Assessment of a Enhanced ResultSet Component for Accessing Relational Databases

Oscar M Pereira, Rui L Aguiar
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*—Call Level Interfaces (CLI) provide services aimed at easing the integration of database components and components from client applications. CLI support native SQL statements keeping this way expressiveness and performance of SQL. Thus, they cannot be discarded as a valid option whenever SQL expressiveness and SQL performance are considered key requirements. Despite the aforementioned performance advantage, CLI do not comprise other important performance features, as concurrency over the in-memory data. In this paper we present and assess a component that is a concurrent version of the ResultSet interface from the JDBC API. Several threads may interact simultaneously in the same instance of the ResultSet in a concurrent fashion and can be simultaneously connected to the underlying database. The main contributions of this paper are twofold: i) the design of an Enhanced ResultSet Component to provide a concurrent access to relational databases; ii) the evaluation of its performance. The Enhaced ResultSet performance will be assessed in a real scenario. The outcome shows that the gain in performance may increase until 80%.**

Keywords - performance; Call Level Interfaces, ResultSet, concurrency.

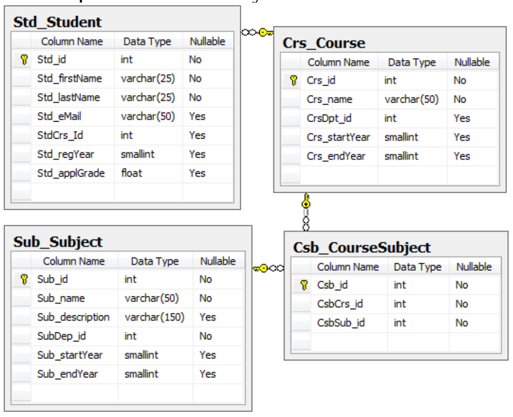
I. INTRODUCTION

Performance is a non functional software requirement that evaluates how well a system or a component copes with its requirements namely for timeless [1]. There are two dimensions: responsiveness and scalability. Responsiveness evaluates the system conformance to response time requirements. It may refer to the amount of time to accomplished a task or the number of tasks that can be accomplish in a given amount of time. Scalability evaluates the capacity of a system to handle growing demand of power computation while keeping its responsiveness. Performance is a pervasive outcome of software systems [2]. Everything affects it: software design, programming paradigms and languages, compilers, communication networks, hardware and third party software, among others. As a pervasiveness quality, performance opens many opportunities to research contributions.

Very often, performance 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 decision will have an impact on the overall performance. There are many technological solutions for the connection layer between client side applications and server side databases each one with its own characteristics: ORM [3-5], embedded SQL, CLI [6-9], persistent frameworks [10-11]. If performance is considered a key requirement CLI have to be considered as a promising alternative [12]. CLI are programming API aimed at easing the integration of client software components and database components. They rely on SQL statements promoting this way SQL expressiveness and SQL performance. CLI provide mechanisms to encode create, read, update, and delete (CRUD) expressions inside strings, easily incorporating the power and the expressiveness of SQL. Nevertheless, CLI do not provide some of the most well known and common features to improve system performance being concurrency the most paradigmatic case.

This paper addresses concurrency of the most critical component of CLI: the component that holds data from a database and provides an interface to client applications: ResultSet [13] in JDBC and RecordSet [14] in ADO.Net. Through this interface applications may read, update, delete and insert data into databases. We will present a solution for a concurrent (thread safe) ResultSet, known as Enhanced ResultSet Component (ERC) and we will also assess its performance in a real scenario.

For conciseness, Figure 1 presents a partial view of a database schema which will be used throughout this paper. This database is associated with the academic life, and as such we expect it to be easily understood.


Std_Student

Column Name	Data Type	Nullable
Std_id	int	No
Std_firstName	varchar(25)	No
Std_lastName	varchar(25)	No
Std_eMail	varchar(50)	Yes
StdCrs_Id	int	Yes
Std_regYear	smallint	Yes
Std_applGrade	float	Yes

Crs_Course

Column Name	Data Type	Nullable
Crs_id	int	No
Crs_name	varchar(50)	No
CrsDpt_id	int	Yes
Crs_startYear	smallint	Yes
Crs_endYear	smallint	Yes

Sub_Subject

Column Name	Data Type	Nullable
Sub_id	int	No
Sub_name	varchar(50)	No
Sub_description	varchar(150)	Yes
SubDep_id	int	No
Sub_startYear	smallint	Yes
Sub_endYear	smallint	Yes

Csb_CourseSubject

Column Name	Data Type	Nullable
Csb_id	int	No
CsbCrs_id	int	No
CsbSub_id	int	No

Figure 1. Partial view of the database schema

Throughout this paper all examples are based on Java, SQL Server 2008 and JDBC (CLI) for SQL Server (sqljdbc4). Code may not execute properly since we will only show the relevant code for the points under discussion.

This paper is organized as follows. Section II presents the motivation for this work; Section III presents related work; Section IV presents the ERC; Section V presents the results of the assessments and finally, Section VI presents some final conclusions and remarks.

II. MOTIVATION

Database applications very often are very demanding regarding the way client side applications deals with data. Traditional approaches rely on a pattern where each client side thread manages locally its own data in an exclusive mode. Each thread retrieves what is needs. There is no standard for sharing resources in a concurrent way. This may lead to situations where different threads are asking for the same data promoting performance decay in several ways: by increasing the work load of the database server, by increasing the traffic in all layers of the network, by increasing the memory usage, etc. Database applications performance should improve if different threads were able to share common resources. The question is where should we provide those services? Should they be provided at the application layer? Or should they be provided at a lower level? Any option may have its own pros and cons.

