide to Creating Responsive, Scalable Software*, 1st ed.: Addison Wesley, 2001.

[15] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," presented at the FOSE '07- Future of Software Engineering, Minneapolis,MN,USA, 2007.

[16] IEEE, *SWEBOK*, 2004 ed. Los Alamitos,CA: IEEE Computer Society.

[17] J. Nielson, *Usability Engineering*. San Francisco, CA: Morgan Kaufman, 1993.

[18] D. G. John and L. Clayton, "Designing for Usability: Key Principles and What Designers Think," *Communications of the ACM*, vol. 28, pp. 300-311, 1985.

[19] *ISO 9241-11: Ergonomic Requirements for Office Work With Visual Display Terminals*, fdew, 1998.

[20] *ISO 13407: Human-Centered Design Processes for Interactive Systems*, 1999.

[21] *ISO/IEC 9126-1: Software Engineering - Product Quality*, 2001.

[22] *ISO/TR 16982: Usability Methods Supporting Human Centered Design*, 2002.

[23] M. J. Suárez-Cabal and J. Tuya, "Using an SQL coverage measurement for testing database applications," presented at the FSE'04 - ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering, Newport Beach-CA-USA, 2004.

[24] E. Vincent, "Is ISSTA research relevant to industrial users? panel - ISSTA 2002: empowering the developer to be a tester too!," *ACM SIGSOFT Software Engineering Notes*, vol. 27, pp. 203-204, 2002.

[25] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," ed: National Institute of Standards and Technology, 2002, pp. Planning Report 02-3.

[26] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," presented at the FOSE '07- Future of Software Engineering, Minneapolis,MN,USA, 2007.

[27] Y. Singh and B. Goel, "A step towards software preventive maintenance," *ACM SIGSOFT Software Engineering Notes*, vol. 32, 2007.

[28] B. P. Lientz and E. B. Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading,MA: Addison Wesley, 1980.

[29] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," presented at the FOSE'00 - Future of Software Engineering, Limerick,Ireland, 2000.

[30] J. W. Moore, "The ANSI binding of SQL to ADA," *Ada Letters*, vol. XI, pp. 47-61, 1991.

[31] W. Keller, "Mapping Objects to Tables - A Pattern Language," in *European Conference on Pattern Languages of Programming Conference (EuroPLoP)*, Irsse, Germany, 1997.

[32] R. Lammel and E. Meijer, "Mappings Make data Processing Go 'Round: An Inter-paradigmatic Mapping Tutorial," in *Generative and Transformation Techniques in Software Engineering*, Braga, Portugal, 2006.

[33] C. Pablo, M. Sergey, and A. Atul, "ADO.NET entity framework: raising the level of abstraction in data programming," in *ACM SIGMOD International Conference on Management of Data*, Beijing,China, 2007, pp. 1070-1072.

[34] R. C. William and R. Siddhartha, "Safe query objects: statically typed objects as remotely executable queries," in *27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005, pp. 97-106.

[35] A. M. Russell and H. K. Ingolf, "SQL DOM: compile time checking of dynamic SQL statements," in *27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005, pp. 88-96.

[36] W. Gary, G. Carl, S. Zhendong, and D. Premkumar, "Static checking of dynamically generated queries in database applications," *ACM Transansactions on Software Eng. Methodology*, vol. 16, p. 14, 2007.

[37] Microsoft Corporation. (2011 May). *ADO.NET*. Available: http://msdn.microsoft.com/en-us/library/aa286484.aspx

[38] M. Andy, E. Wolfgang, and S. R. David, "Impact analysis of database schema changes," in *30th International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 451-460.

[39] B. Gregory, W. W. Bruce, and A. G. S. Paolo, "Using parse tree validation to prevent SQL injection attacks," in *5th International Workshop on Software Engineering and Middleware*, Lisbon, Portugal, 2005.

[40] Oracle. (2011 May). *Interface Statement*. Available: http://download.oracle.com/javase/6/docs/api/java/sql/Statement.html

[41] Oracle. (2011 May). *Interface ResultSet*. Available: http://download.oracle.com/javase/6/docs/api/java/sql/ResultSet.html

[42] Oracle. (2011 May). *Interface PreparedStatement*. Available: http://download.oracle.com/javase/6/docs/api/java/sql/PreparedStatement.html

[43] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. 12, pp. 157-171, 1986.

[44] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Softw. Eng.*, vol. 12, pp. 157-171, 1986.

[45] Microsoft. (2011 May). *SQL Server JDBC Driver 2.0 Documentation*. Available: http://technet.microsoft.com/en-us/library/ff928320(SQL.10).aspx

[46] Microsoft. (2011 May). *[MS-TDS]: Tabular Data Stream Protocol Specification*. Available: http://msdn.microsoft.com/en-us/library/dd304523(v=prot.13).aspx