Concurrent Call Level Interfaces

Based on an Embedded Thread Safe Local Memory Structure

Óscar Mortágua Pereira
Instituto de Telecomunicações
DETI – University of Aveiro
Aveiro, Portugal
omp@ua.pt

Rui L. Aguiar
Instituto de Telecomunicações
DETI – University of Aveiro
Aveiro, Portugal
ruilaa@ua.pt

***Abstract*— Performance is traditionally considered one of the most significant concerns in intensive database applications. Several architectural tactics may be taken to minimize the possibility of coming across with any performance bottleneck. One of them is the usage of Call Level Interfaces (CLI). CLI are low level API that provide a high performance environment to execute SQL statements on relational and also on some NoSQL database servers. In spite of this, CLI are not thread safe, this way preventing distinct threads from sharing datasets retrieved from databases through Select statements. Thus, in situations where two or more threads need to access datasets retrieved from the same Select statement, there is no other alternative than providing each thread with its own dataset, this way consuming important computational resources. In this paper we propose a new design for CLI to overcome the aforementioned drawback. Unlike current implementations of CLI, now they are natively thread-safe. The implementation herein presented is based on a thread safe updatable local memory structure where data retrieved from databases is kept. A proof of concept based on Java Database Connectivity type 4 (JDBC) for SQL Server 2008 is presented and also a performance assessment.**

Keywords— call level interfaces, concurrency, performance, databases, middleware, 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 be bridged 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]. These challenges are especially noticeable in environments where code production is under strict development deadlines and where code development efficiency is a major concern. In order to overcome the impedance mismatch issue, several solutions have emerged [1][2][3][4][5][6]. Despite their individual advantages, these solutions have not been developed to address situations where users need to implement concurrent mechanisms over the in-memory data structures returned by Select statements. As generally accepted, performance is one of the most challenging non-functional software requirements in database applications. Here, Call Level Interfaces (CLI) have to be considered as a promising alternative [2]. CLI are programming API aimed at easing the integration of client software components with database components. They use native SQL statements, this way promoting the SQL expressiveness and the SQL performance. Nevertheless, CLI do not provide some of the most well-known and common features to improve system performance, being concurrency the most paradigmatic case. We cannot forget that the speed of data creation and data storage increases every passing day, which is followed by an increased need of power computation to process it. Very often, part of the increased need of power computation derives from the incapacity of current systems to share resources that have already been become available. For example, let us consider a Select statement that retrieves a dataset from a database, which is kept in local memory structures (LMS): ResultSet [7] and RecordSet [8] are examples of LMS for JDBC and ODBC, respectively. Probably, the data contained in LMS could be made available and shared concurrently to several consumers (threads). This possibility would eliminate the need to retrieve and duplicate the same datasets over and over again for each thread (in different LMS instances). To achieve this goal, the access to LMS needs to be thread-safe to avoid unwanted conflicts. Unfortunately, current tools used to develop business logics are not thread-safe. They were mainly designed to minimize the impedance mismatch between the object-oriented and the relational paradigms. Among those options, CLI are considered the best option whenever performance is a non-functional key requirement [9]. For this reason, CLI were chosen to design a thread-safe API to be used on the building process of business logics. The proposal herein presented is based on a modification on the native internal LMS structure in order to make it natively thread-safe. This means that CLI are now implemented with embedded concurrent mechanisms, in opposite to [10][11]. A proof of concept based on Java Database Connectivity type 4 (JDBC) is presented. A performance assessment is also conducted in order to evaluate the performance of both architectures.

The remainder of this paper is organized as follows: chapter II presents the motivation for this research, chapter III describes the current state of the art, chapter IV presents the required background to keep this paper as self-contained

DOI reference number: 10.18293/SEKE2017-120

as possible, chapter V presents the proposal for thread-safe CLI**,** chapter VI presents a performance assessment and, finally, chapter VII presents the final conclusion.

II. MOTIVATION

In this section we present the limitations of current CLI and the goal we want to achieve. The presentation is based on simple examples to avoid any discomfort of readers less knowledgeable about CLI.

The scenario to be addressed comprises situations where there is the need to concurrently share access to datasets retrieved by Select statements. These datasets are managed by LMS. The access to data contained by LMS is row and attribute oriented. This means that at any given moment just one row may be selected and, then, the access to data is processed by selecting one attribute at time. After being processed, another row may be selected and the process continues. If no other control is implemented, this context is not compatible with multi-thread environments. Listing 1 depicts a situation where two threads are using the same LMS instance (lms). Regarding thread A, it scrolls to the next row and then it reads the first attribute. Thread B moves to row number 10 and then it reads the third attribute. In preemptive multitasking [12] environments, these two threads may enter in a conflict state. Suppose that Thread A is the running and it scrolls to line 5. If Thread B becomes the running thread, it will move to row 10, it reads the attribute number 3 and then voluntarily it suspends itself. When thread A is resumed, it will read the attribute number 1, not from row 5 as initially expected but wrongly from row 10. CLI do not provide any feature to prevent this from happening. To overcome this situation, an initial approach has been already proposed to overcome this CLI drawback, which it is based on a wrapper that hides the functionalities of CLI and exposes thread safe services [10][11].

```
// Thread A
1 lms.moveNextRow();
2 id=lms.read(1);
3 …
```

```
// Thread B
1 lms.moveToRow(10);
2 name=lms.read(3);
3 thread.suspend();
4 …
```

Listing 1. Two threads accessing the same LMS.