We claim that gains in performance is maximized when concurrency is implemented at a lower level. At a lower level, resources may be optimized avoiding additional copies of data, avoiding additional code for data manipulation and chiefly to take advantage of the interaction with low level API as JDBC. Therefore, in this paper we will implement a concurrent component relying on JDBC API. ResultSet [13] interface is a key component in JDBC and also our target component. Among the several important functionalities it provides, we stress two of them:

- data returned from *Select* statements are made available to applications through the ResultSet interface;
- updatable ResultSets provide an additional functionality which consists in the possibility of executing the following actions: updating, deleting and inserting rows in the current ResultSet. These changes are also executed on the database server.

Despite being a key component it does not provide any concurrent mechanism to deal with the in-memory data it manages. It is our goal to foresee a thread safe implementation for the standard ResultSet interface.

III. RELATED WORK

To the best of our knowledge, no similar work was found, involving concurrency at the ResulSet/RecordSet level. Thus, some research was done around tools aimed at integrating client applications and databases. A survey was made for the most popular tools, as Hibernate [4], Spring [15], TopLink [5], JPA [11] and LINQ [16]. These tools may provide concurrency but always at a very high level. Basically, they provide some locking policies implemented in order to synchronize read/write actions. But these read/write synchronized actions are not executed over the same memory location. They are executed over distinct objects, as sessions in Hibernate. These objects (as sessions) are not thread-safe and therefore do not provide any protocol to access concurrently the in-memory data.

IV. ENHANCED RESULTSET COMPONENT

Before delving into the concurrent version of the ResultSet interface we will emphasize some of the most important features of the ResultSet interface namely the services aimed at dealing with in-memory data.

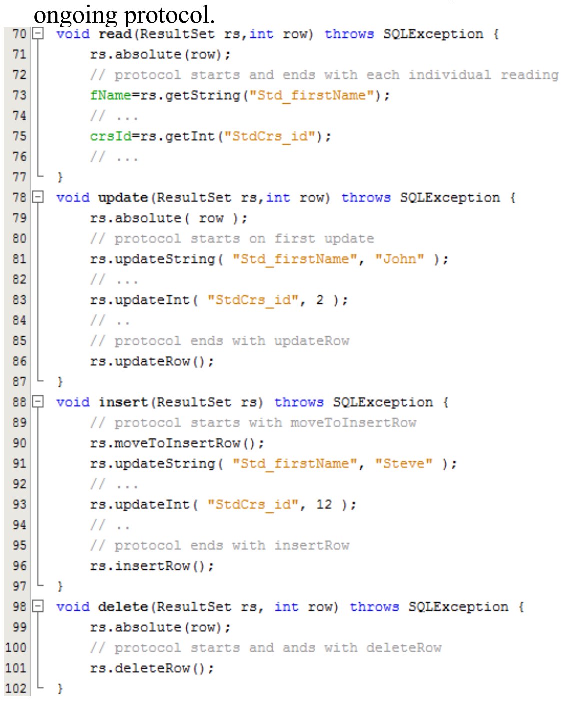
A. ResultSet Interface

The Statement interface [17] is used for executing SQL statements and returning the results it produces. The returned results are managed by a ResultSet interface [13]. Loosely speaking ResultSet interface provides two orthogonal functionalities: *scrollability* and *updatability*. Scrollability defines the ability to scroll over the in-memory rows retrieved from a 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 capability to update the in-memory rows retrieved from a database. There are two main possibilities: *read only* – update, insert and delete actions cannot be performed over the ResultSet; *updatable* – read, update, insert and delete actions may be performed on the ResultSet. These functionalities are defined at instantiation time of the parent Statement.

These different types of ResultSets raise an important question. Is it necessary to provide concurrency for all types of ResultSets? Regarding scrollability, forward only ResultSets are very restrict because all threads should always and simultaneously point to the same row. Regarding updatability, concurrency makes sense for both types: read only and updatable. Read only ResultSets always provide a subset of the updatable interface. In order to address the most general situation we chose to implement a concurrent version for a scrollable and updatable ResultSet interface.

Concurrency over ResultSets raise some difficulties because some usage protocols of ResultSets are complex comprising several instructions. Figure 2 depicts protocols for 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 (implicitly for update and explicitly for insert) and an end instruction. Besides the starting and ending instruction, the main issue is that for the update and insert protocols the cursor cannot be moved from the current selected row while the protocol is being executed. If any thread moves the cursor from the current row the following situations will occur: the insert protocol will be aborted; the update protocol will discard all previous updates. For read and delete protocols, as for update and insert, it is always necessary to assure the correct cursor position. Thus, in a concurrent environment, one must guarantee that:

- while some ongoing protocols are taking place (update and insert) the cursor cannot be moved into another position;
- whenever a thread becomes the running thread, its cursor context must be restored regardless the ongoing protocol.



```java
void read(ResultSet rs,int row) throws SQLException {
    rs.absolute(row);
    // protocol starts and ends with each individual 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", "John" );
    // ...
    rs.updateInt( "StdCrs_id", 2 );
    // ..
    // protocol ends with updateRow
    rs.updateRow();
}
void insert(ResultSet rs) throws SQLException {
    // protocol starts with moveToInsertRow
    rs.moveToInsertRow();
    rs.updateString( "Std_firstName", "Steve" );
    // ...
    rs.updateInt( "StdCrs_id", 12 );
    // ..
    // protocol ends with insertRow
    rs.insertRow();
}
void delete(ResultSet rs, int row) throws SQLException {
    rs.absolute(row);
    // protocol starts and ands with deleteRow
    rs.deleteRow();
}
```

