

A Concurrent Tuple Set Architecture for Call Level Interfaces

Óscar Mortágua Pereira¹, Rui L. Aguiar²

Instituto de Telecomunicações
DETI – University of Aveiro
Aveiro, Portugal
{omp¹, ruilaa²}@ua.pt

Maribel Yasmina Santos

Centro Algoritmi
DSI – University of Minho
Guimarães, Portugal
maribel@dsi.uminho.pt

Abstract— Call Level Interfaces (CLI) are low level API aimed at providing services to connect two main components in database applications: client applications and relational databases. Among their functionalities, the ability to manage data retrieved from databases is emphasized. The retrieved data is kept in local memory structures that may be permanently connected to the host database. Client applications, beyond the ability to read their contents, may also execute *Insert*, *Update* and *Delete* actions over the local memory structures, following specific protocols. These protocols are row (tuple) oriented and, while being executed, cannot be preempted to start another protocol. This restriction leads to several difficulties when applications need to deal with several tuples at a time. The most paradigmatic case is the impossibility to cope with concurrent environments where several threads need to access to the same local memory structure instance, each one pointing to a different tuple and executing its particular protocol. To overcome the aforementioned fragility, a Concurrent Tuple Set Architecture (CTSA) is proposed to manage local memory structures. A performance assessment of a Java component based on JDBC (CLI) is also carried out and compared with a common approach. The main outcome of this research is the evidence that in concurrent environments, components relying on the CTSA may significantly improve the overall performance when compared with solutions based on standard JDBC API.

Keywords—Call Level Interfaces, O/RM, Concurrency, databases, software architecture.

I. INTRODUCTION

Database applications comprise at least two main components: database components and application components. In our context, application components are developed in the object-oriented paradigm and database components rely on the relational paradigm. The two paradigms are simply too different to bridge seamlessly, leading to difficulties informally known as impedance mismatch [1]. The diverse foundations of both paradigms are a major hindrance for their integration, being an open challenge for more than 50 years [2]. In order to overcome the impedance mismatch issue, several solutions have emerged such as, embedded SQL (SQLJ [3]), language extensions (LINQ [4]), Call Level Interfaces [5] (CLI) (JDBC [6], ODBC [7]), object/relational mappings (O/RM) (Hibernate [8], TopLink [9], LINQ) and persistent frameworks (JDO [10], JPA [11], SDO [12], ADO.NET

[13]). Despite their individual advantages, these solutions have not been designed to manage concurrency on the client side of database applications. Currently, concurrency is managed by database management systems through database transactions. Moreover, whenever the same data is needed by different client-threads, each thread behaves as an independent entity requesting its own data set. In other words, instead of sharing the data returned by a unique execution of a Select expression, each thread executes a Select expression independently from other threads. This leads to a waste of resources, namely it requires more memory, it requires more power computation, and performance is very probably affected negatively. Current tools use local memory structures (LMS) to manage the data returned by Select expressions. Beyond services to read the data kept by LMS, LMS provide services to execute three additional main protocols on their in-memory data: update data, insert new data and delete data. Thus, client-applications are able to update data, insert data and delete data without the need to explicitly execute Update, Insert and Delete expressions, respectively. Once again, these protocols are not thread-safe not promoting this way the use of LMS on concurrent environments. Listing 1 and Listing 2 present a typical case where one table attribute needs to be updated. The value to be used to update the attribute is dependent on the table primary key (PKs). Listing 1 presents the current approach and Listing 2 presents an approach based on thread-safe LMS. In Listing 1 each thread is created and then it runs (*dolt*) to execute a task. Each thread has its own LMS this way preventing any concurrency at the LMS level. Listing 2 presents the equivalent solution based on an approach where all threads share the same

```
void begin() {  
    foreach thread  
        creat thread  
        thread.dolt(PKs)  
    end  
}  
void dolt(PKs) {  
    LMS=execute Select expression  
    while more rows on the LMS  
        if PK is in PKs  
            then update row  
            move to the next row  
        end while  
    }
```

Listing 1. Current approach to update data concurrently.

```

void begin() {
  LMS=execute Select expression
  foreach thread
    create thread
    doIt(LMS,PKs)
  end
}
void doIt(LMS, PKs) {
  while more rows on the LMS
    if PK is in PKs
      then update row
      move to the next row
    end while
  }
}

```

Listing 2. Alternative approach to update data concurrently.