III. STATE OF THE ART

A survey has been carried out around tools aimed at integrating client applications and databases. The survey comprises the most popular tools, such as Hibernate [4], Spring [13], TopLink [14], JPA [5] and LINQ [15]. These tools may provide concurrency but always at a very high level. Basically, they provide some locking policies 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 contained on LMS.

A survey has also been carried out about two approaches proposed by the research community: SQL DOM [16] and Safe Query Objects [17]. SQL DOM 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. Safe Query Objects 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. These proposals are focused on minimizing the impedance mismatch. None of these approaches address concurrency at any level.

In [18] a different approach is presented to address the lack of concurrent mechanisms of CLI. Concurrency is implemented by an explicit locking mechanism based on two methods: *lock()* and *unlock*. Programmers are responsible for invoking these methods correctly in order to control the exclusive access mode to LMS. Additionally, the conducted assessment is based on a fixed number of rows which does not convey a dynamic perspective of the performance for different scenarios.

Aspect-oriented programming [19] community considers persistence as a crosscutting concern [20]. Several works have been presented but none addresses the points here under consideration. The following works are emphasized: [21] is focused on separating scattered and tangled code in advanced transaction management; [20] addresses persistence relying on AspectJ; [22] presents AO4Sql as an aspect-oriented extension for SQL aimed at addressing logging, profiling and runtime schema evolution. It would be interesting to see an aspect-oriented approach for the points herein under discussion.

The research presented in [11][10] proposes an architecture based on a wrapper which hides the CLI functionalities and exposes thread-safe services. It is known as CTSA – Concurrent Tuple Set Architecture. This approach is clearly an improvement when compared with the one presented in [18] but the thread-safe mechanisms are not embedded on CLI as proposed in this research. Nevertheless, that approach will also be used to compare their results with the results obtained by the approach herein proposed. As it will be shown, the herein presented approach clearly improves the performance achieved with CTSA.

In this master thesis [23] an identical approach, as the one herein presented, has been designed but the final results were not convincing. This paper presents a new implementation.

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

IV. BACKGROUND

In this section we present the necessary background to make this a self-contained paper. It is divided in two main sub-sections. In the first one, some fundamental functionality of LMS are provided and in the second one a brief description is given how CLI and Relational Database Management Systems (RDBMS) interact with each other.

A. Functionality of LMS

LMS are client-side object-oriented abstractions of a

relational concept: the server side cursor. LMS are instantiated to manage relations returned by Select expressions. At instantiation time, some runtime properties of LMS are defined to characterize their functionalities. Two main groups of functionalities are herein emphasized: scrolling functionalities and accessing functionalities (they are orthogonal). Scrolling functionalities provide two main types of LMS (they are mutual-exclusive): forward-only – rows are read sequentially from the first one till the last one, and scrollable – rows can be randomly read. Accessing functionalities provide two main types of LMS (they are mutual-exclusive): read-only and updatable LMS. While read-only LMS only provide one protocol to read their contents, updatable LMS, beyond the read protocol, also provide three additional protocols: update (to update their contents), insert (to insert new rows) and delete (to delete existing rows). Another relevant issue is the mechanism implemented for each protocol (read, update, insert and delete). LMS are row oriented and protocol oriented. This has two main implications. First, at any time only one row can be selected as the target row. Second, if an update or insert protocol is being executed, applications cannot start any another protocol. If this rule is not fulfilled, LMS discard changes made during the previous protocol. Table 1 concisely presents how the 4 main LMS protocols work: 1 – read protocol, 2 – update protocol, 3 – insert protocol and 4 – delete protocol.

Table 1. 4 main protocols of LMS.

1	Point to a row Read attributes Point to another row	2	Point to a row Update attributes Commit update
3	Start insert Insert attributes Commit insert	4	Point to a row Delete row

B. *Interaction Between CLI and RDBMS*

The communication between CLI and RDBMS relies on proprietary protocols of RDBMS vendors but their general interaction follows the structure presented in Figure 1. When a Select expression is executed, RDBMS create a server dataset with the retrieved data and also a server cursor. All or only a part of the retrieved data is copied from server datasets to LMS depending on the LMS properties. When data is partially transferred to LMS, new blocks of data are transferred whenever client applications need to access data not contained locally in the LMS. The relationship between LMS and cursors, and between cursors and LMS are all 1 to 1. This means that whenever a Select statement is executed, there will be one additional LMS and one additional server cursor. We emphasize that one additional means resources (LMS, cursors and datasets) that are being replicated, very

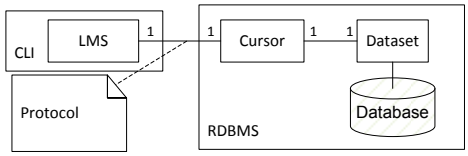

Figure 1. Connection between LMS and RDBMS.

probably unnecessarily. In scenarios where there are simultaneously several server datasets from the same Select statement, we are before a situation where there is an unnecessarily wasting of computational resources.