Figure 2. Read, Update, Insert and Delete protocols

B. *ERC Features*

ERC implementation relies on the following features:

- ERC interface usage protocols and ResultSet usage protocols are basically the same. Conceptually, the main difference is the possibility of having several cursors pointing simultaneously to different rows in the same ResultSet instance. The main usage differences are: the possibility of creating new cursors, a group of two new methods and a new exception that have been included (explained below) .
- ERC interface is thread safe. Among others, it deals with situations where atomic operations aggregate complex protocols, as is the case of inserting a new row.
- From users' point of view, ERC concurrency is provided in a transparent way by individual cursors.
- Each cursor is mainly characterized by a pointer to a row in the underlying ResultSet.
- ERC have two main super-states: locked and unlocked.
- Only one cursor at time may lock the ERC. Other cursors have to wait until it becomes unlocked.
- Cursors are locked when: 1) explicitly ordered by users; 2) when one of the following protocols is in course: inserting a new row or updating a row; 3) during the execution of individual operations, as *next()*, *getInt(...)* and *deleteRow()*.
- ERC goes to unlock state as sequence of: a) explicitly ordered by users; b) invocation of any method that ends or aborts the ongoing protocol.
- ERC promotes concurrency to the lowest possible grain. The grain is determined by each individual protocol to access the underlying ResultSet. In most of the situations, concurrency grain is controlled in an instruction by instruction basis, as *next()*, *getInt(...)*, *isLast()*.
- Each user thread may create as many cursors as necessary over the same ERC instance.
- One thread may only have one cursor locked at a time. In order to avoid deadlock situations, an exception is raised whenever a thread tries to have two cursors simultaneously locked (this is the new exception).
- In order to improve ERC performance, two additional methods are provided to explicitly handle lock states: *lock()* and *unlock()*. These methods should be used carefully because the gain in performance is paid with less concurrency (these are the two new methods).

C. *ERC Architecture*

ERC comprises 2 classes and 2 interfaces as shown in Figure 3 (EResultSet), Figure 4 (IEResultSet), Figure 6 (Cursor) and Figure 7 (ICursor). Figure 3 shows the class diagram for EResultSet. Only some relevant information is shown in order to avoid overcrowd the diagram (this policy has been applied to all the presented class diagrams). This class is responsible for implementing concurrency between cursors. Figure 4 presents the interface through which users should access EResultSet. This interface only provides one method which is responsible for creating new cursors. The remaining methods are only accessible through the interface ICursor (Figure 7). Figure 5 shows the method *next(...)* of the EResultSet class which is responsible for moving the cursor down one row from its current position. This block of code shows that cursor management always conveys some additional processing. This issue is analyzed in some detail in section V. In accordance with the requirements, the cursor moves down one row conveying the same behavior and feedback as if it was used in a standard ResultSet. Figure 6 presents the class diagram for the class Cursor. Users access this class through the interface ICursor presented in Figure 7. Each Cursor instance is characterized by a unique *cursorId*, threadId (from *Thread.currentThread().getId()*) and the instance of EResultSet that have created it. Each method of *Cursor* has a correspondent method in EResulSet. Users do not invoke directly EResultSet methods but always through ICursor interface. Methods of class Cursor basically have a call to the correspondent method in the associated EResultSet instance as shown in Figure 8. From users' point of view, ICursor provides the same interface as the one of the standard ResultSet (exception is the two additional methods and also the additional exception).

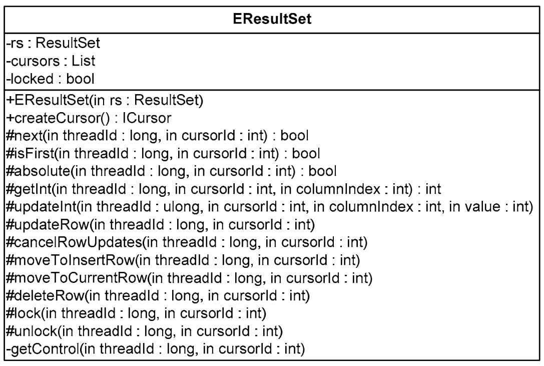

EResultSet
-rs : ResultSet
-cursors : List
-locked : bool
+EResultSet(in rs : ResultSet)
+createCursor() : ICursor
#next(in threadId : long, in cursorId : int) : bool
#isFirst(in threadId : long, in cursorId : int) : bool
#absolute(in threadId : long, in cursorId : int) : bool
#getInt(in threadId : long, in cursorId : int, in columnIndex : int) : int
#updateInt(in threadId : ulong, in cursorId : int, in columnIndex : int, in value : int)
#updateRow(in threadId : long, in cursorId : int)
#cancelRowUpdates(in threadId : long, in cursorId : int)
#moveToInsertRow(in threadId : long, in cursorId : int)
#moveToCurrentRow(in threadId : long, in cursorId : int)
#deleteRow(in threadId : long, in cursorId : int)
#lock(in threadId : long, in cursorId : int)
#unlock(in threadId : long, in cursorId : int)
-getControl(in threadId : long, in cursorId : int)