LMS and update the attribute concurrently. In order to overcome the limitations of CLI, this paper proposes a Concurrent Tuple Set Architecture (CTSA). The CTSA, unlike current solutions, provides thread-safe protocols to interact with the data returned by Select expressions.

JDBC and ODBC are two of the most representative standards of CLI. JDBC and ODBC provide, respectively, ResultSet [14] interface and RecordSet [15] interface as their internal implementations of LMS.

The main contributions of this paper are twofold: 1) to present the CTSA based on CLI and with embedded concurrency at the level of LMS; 2) to carry out a performance assessment of a case study based on a JDBC component derived from the proposed architecture. It is expected that the outcome of this paper may contribute to open a new approach to improve the performance of database applications whenever several threads need to share the same LMS instances.

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

The paper is structured as follows: section II presents the required background; section III presents the related work; section IV presents the proposed architecture; Section 4 presents the CTSA; section V presents the performance assessment and Section VI presents the conclusion.

II. BACKGROUND

LMS have been loosely presented and some properties have also been already described. Next follows a more detailed description about the features of LMS.

LMS are instantiated by CLI to manage the data returned by Select expressions. As such, at this point it is advisable to discuss some LMS features that are relevant to this research. Figure 1 presents a general LMS containing 5 tuples (1 to 5) and 6 attributes (a, b, c, d, e, f). This LMS could have been instantiated to manage the data returned by the following CRUD expression: *Select a, b, c, d, e, f from Table Where* In this case, the CRUD expression has returned 5 tuples (rows) and the current selected tuple is row number 2. The access to LMS

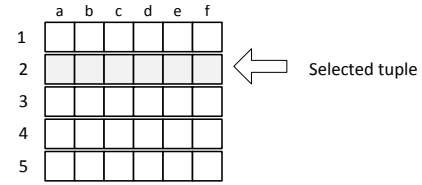


Figure 1. LMS with 5 tuples (rows) and 6 attributes (a till f).

attributes is accomplished by selecting a tuple and then, through an index or through a label (usually the attribute name), by selecting one attribute at a time. For example, to execute an action *action* (read, insert or update) on attribute *c* of tuple 2 the following steps are necessary: select tuple 2 and then execute *action(index of attribute c)* or *action(label of attribute c)*. CLI are responsible for providing services to allow applications to scroll on LMS, to read their contents and to modify (insert, update, delete) their internal contents. Other services are also available but they are not relevant for this research. Services may be split in two categories: basic services and advanced services. Basic services comprise two groups of protocols: the scrolling protocols are aimed at scrolling on tuples and the read protocol is aimed at reading the tuples' attributes. Advanced services are available only if LMS are updatable. In this case applications are allowed to change the internal state of LMS. Advanced services comprise three protocols: insert protocol to add new tuples, update protocol to update existent in-memory tuples and, finally, delete protocol to delete existent tuples. After being committed, the new states of LMS are automatically committed into the host database. To execute any of the previous services it is necessary to know that the access to LMS is simultaneously tuple oriented and protocol oriented. This has two main implications. First, at any time only one tuple may be selected as the target tuple. Second, if a protocol is being executed, applications should not start any other protocol. If this rule is not fulfilled, LMS may lose their previous states. For example, if an advanced service is being executed and another protocol is triggered, LMS discard all changes made during the first protocol. TABLE I concisely presents four protocol that are used to interact with data of LMS.

TABLE I. MAIN PROTOCOLS OF LMS.

ID	Protocol	Id	Protocol
1	Point to a tuple Read attributes	2	Point to a tuple Start update protocol Update attributes Commit update
3	Start insert protocol Insert attributes Commit insert	4	Point to a tuple Delete tuple

Read Protocol: During the read protocol, attributes are read one by one and always from the current selected tuple. If a different tuple is selected, the next attribute value will be retrieved from the new selected tuple. **Update**

Protocol: During the update protocol, attributes are updated one by one on the current selected tuple. The protocol may or may not be triggered by invoking a specific method. It ends when a specific method is invoked to commit the updated attributes. If another tuple or protocol (except the read protocol) is selected while it is being executed, all previous changes will be discarded. **Insert Protocol:** The insert protocol is triggered by invoking a specific method. Then, each attribute is inserted one by one. After all attributes have been inserted, the protocol ends when a specific method is invoked to commit the inserted tuple. If another tuple or protocol (except the read protocol) is selected while it is being executed, all previous changes will be discarded. **Delete Protocol:** The delete protocol comprises a single method that removes the current selected tuple from the in-memory of LMS. The delete action is also automatically committed in accordance with the established policy.

III. RELATED WORK

A research has been carried out around tools aimed at integrating client applications and databases. A survey was made for the most popular tools, such as Hibernate [8], Spring [16], TopLink [17], JPA [11] and LINQ [18]. These tools may provide concurrency but always at a very high level. Basically, they provide some locking policies implemented in order to synchronize read and write actions. But these read and write synchronized actions are not executed over the same memory location. They are executed over distinct objects, such as sessions in Hibernate. These objects (sessions) are not thread-safe and therefore do not provide any protocol to access concurrently the in-memory data.

[19] presents a concurrent version of the TDS protocol [20]. Unlike CTSA, the concurrency is internally implemented at the level of the TDS protocol through the services stacked above the TDS protocol. Authors have achieved significant results for the services they have implemented. Unfortunately, the research only addressed a restrict number of services not leading to a replicable and usable approach.

To the best of our knowledge no other researches have been conducted around concurrency on LMS of CLI.

IV. CTSA

In this section we start to present CTSA and then a proof of concept is also presented.

A. CTSA Presentation

CTSA defines the concept of *execution context* as the information needed to characterize, at any time, the interaction between a thread and a component based on the CTSA. The execution context of each thread comprises the protocol that is being executed and the current selected tuple. This concept is very important because it is the basis for the concurrent implementation of LMS. In concurrent environments, each thread must have a complete control on the tuple and on the protocol it is executing. If this is not ensured, a running thread may be preempted by

another thread that changes the execution context. The first thread will never be aware about this situation and when it becomes the running thread it will execute its actions in a different execution context. In order to keep full control on the execution context, each thread needs to access the LMS in exclusive mode and also to be able to assure that it runs on its own execution context. The former condition ensures that other threads are not allowed to change the execution context of protocols that are being executed. The latter condition ensures that at the beginning of any protocol, if necessary, every thread is able to restore its execution context. To decide upon which strategy to follow to implement both conditions, two possibilities were considered and tested: 1) method oriented: execution context is managed method by method; 2) protocol oriented: execution context is managed at the protocol level. TABLE II briefly shows the logic associated with each approach. The scrolling process involves one method at a time and, therefore, it is implemented as method oriented access mode. Access modes for Insert, Update and Delete protocols do not have any other alternative but be implemented as protocol oriented. This derives from the fact, as mentioned before, that these protocols cannot be preempted to start a different protocol. Read protocol may be implemented in any access mode. To decide upon which access mode to implement some tests with the two access modes were carried out. The collected results have shown, for the same scenarios, that performance and concurrency improvement depend on the same variable but in opposite ways. They depend on the number of times that threads are preempted by other threads. Every time this occurs, a change in the execution contexts must be performed. When this number increases, performance tends to decrease and concurrency tends to increase. When this number decreases, performance tends to increase and concurrency tends to decrease. Thus, in order to improve performance, it was decided to implement the Read protocol based on the protocol oriented access mode.

TABLE II. APPROACHES FOR THE EXCLUSIVE ACCESS MODE.

Method oriented	Protocol oriented
1. get exclusive access 2. set execution context 3. execute method 4. store execution context 5. release exclusive access	1. get exclusive access 2. set execution context 3. while protocol is not over execute method 4. store execution context 5. release exclusive access

Figure 2 presents the interfaces for the five main protocols: IRead (read protocol), IInsert (insert protocol), IUpdate (update protocol), IDelete (delete protocol) and IScroll (scroll protocol). Only the main methods of IRead, IUpdate, IInsert and IScroll have been presented in order not to overcrowd the class diagrams. Exclusive access modes based on the protocol oriented strategy are started by the execution of an explicit starting method (beginRead, beginUpdate and beginInsert) and released only after the execution of another explicit method

(endRead, endUpdate and endInsert). This strategy ensures the exclusive access to LMS while the protocol is being executed and also the initialization of the correct execution context before any access to the LMS. *getInt* and *getString* methods read attributes (read protocol) of types *integer* and *string*, respectively, from LMS. *setInt* and *setString* methods set the values for the attributes (Update and Insert protocols) of type *integer* and *string*, respectively. Beyond these methods (*get* and *set*), there are other methods each one suited to deal with one data type of the host programming language. Exclusive access mode of IScroll methods and IDelete method are method oriented and, therefore, no additional methods are needed.