Figure 3. *EResultSet* class diagram

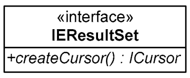
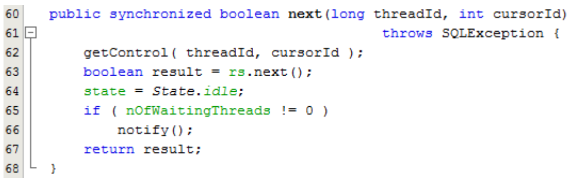


Figure 4. IEResultSet class diagram



```
public synchronized boolean next(long threadId, int cursorId)
                                throws SQLException {
    getControl( threadId, cursorId );
    boolean result = rs.next();
    state = State.idle;
    if ( nOfWaitingThreads != 0 )
        notify();
    return result;
}
```

Figure 5. method *next()* from class *EResultSet*

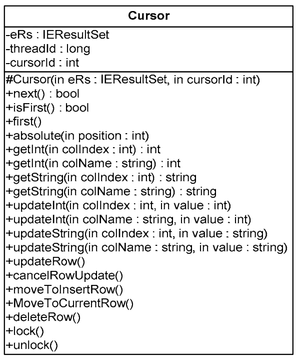

Cursor
-eRs : IEResultSet
-threadId : long
-cursorId : int
#Cursor(in eRs : IEResultSet, in cursorId : int)
+next() : bool
+isFirst() : bool
+first()
+absolute(in position : int)
+getInt(in colIndex : int) : int
+getInt(in colName : string) : int
+getString(in colIndex : int) : string
+getString(in colName : string) : string
+updateInt(in colIndex : int, in value : int)
+updateInt(in colName : string, in value : int)
+updateString(in colIndex : int, in value : string)
+updateString(in colName : string, in value : string)
+updateRow()
+cancelRowUpdate()
+moveToInsertRow()
+MoveToCurrentRow()
+deleteRow()
+lock()
+unlock()

Figure 6. *Cursor* class diagram

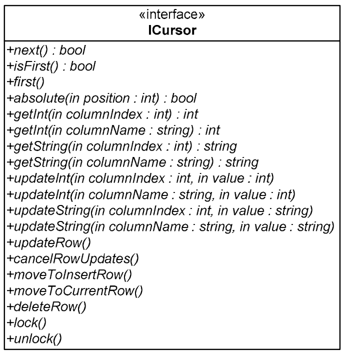
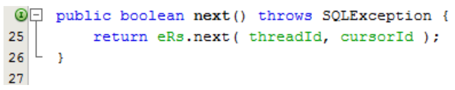

«interface»
ICursor
+next() : bool
+isFirst() : bool
+first()
+absolute(in position : int) : bool
+getInt(in columnIndex : int) : int
+getInt(in columnName : string) : int
+getString(in columnIndex : int) : string
+getString(in columnName : string) : string
+updateInt(in columnIndex : int, in value : int)
+updateInt(in columnName : string, in value : int)
+updateString(in columnIndex : int, in value : string)
+updateString(in columnName : string, in value : string)
+updateRow()
+cancelRowUpdates()
+moveToInsertRow()
+moveToCurrentRow()
+deleteRow()
+lock()
+unlock()

Figure 7. *ICursor class diagram*



```
public boolean next() throws SQLException {
    return eRs.next( threadId, cursorId );
}
```

Figure 8. method next() from class Cursor

D. *Users' Perspective*

From a design perspective, Figure 9 presents a possible implementation where *Root* is the main class which is responsible for instantiating EResultSet. Then, the IEResultSet interface may be shared by as many threads (*Student*) as necessary. Then, each Student may create as many cursors as necessary.

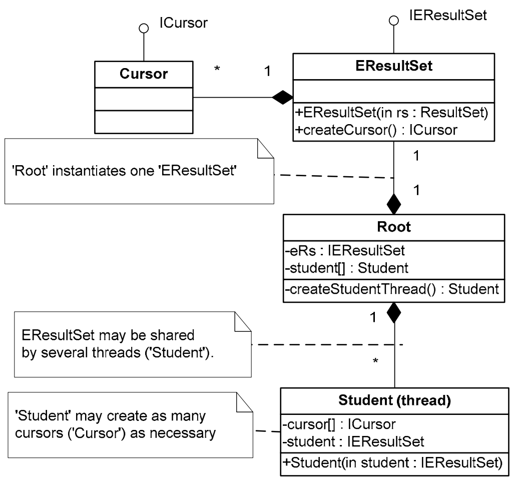
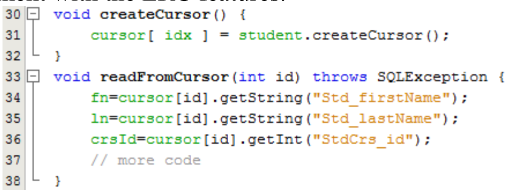

ICursor
IEResultSet
Cursor
*
1
EResultSet
+EResultSet(in rs : ResultSet)
+createCursor() : ICursor
'Root' instantiates one 'EResultSet'
1
1
Root
-eRs : IEResultSet
-student[] : Student
-createStudentThread() : Student
EResultSet may be shared by several threads ('Student').
1
*
'Student' may create as many cursors ('Cursor') as necessary
Student (thread)
-cursor[] : ICursor
-student : IEResultSet
+Student(in student : IEResultSet)

Figure 9. Design perspective for ERC

Figure 10 presents IEResultSet and ICursor from users' perspective. IEResultSet only provides one single method aimed at creating new cursors (line 31). From users' perspective, as already mentioned, ICursor interface is equivalent to the standard ResultSet interface (lines 34-36) in agreement with the ERC features.



```
30  void createCursor() {
31      cursor[ idx ] = student.createCursor();
32  }
33  void readFromCursor(int id) throws SQLException {
34      fn=cursor[id].getString("Std_firstName");
35      ln=cursor[id].getString("Std_lastName");
36      crsId=cursor[id].getInt("StdCrs_id");
37      // more code
38  }
```

Figure 10. ICursor from users' perspective

V. PERFORMANCE ASSESSMENT

Performance is an indicator of how well a software system or component meets its requirements namely for timeliness [1]. There are usually two dimensions considered: responsiveness and scalability. This paper is devoted to responsiveness. Scalability will be considered in a near future. Hereafter, performance should be understood as the responsiveness dimension.