«interface» IRead	«interface» IInsert
+beginRead() +endRead() +getInt(in idx : long(idl)) : long(idl) +getString(in idx : long(idl)) : string(idl) +...()	+beginInsert() +endInsert() +cancelInsert() +setInt(in idx : long(idl), in value : string(idl)) +setString(in idx : long(idl), in value : string(idl)) +...()
«interface» IScroll	«interface» IUpdate
+moveNext() : bool +moveFirst() : bool +moveAbsolute(in position : int) : bool +isFirst() : bool +...()	+beginUpdate() +endUpdate() +cancelUpdate() +setInt(in idx : long(idl), in value : string(idl)) +setString(in idx : long(idl), in value : string(idl)) +...()

Figure 2. CTSA main protocols.

Figure 3 presents a simplified CTSA class diagram. Concurrent threads sharing the same LMS receive a new CTSA instance where all CTSA instances share the same LMS. *lms* is the LMS instance, *currentTuple* (current selected tuple) and *protocol* (protocol being used) define the execution context of the owner thread, *setExecution* restores the execution context of the running thread and *storeExecution* stores the current execution context.

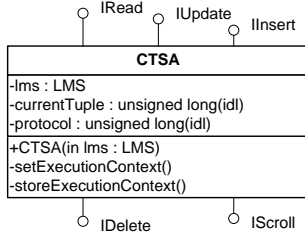


Figure 3. CTSA class diagram.

B. Proof of Concept

This section evaluates CTSA using a proof of concept implemented in Java and JDBC and it uses the *ReentrantLock* [21] to guarantee exclusive access to shared data structures when threads interact with component based on the CTSA. Any other java mechanism, such as *synchronized methods* [22], could have been used. Due to space limitations we will only present CTSA from users' perspective, see Figure 4. The thread receives a CTSA instance (line 23). When the thread enters the running state (line 27), it iterates the LMS one tuple at a time (line 30).

This access mode is method oriented and, as such, there is no starting trigger. The tuple is read (line 32-33). The access mode of the read protocol is protocol oriented and, therefore, there is a trigger to start (line 31) and a trigger to stop it (line 34). This example shows that users of components based on the CTSA have the advantage of using thread safe LMS without any concern about its implementation. The use of the remaining protocols is very similar to the ones presented in Figure 4 and, therefore, no additional examples are needed.

```

23 User( CTSA ctssa ) {
24     this.ctssa = ctssa;
25 }
26
27 @Override
28 public void run() {
29     // ... code
30     try {
31         while ( ctssa.moveToNext() ) {
32             ctssa.beginRead();
33             id = ctssa.getInt( 1 );
34             // ... read other attributes
35             ctssa.endRead();
36             // other protocols
37         } catch ( SQLException ex ) {}
38     }

```

Figure 4. CTSA from users' perspective.

V. CTSA ASSESSMENT

Performance assessment was carried out comparing two entities known as the Component CTSA (C-CTSA) and the Concurrent JDBC (C-JDBC). C-CTSA is responsible for evaluating components relying on the CTSA architecture and it is based on a component derived from the proof of concept here presented. C-JDBC is responsible for evaluating a concurrent approach based on the standard JDBC. The evaluation of both entities comprises a single façade: performance. Three scenarios were defined for both components: Select (*s*), Update (*u*) and Insert (*i*). Each scenario comprises a set of several numbers of tuples to be processed [*nr*] and a set of several numbers of simultaneous running threads [*nt*]. In order to formalize the entities' representation we define $E_{(a,p,\gamma)}([nt], [nr])$ where $a \in \{c\text{-ctsa}, c\text{-jdbc}\}$, p is for performance façade and $\gamma \in \{s, u, i\}$. To simplify, $E_{(a,p,\gamma)}([nt], [nr])$ is represented by default as $E_{(a,p,\gamma)}$. Each scenario comprises a specific goal which is known as a *task*. A task represents a particular case for the use of C-CTSA and C-JDBC regarding the LMS. The tasks to be performed are: Read (read [*nr*] tuples from the LMS), Update (update [*nr*] tuples of a LMS) and Insert (insert [*nr*] tuples into a LMS). It was decided to create a favorable environment to C-JDBC and an unfavorable environment for C-CTSA to execute the defined tasks. This way, the minimum performance of real scenarios should be delimited by the collected measurements. This issue will be addressed in more detail after explaining the SQL Server behavior about LMS.

The test-bed comprises two computers: 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); PC2 – Asus-P5K-VM, Intel Duo Core E6550 @2,33 GHz, 4.00 GB RAM, Windows XP Professional Service Pack 3, SQL Server 2008. C-JDBC and C-CTSA are executed in PC1 and SQL Server runs in PC2. 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 in both PCs; a direct and dedicated network cable connecting PC1 and PC2 has been used in exclusive mode and performing 100MBits of bandwidth. Transactions were not used and *auto-Commit* has been always enabled. A new database was created in conformance with the schema presented in Figure 5 to assess both entities. In order to avoid any overhead added by SQL Server, some default SQL Server database properties were changed as, Auto Update Statistics = false and Recovery Model = Simple.

Std_Student	
Column Name	Data Type
Std_id	int
Std_firstName	varchar(25)
Std_lastName	varchar(25)
StdCrs_id	int
Std_regYear	smallint
Std_applGrade	float

Figure 5. Std_Student schema.

Some important aspects are out of the scope of this study. Aspects as database server performance, network delays and memory consumption are not individually addressed but considered as part of the overall environment. This has been assumed because both entities share the same infrastructure.

It is essential to have some knowledge about SQL Server behavior, which is similar to most of the other relevant relational database management systems, to completely understand the details of each defined task and also to understand the collected results. When a Select statement is executed using a scrollable or an updatable LMS, SQL Server creates a server cursor with all the selected tuples. These tuples are dynamically transferred in blocks, from the server to LMS, whenever necessary. This means that at any time LMS may not have all the tuples but only a sub-set of all tuples. When users point to a tuple that is not present in the LMS, the TDS protocol discards the current LMS content and fetches the block containing the desired tuple. This has a deep implication. If threads are always requesting tuples that are not present in the LMS, SQL Server has to transfer the correspondent block for each request. In an extreme scenario, each individual action over the LMS could imply a new transference of tuples. From the previous statements, it is expected that the number of blocks to be transferred will increase when the number of tuples, inside server cursors, increases and also

when the dispersion of the used policy to select tuples, contained by server cursors, increases. Thus, to create the environments for both entities, the following decisions were taken:

C-JDBC (favorable environment): each thread will always access tuples sequentially from the first one till the last one.

C-CTSA (unfavorable environment): two conditions were implemented: 1) after accessing a tuple, each thread will give the opportunity for other threads to become the running thread. This will maximize the number of changes in the execution context; 2) each thread will have its own set of tuples, not shared with any other thread. This will maximize the number of blocks of tuples to be transferred from the server cursor to LMS.

TABLE III shows the algorithm for the assessment of $E_{(c-ctsa,p,\gamma)}$. The same ResultSet is shared by all $[nt]$ threads. Each thread executes its scenario for a group of $\psi=[nr]$ adjacent tuples and auto-suspends itself after accessing each tuple. The intersection of all $\psi=\emptyset$.

TABLE III. ALGORITHM FOR $E_{(c-ctsa,p,\gamma)}$ ASSESSMENT.

1. Delete all rows from Std_Student
2. Fill Std_Student with $[nr]*[nt]$ rows (zero rows for insert)
3. Start counter
4. Select all rows from Std_Student into one single ResultSet
5. Create all threads. Each thread (ψ tuples)
 - 5.1 for each tuple
 - 5.1.1 read/update/insert (tuple)
 - 5.1.2 suspend thread
 - 5.2 dies
6. Wait all threads to die
7. Stop counter

TABLE IV shows the algorithm for the assessment of $E_{(c-jdbc,p,\gamma)}$. Each thread creates its own ResultSet (LMS) containing/inserting a group of $\psi=[nr]$ adjacent tuples. The intersection of all $\psi=\emptyset$.

TABLE IV. ALGORITHM FOR $E_{(c-jdbc,p,\gamma)}$ ASSESSMENT.

1. Delete all rows from Std_Student
2. Fill Std_Student with $[nr]*[nt]$ rows (zero rows for insert)
3. Start counter
4. Create all threads. Each thread:
 - 4.1 select ψ tuples into its own ResultSet
 - 4.2 for each tuple
 - 4.2.1 read/update/insert a tuple
 - 4.3 dies
5. Wait all threads to die
6. Stop counter