The performance assessment here presented covers the standard ResultSet and the ERC. Two computers were used to accomplish the assessments: PC1 - Dell Latitude E5500, Intel Duo Core P8600 @2.40GHz, 4.00 GB RAM, Windows Vista Enterprise Service Pack 2 (32bits), Java SE 6, JDBC(sqljdbc4), NetBeans 6.8; PC2 – Asus-P5K-VM, Intel Duo Core E6550 @2,333GHz, 4.00 GB RAM, Windows XP Professional Service Pack 3, SQL Server 2008. The minimum used counting interval assumed in all assessments is 0,1ms. 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 cancelled; a dedicated local network connecting PC1 and PC2 has been used in exclusive mode and performing 100MBits of bandwidth.

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 SQL Server database properties were changed as, *Auto Update Statistics = false* and *Recovery Model = Simple*.

A. Methods of EResultSet

All EResultSet methods have equivalent boilerplate code conveying also an equivalent performance decay in each method. Therefore, our attention may be focused on paths inside the boilerplate code that may influence differentially the performance decay. Figure 11 depicts the basic flowchart diagram which comprise 3 distinct paths. The three main paths are: P1) no change on cursor context – current cursor is the same as the last cursor; P2) change on cursor context – current cursor (*cursorId*) is not the same as the last one, and finally P3) wait for unlocking – cursor has to wait because EResultSet is locked.

In real situations, individual paths are combined in real paths: P1, P2+P1 and P3+P2+P1. Path P2 has the most relevant overhead. In highly stressed situations were path P2 may occur very frequently, its impact may not be negligible. P2 comprises always two actions over the underlying ResultSet: keep the current cursor context (belongs to the previous *cursorId*) and restore the current context for the current *cursorId*. The policy for swapping cursor context is lazy swap. An effort has been made to minimize the overhead.

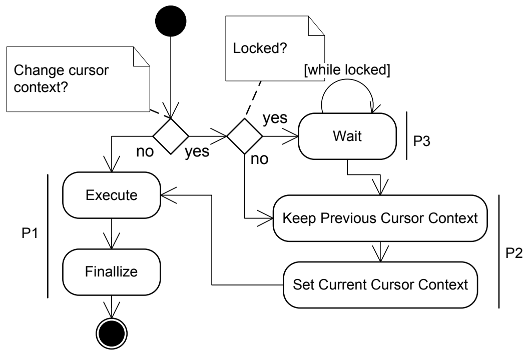


Figure 11. *Basic flowchart diagram*

As mentioned before, two new methods were added, *lock* and *unlock*. These methods actively empower users giving them control over the blocking process. After invoking *lock*, ERC will remain locked until the same *cursorId* invokes *unlock*. In the meanwhile no swap on cursor context is allowed. This approach definitely improves ERC performance whenever swaps in cursor context are not welcome, as it will be shown. However, these methods should be used discretionarily because concurrency is suspended whenever an exclusive access is active.

B. Scenario

A scenario was set up to accomplish the performance assessment. Two personal computers were used: PC1 as a client running the assessment for ResultSet and also for ERC, and PC2 as a database server. This scenario confines a client server architecture. In order to have a more detailed assessment of the ERC, for each statement type (Select, Insert an Update), we have enforced some conditions to the lock state of ERC, as always locked (AL), locked on row by row basis (LR) or never locked (NL). AL locks explicitly ERC before executing any action over the underlying ResultSet and unlocks it after accomplishing all the schedule actions over the same ResultSet. LR locks explicitly ERC before executing the first action over a row and unlocks it after accomplishing the last action over the same row. NL never locks explicitly the ERC delegating this responsibility to the underlying ERC. These conditions have impact on changes of context of cursors and therefore on the overall performance of ERC. Table 1shows the results for the assessment. Each individual result is characterized by the context in which it took place: the number of Running Threads (1-500) and the Control (A-action, T-type, L-lock policy). In order to get reliable results, each individual assessment result was computed as the mean value of the 10 best times out of 100. Most of the assessments were executed two or more times in order to avoid abnormal circumstances and therefore check the validity of the results. One significant difference exist between the assessment over the standard ResultSet and the ERC: while the standard version has to obtain a ResultSet for each running thread, the ERC obtains one single ResultSet for all running threads. It means that for the standard ResultSet it is necessary to execute one query for each thread while for the ERC it is only necessary to execute one single query. As an example, let's focus on the *Select* statement and define a context where there are 10 threads to be feed by the content of a table containing 100 rows. For the standard ResultSet it is necessary to execute 10 *Select* statements, one for each ResultSet and thread (every ResultSet will the same content). For the ERC, it is necessary to execute a single *Select* statement (the same as the one executed for the standard ResultSet*)* and share the ResultSet among all 10 threads. In order to avoid performance decay in SQL Server, the table used for the assessment (*Std_Student*) was always cleaned between individual assessments.

Table 1 presents the obtained results for the performance assessment. Values are presented in units of 0.1ms. Figure 12 shows the main control cycle for all assessments. This cycle assures that the obtained results always reflect the times for setting up the require context to simulate real situations. We will survey the assessment analyzing the results by the type of each SQL statement: Read, Update and Insert. The Delete statement has not been addressed because it would require a very different context for its execution. While, for example, it is possible to update a row as many times as necessary, a row may only be deleted once.

TABLE I. RESULTS OF ASSESSMENT

Control			Running Threads										
A	**T**	**L**	**1**	**5**	**10**	**25**	**50**	**75**	**100**	**150**	**200**	**250**	**500**
Read	ResultSet	-	49	217	428	1,065	2,139	3,199	4,257	6,387	8,528	10,692	22,083
	ERC	AL	250	330	431	736	1,250	1,787	2,348	3,426	4,535	5,682	12,307
		NL	250	381	555	1,103	1,893	2,607	3,715	4,863	6,609	8,796	17,940
		RL	251	362	483	844	1,532	2,018	2,798	4,202	5,960	7,465	15,546
Update	ResultSet	-	253	1,229	2,475	6,217	12,315	18,493	25,063	38,775	50,993	65,216	133,214
	ERC	AL	471	1,398	2,572	6,076	11,936	17,823	23,773	35,823	48,435	61,945	128,668
		NL	472	1,401	2,581	6,102	11,999	17,897	23,853	35,885	49,666	64,465	131,725
Insert	ResultSet	-	507	1,179	2,370	5,896	11,860	17,818	23,830	36,105	48,976	61,473	128,738
	ERC	AL	468	1,039	2,092	5,224	10,439	15,668	20,912	31,519	42,356	53,142	114,785
		NL	497	1,038	2,091	5,224	10,435	15,669	20,934	31,476	42,082	53,001	111,648

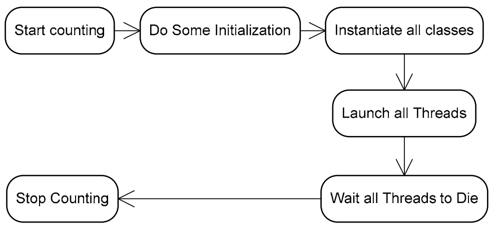
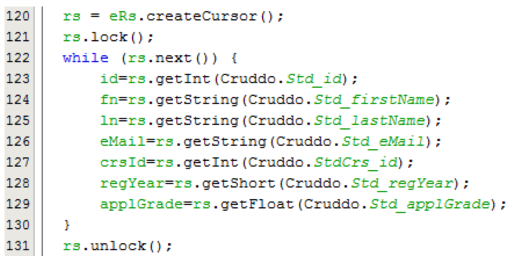


Figure 12. Control cycle for all assessments



```
rs = eRs.createCursor();
rs.lock();
while (rs.next()) {
    id=rs.getInt(Cruddo.Std_id);
    fn=rs.getString(Cruddo.Std_firstName);
    ln=rs.getString(Cruddo.Std_lastName);
    eMail=rs.getString(Cruddo.Std_eMail);
    crsId=rs.getInt(Cruddo.StdCrs_id);
    regYear=rs.getShort(Cruddo.Std_regYear);
    applGrade=rs.getFloat(Cruddo.Std_applGrade);
}
rs.unlock();
```

Figure 14. Code for the EResultSet assessment

1) *Assessment by SQL statements*

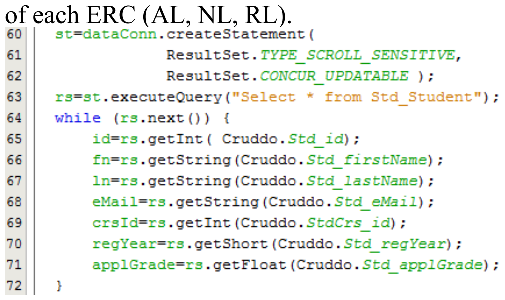
a) *Select*

All assessments for the *Select* statement were performed on the following context:

- Underlying statement: "select * from Std_Student";
- Table "*Std_Student*" pre-filled with 50 rows;
- Sequentially read all attributes of all rows;
- All attributes indexed by column index;

Figure 13 shows the main block of code for each thread of the standard ResultSet. Figure 14 shows the main block of code for each thread of the ERC (AL).

Four tests were carried out: one for the standard ResultSet and three for the ERC (AL, NL, RL). Figure 15 shows graphically the obtained results and Figure 16 shows the ratios between the results of ResultSet and the results of each ERC (AL, NL, RL).



```
st=dataConn.createStatement(
            ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE );
rs=st.executeQuery("Select * from Std_Student");
while (rs.next()) {
    id=rs.getInt( Cruddo.Std_id);
    fn=rs.getString(Cruddo.Std_firstName);
    ln=rs.getString(Cruddo.Std_lastName);
    eMail=rs.getString(Cruddo.Std_eMail);
    crsId=rs.getInt(Cruddo.StdCrs_id);
    regYear=rs.getShort(Cruddo.Std_regYear);
    applGrade=rs.getFloat(Cruddo.Std_applGrade);
}
```

Figure 13. Code for the standard *ResultSet* assessment

From graphic in Figure 15, it is clear that ERC has better scores than ResultSet only if the total number of concurrent threads is greater than 10 for AL, greater than 15 for RL and greater than 25 for NL. From Figure 16, we may say that ERC performance increases, when compared to the standard ResultSet, permanently until reaching 75 threads. In the range of 75-500 simultaneous running threads the ERC performance is about 1.8 (AL), 1.5 (RL) and 1.2 (NL) times of the standard ResultSet. Another important issue is related to the changes on contexts of cursors as we have suggested previously in this section. As expected, performance decays from AL, towards RL and finally towards NL. This issue will be addressed in more detail in 2) in this section.