To contextualize the performance assessment environment some initial measurements were carried out to delimit the range of $[nt]$ and $[nr]$ to be used. In order to emphasize concurrency mechanisms, priority was given to the range of $[nt]$ in detriment of $[nr]$. Values for these metrics were collected by empirical experimentation based

on an iterative process. The idea is to gather a set of values for $[nt]$ and $[nr]$ that may be used to assess and compare the performance of both $E_{(a,p,\gamma)}$ entities. To accomplish this, both entities, $E_{(c-ctsa,p,\gamma)}$ and $E_{(c-jdbc,p,\gamma)}$ were executed under several combinations of $[nt]$ and $[nr]$ until the collected values comprise a range of behaviors considered satisfactory to accurately assess and compare the performance of both entities. After several iterations it was decided that a reliable execution environment should be defined as:

$[nt]=\{1, 5, 10, 25, 50, 75, 100, 150, 200, 250, 350, 500\}$
 $[nr]=\{5, 10, 25, 50, 75, 100\}$

In accordance with the requirements, this execution environment evaluates the performance by maximizing the number of simultaneous running threads in detriment of the number of tuples. With 500 threads and 100 tuples it was possible to accurately assess and foresee the performance of both entities. This was the main reason for their acceptance. The intermediate collected values showed to be enough to obtain well defined charts for the behaviors of both entities. Just as a final note, some scenarios took some minutes to setup and to process the highest values of $[nt]$ and $[nr]$. This knowledge was also considered to delimit the two top values ($nr=100$ and $nt=500$), this way avoiding any risk to successfully accomplish the collecting process of all necessary measurements. 25 raw measures were collected for each $E_{(a,p,\gamma)}([nt],[nr])$ leading to $(2 \times 3 \times 12 \times 6) \times 25 = 10,800$ raw measurements. Intermediate measurements were computed from the average of the 5 best measures of each $E_{(a,p,\gamma)}([nt],[nr])$ leading to a total of $2 \times 3 \times 12 \times 6 = 432$ measurements. The final measurements used in the next charts represent the ratios between $E_{(c-jdbc,p,\gamma)}$ and $E_{(c-ctsa,p,\gamma)}$ for each $([nt],[nr])$. In all charts the vertical axis is for the ratios and the horizontal axis is for the $[nt]$.

Select scenario: the chart for the select scenario is shown in Figure 6. From it, it is clear that the ratios decrease whenever the number of tuples increases and whenever the number of threads increases. $E_{(c-jdbc,p,\gamma)}$ have $[nt]$ server cursors and each thread sequentially reads its own tuples from the first one till the last one. Thus, the transference of block of tuples only happens when a thread tries to read the next tuple that is after the last one contained in the ResultSet. Moreover, for this entity different threads do not compete for the same ResultSet this way avoiding any randomness in the tuples to be selected. Regarding $E_{(c-ctsa,p,\gamma)}$ there is only one server cursor shared by all threads. The implemented Read task significantly increases the possibility of each thread to be requesting a tuple that is not present in the ResultSet and, therefore, to trigger a new transference of block of tuples. With other strategies where threads read shared sets of tuples, the block transference rate should be much lower, this way increasing the ratios between the two entities. Another relevant issue is that the Select scenario is a light scenario mainly because the Select expression and the read protocol are very efficient when compared with the other CRUD expressions and other protocols. Thus, the

overhead induced by the blocks transference have a deeper impact in the overall performance. The impact increases with the number of tuples and the number of threads as expected. Anyway, the collected results show that for lower values of number of tuples and lower values of number of threads the ratios vary between 1.02 and 3.44 times, as shown in Figure 7. It may also be seen that the worst ratio achieves 0.80 for $nt=100$ and $nr=25$. These results show that despite the unfavorable test conditions for C-CTSA, C-CTSA still achieves significant results. For example, the relative highest gain in performance (3.44) is much more significant than the relative highest lost in performance (0.8). Figure 7 presents a detailed vision for the ratio between both entities for all combinations of $[nr]$ and $[nt]$.

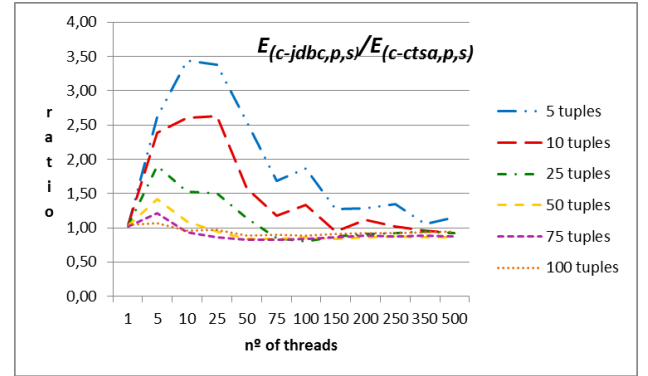


Figure 6. $E_{(c-jdbc,p,s)} / E_{(c-ctsa,p,s)}$ chart.