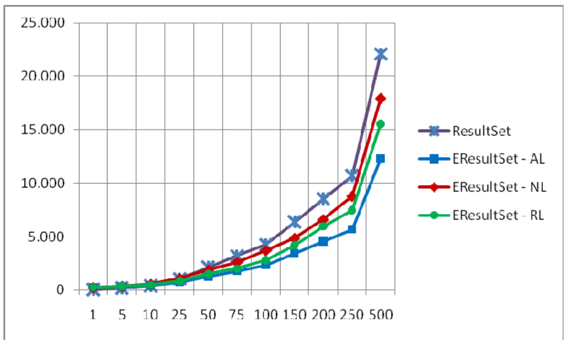


Figure 15. Results of assessments for the *Select* statement

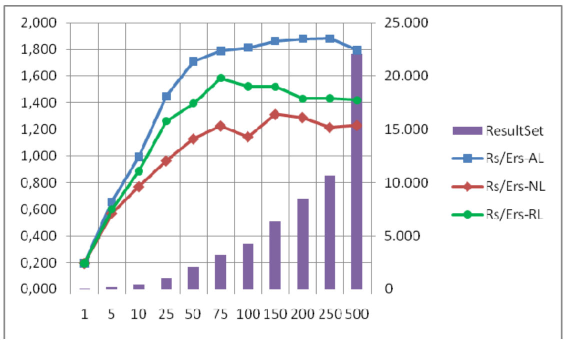


Figure 16. Ratios between ResultSet and EResultSet results for the Select statement

b) *Update*

All tests for the *Update* statement were performed on following context:

- Underlying statement: “select * from Std_Student”;
- Table “*Std_Student*” pre-filled with 50 rows;
- Sequentially update all attributes of all rows;
- All attributes indexed by column index;

The code to perform the update tests is similar to the code presented for the select assessment. The main difference is the substitution of *get* methods by *update* methods. Three tests were carried out: one for the standard ResultSet and two for the ERC (AL, NL). No assessment was carried out for RL because the first update action (*rs.update(...)*) locks the underlying ResultSet conveying this way a similar effect as an explicit lock on a row by row basis. Figure 17 shows graphically the obtained results and Figure 18 shows the ratios between the results of ResultSet and the results of each of ERC (AL, NL). These two figures, complemented with the information contained in Table 1, shows that performance is only susceptible to ResultSet vs ERC if the number of simultaneous threads is under 25. In this range, ERC performance is about 55% to 100% of the standard ResultSet. For values above 25 threads, ERC performance is slightly better than the one of standard ResultSet.

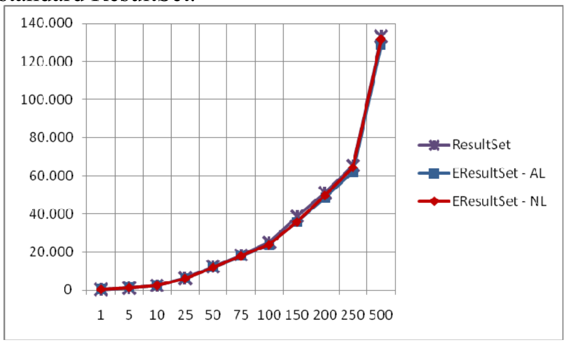


Figure 17. Results of assessments for the Update statement

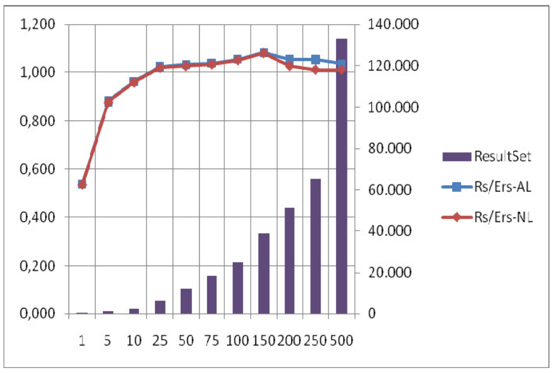


Figure 18. Ratios between ResultSet and EResultSet results for the Update statement

c) *Insert*

All tests for the *Insert* statement were performed on following context:

- Underlying statement: “select * from Std_Student”;
- Table “*Std_Student*” with no rows;
- Sequentially insert all attributes for 50 rows;
- All attributes indexed by column index;

The code to perform the *Insert* assessment is similar to the code presented for the update test. Three tests were carried out: one for the standard ResultSet and two for the ERC (AL, NL). No test was carried out for RL based on the same arguments presented for the *Update* statement. Figure 19 shows graphically the obtained results and Figure 20 shows the ratios between the results of ResultSet and the results of each ERC (AL, NL).

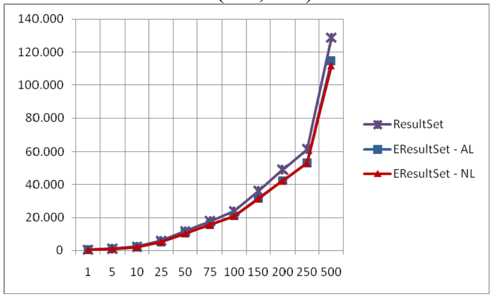


Figure 19. Results of assessments for the Insert statement

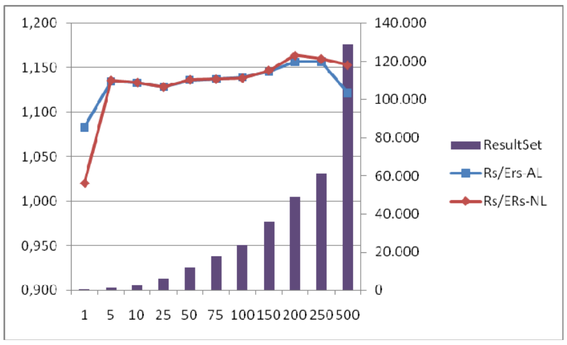


Figure 20. Ratios between ResultSet and EResultSet results for the Insert statement

Figure 19 and Figure 20 show us that, on the contrary of the update statement, the standard ResultSet performance and the ERC performance have some relevant differences. ERC performance is always better than standard ResultSet one. When more than 5 threads are running simultaneously, ERC performance is about 110% to 115% of the standard ResultSet.

2) Changes on context of cursors

Changes on context of cursors deserve a closer attention. Figure 21 shows the ratios between AL and all other used policies for each SQL statement. Figure 21 stresses the idea that changes on context of cursors may have a huge impact on ERC performance. While statements *Update* and *Insert* seem (effectively they are not exempt) exempt from their impact, *Select* statement may have a performance decay about 30-40%. This realization confirms our concern that this is a key issue needing an additional attention in order to reduce its negative effect.

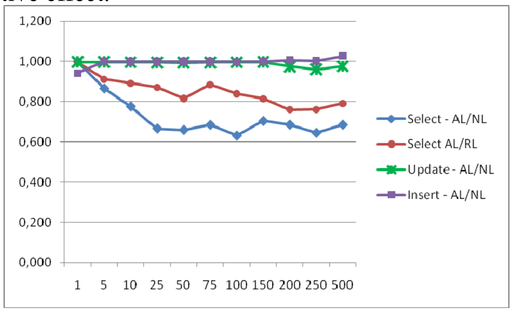


Figure 21. Ratios between AL, NL and LR policies

VI. CONCLUSION

ERC proved to be a promising solution for situations were performance and concurrency are considered key issues. Results show that ERC performance is better than the standard ResultSet performance practically in all situations. Exceptions exist only in cases where the number of concurrent threads is very low. The most outstanding results were achieved for the *Select* statement. Here the gain in performance may achieve 1.8 times the performance of the standard ResultSet.

The assessment took place under optimal conditions. In real situations, networks are shared and very often congested and database servers are overloaded. Therefore, in real situations it is expected to obtain better results for the ERC than the ones obtained in this one.

Regarding *ERC* usage, its interface is mostly based on the ResultSet interface. Only, some additional protocols are required to create cursors. An effort was done to ensure to current ResultSet users a seamless transition to ERC.

Future work will address to main issues: ERC performance and ERC usability. In order to improve ERC performance we will optimize the boilerplate code by providing typestate [18] oriented interfaces. These interfaces provide state information which may be used to optimize ERC performance, by avoiding the execution of unnecessary code. Additionally, this typestate oriented interfaces will promote ERC with an improved usability. Some tests have been already done and the preliminary results are very promising.

REFERENCES

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

[2] M. Woodside, et al., "The Future of Software Performance Engineering," presented at the FOSE '07- Future of Software Engineering, Minneapolis,MN,USA, 2007.

[3] A. Hejlsberg. (2010 Mar 15). The LINQ Project. Available: http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx

[4] Hibernate. (2010). Hibernate. Available: http://www.hibernate.org/

[5] Oracle. (2010). Oracle TopLink. Available: http://www.oracle.com/technology/products/ias/toplink/index.html

[6] ISO. (2003, 2010) ISO/IEC 9075-3:2003. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134

[7] S. Microsystems. (2010). JDBC Overview. Available: http://java.sun.com/products/jdbc/overview.html

[8] Microsoft. (2010 Mar 18). Microsoft Open Database Connectivity. Available: http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx

[9] Microsoft. (2010). Overview of ADO.NET. Available: http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx

[10] S. Microsystems. (2010 Mar 23). Java Data Objects (JDO). Available: http://java.sun.com/jdo/

[11] Sun.Microsystems. (2010 Feb 25). JPA - Java Persistent API. Available: http://java.sun.com/javaee/technologies/persistence.jsp

[12] W. Cook and A. Ibrahim. (2009). Integrating programming languages and databases: what is the problem? Available: http://www.odbms.org/experts.aspx#article10

[13] S. Microsystems. (Mar). Interface ResultSet. Available: http://java.sun.com/javase/6/docs/api/java/sql/ResultSet.html

[14] Microsoft. (Mar). Recordset Object (ADO). Available: http://msdn.microsoft.com/en-us/library/ms681510(VS.85).aspx

[15] Spring. (2010). Spring. Available: http://www.springsource.org/

[16] M. Erik, et al., "LINQ: reconciling object, relations and XML in the .NET framework," in ACM SIGMOD International Conference on Management of Data, Chicago,IL,USA, 2006, pp. 706-706.

[17] S. Microsystems. (Mar). Interface Statement. Available: http://java.sun.com/javase/6/docs/api/java/sql/Statement.html

[18] 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.