NR/NT	5	10	25	50	75	100
1	1,03	1,04	1,03	1,03	1,02	1,05
5	2,61	2,38	1,90	1,41	1,22	1,07
10	3,44	2,60	1,53	1,09	0,93	0,96
25	3,38	2,63	1,51	0,94	0,86	0,97
50	2,54	1,57	1,14	0,83	0,83	0,89
75	1,69	1,18	0,85	0,84	0,83	0,89
100	1,86	1,34	0,80	0,86	0,84	0,88
150	1,28	0,95	0,87	0,84	0,86	0,91
200	1,28	1,12	0,90	0,87	0,88	0,92
250	1,35	1,02	0,92	0,87	0,87	0,92
350	1,06	0,95	0,94	0,86	0,89	0,93
500	1,16	0,92	0,93	0,86	0,87	0,94

Figure 7. $E_{(c-jdbc,p,s)} / E_{(c-ctsa,p,s)}$ details.

Update scenario: the chart for the update scenario is shown in Figure 8. The comments made to the Select scenario are also applied to the Update scenario regarding the transference rate of block of tuples. The most significant differences are: 1) the update protocol is a heavy protocol and, thus, its overhead has a deep impact on both entities and in the collected measurements; 2) the $E_{(c-jdbc,p,\gamma)}$ entity has $[nr]$ server cursors, each one competing with the others to update the same requested attributes while $E_{(c-ctsa,p,\gamma)}$ entity has only one server cursor and the competition is performed at the client side. Despite the unfavorable conditions for C-CTSA, in this scenario, the ratio is always significantly greater than 1. It increases in the range $1 < nt < 10$ and for $nt > 10$ the ratios are practically stable for each individual $[nr]$ (except for $nr=5$). Another relevant issue is that the ratios decrease

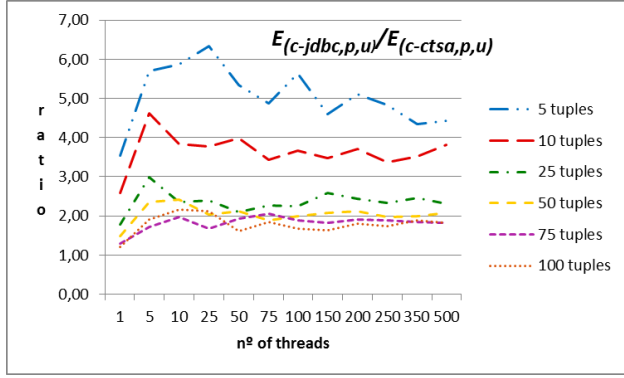


Figure 8. $E_{(c-jdbc,p,u)} / E_{(c-ctsa,p,u)}$ chart.

when $[nr]$ increases for every $[nt]$.

Insert scenario: the chart for the insert scenario is shown in Figure 9. The most relevant aspect is the slight but constant ratios increase with $[nt]$ for each $[nr]$. In the initial stage, ResultSets are empty and tuples are sequentially inserted and committed one by one in the host database table. In this scenario, in opposite to the others, all $E_{(c-ctsa,p,i)}$ threads insert adjacent tuples this way minimizing the number of blocks to be transferred. In spite of being a very heavy scenario for both entities, the differences between C-CSTA and C-JDBC are enough to be noticed in the ratios. It is always greater than 1 and higher values of $[nr]$ cause a decreasing in the ratios.

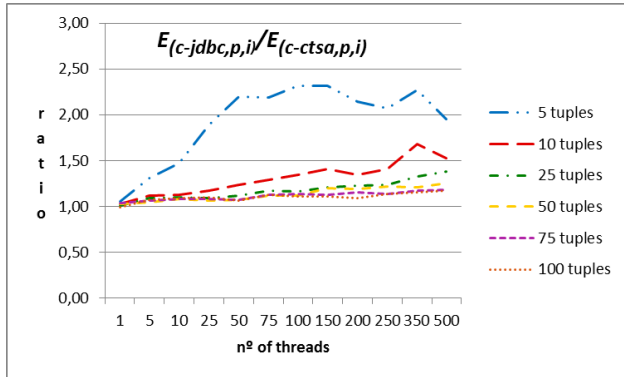


Figure 9. $E_{(c-jdbc,p,i)} / E_{(c-ctsa,p,i)}$ chart.

VI. CONCLUSION

In this paper an architecture for a concurrent LMS, herein known as CTSA, has been presented. A proof of concept has also been presented based on a standard JDBC API. In order to assess CTSA performance in a concurrent environment and compare it with an equivalent environment based on a standard JDBC solution, a test-bed has been defined and implemented with two concurrent entities: C-JDBC and C-CTSA. C-CTSA was assessed in unfavorable conditions and C-JDBC has been assessed in favorable conditions in order to delimit and evaluate C-CTSA performance minimum gain. In spite of these conditions, C-CTSA always gets better scores for the update and for the insert scenarios. In the Select scenario,

C-CTSA obtained significant scores in the range of lower values of $[nr]$ and $[nt]$. Anyway, for higher values of $[nr]$ and $[nt]$ the minimum ratio did not go below 0.8 which is still a remarkable score.

The outcome of this research should encourage CLI providers to release CLI with internal embedded concurrency. Embedded concurrency has the advantage of accessing the LMS's internal data structures to optimize the implementation of the different protocols.

REFERENCES

- [1] M. David, "Representing database programs as objects," Advances in Database Programming Languages, F. Bancilhon and P. Buneman, eds., pp. 377-386, N.Y.: ACM, 1990.
- [2] W. Cook, and A. Ibrahim, "Integrating programming languages and databases: what is the problem?," 2011 May: ODBMS.ORG, Expert Article, 2005.
- [3] Part 1: SQL Routines using the Java (TM) Programming Language, 1999.
- [4] D. Kulkarni, L. Bolognese, M. Warren et al., "LINQ to SQL: .NET Language-Integrated Query for Relational Data," Microsoft.
- [5] ISO. "ISO/IEC 9075-3:2003," [2011 May; http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134.
- [6] M. Parsian, JDBC Recipes: A Problem-Solution Approach, NY, USA: Apress, 2005.
- [7] Microsoft. "Microsoft Open Database Connectivity," Jul, 2012; [http://msdn.microsoft.com/en-us/library/ms710252\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx).
- [8] B. Christian, and K. Gavin, Hibernate in Action: Manning Publications Co., 2004.
- [9] Oracle. "Oracle TopLink," Oct, 2011; <http://www.oracle.com/technetwork/middleware/toplink/overview/index.html>.
- [10] Oracle. "Java Data Objects (JDO)," 2011 Nov; <http://www.oracle.com/technetwork/java/index-jsp-135919.html>.
- [11] D. Yang, Java Persistence with JPA, pp. 390: Outskirts Press, 2010.
- [12] IBM. "Introduction to Service Data Objects," [2011 Nov; <http://www.ibm.com/developerworks/java/library/j-sdo/>.
- [13] G. Mead, and A. Boehm, ADO.NET 4 Database Programming with C# 2010, USA: Mike Murach & Associates, Inc., 2011.
- [14] Oracle. "ResultSet," 2012 Jul; <http://docs.oracle.com/javase/6/docs/api/java/sql/ResultSet.html>.
- [15] Microsoft. "RecordSet (ODBC)," 2011 Jun; <http://msdn.microsoft.com/en-us/library/5sbfs6f1.aspx>.
- [16] Spring. "Spring," 2011; <http://www.springsource.org/>.
- [17] "Oracle Database," 2010 May; <http://www.oracle.com/us/products/database/index.html>.
- [18] M. Erik, B. Brian, and B. Gavin, "LINQ: Reconciling Object, Relations and XML in the .NET framework," in ACM SIGMOD Intl Conf on Management of Data, Chicago, IL, USA, 2006, pp. 706-706.
- [19] D. Gomes, Ó. M. Pereira, and W. Santos, "JDBC (Java DB connectivity) concorrente," DETI, University of Aveiro, ria - institutional repository, <http://hdl.handle.net/10773/7359>, 2011.
- [20] Microsoft. "[MS-TDS]: Tabular Data Stream Protocol Specification," 2012 Jul; [http://msdn.microsoft.com/en-us/library/dd304523\(v=prot.13\).aspx](http://msdn.microsoft.com/en-us/library/dd304523(v=prot.13).aspx).
- [21] Oracle. "Call ReentrantLock," 2012 Nov; <http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/locks/ReentrantLock.html>.
- [22] Oracle. "Synchronized Methods," 2012 Nov; <http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>.