V. PROPOSED ARCHITECTURE FOR CONCURRENT CLI

In this section we present the architecture that has been defined to design concurrent LMS. To achieve the proposed goal, some source code of CLI was redesigned. In this research, in opposite to CTSA, we explored the usage of embedded thread-safe LMS on which threads interact directly with the data retrieved from databases. To achieve this goal, several CLI interfaces (services) need to be rewritten, such as those aimed at: scrolling on LMS, reading data from LMS, updating data on LMS, inserting data on LMS and, finally, deleting data on LMS. These are the fundamental interfaces responsible for providing services through which client applications are currently able to interact with LMS. A new concurrent component, known as CLMS (Concurrent LMS), replaces the default LMS. Figure 2 presents the main architecture of CLMS which contains a local cache to store the retrieved data. Basically it implements a general interface (ICLMS) which extends the 6 fundamental additional interfaces: ICForwardOnly, ICScrollable (forward-only and scrollable CLMS, respectively) and ICRead, ICUpdate, ICInsert and ICDelete (read update, insert and delete protocols, respectively). Please remember that these are the basic services required to interact with CLMS, as previously mentioned in chapter IV.

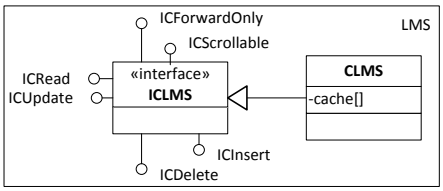

Figure 2. Concurrent architecture for CLMS.

Before, presenting some additional details, the concept of execution context is introduced. Each running thread has its own execution context which consists on the protocol being executed (if any) and the current selected row. This is an important concept because it is based on it that it was possible to design a thread-safe LMS. Basically, every time a thread enters the monitor (thread-safe area) it needs to set its execution context and before leaving the monitor it must store its execution context. This process will ensure that each thread, whenever initiating the access to the monitor, it is able to restore its previous execution context (protocol being executed and row being accessed). Now, we can introduce how an exclusive access mode can be implemented to access CLMS. Two methodologies are proposed: method oriented access mode and protocol oriented access mode. Basically, the method oriented access mode requires a restoring and storing process for the execution context every time a method is executed on CLMS, while the protocol oriented access mode does only require a restoring and storing process by each protocol being executed. Let us take a closer look to the protocols to evaluate the options that are available for each one. The scrolling protocol involves one method at a

time and, therefore, the obvious approach is the method oriented access mode. Access modes for Insert, Update and Delete protocols do not have any other alternative but being implemented as protocol oriented access mode. As mentioned before, this derives from the fact that these protocols, while being executed, cannot be preempted to start any other different protocol. Read protocol may be implemented in any access mode protocol: method access mode (operating on an attribute by attribute basis) or protocol access mode (operating on a row by row basis). In order to implement the exclusive access mode to CLMS it was decided, based on practical evidence and empirical experience, to use method oriented access mode for the ICForwardOnly and ICScrollable interfaces and protocol oriented access mode for the remaining interfaces. We assume that in the most common situations, several attributes are read in each Read protocol, this way not advising the method oriented access mode. With the thread-safe LMS the resort to multiple cursors in the database server is avoided. A single server cursor is able to satisfy simultaneously several client side threads sharing the same LMS.

VI. PERFORMANCE ASSESSMENT

A performance assessment was carried out to compare a solution based on a traditional non-shared (S-JDBC) LMS and the one herein proposed (C-JDBC). The performance assessment ran in a context completely identical to one used in CTSA, this way ensuring that the collected results can be compared to evaluate the impact of thread-safe LMS.

C-JDBC uses a unique LMS that is shared by all threads, while in S-JDBC each thread has its own LMS. All LMS contain the same relation returned by the same Select expression. Concisely, Figure 3, presents the block diagram for the used scenario during the assessment process.

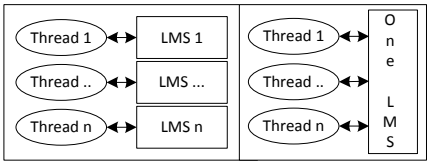

Figure 3. Left side: S-JDBC, right side: C-JDBC.

Three scenarios were defined for the main operations of both solutions: Select (*s*), Update (*u*) and Insert (*i*). Each scenario comprises a set of several numbers of rows to be processed *[nr]* for both S-JDBC and C-JDBC, and a set of several numbers of simultaneous running threads *[nt]* for both S-JDBC and C-JDBC. In order to formalize the entities' representation the following definition is presented: $E_{(\alpha,\gamma)}$ *([nt], [nr])* where $\alpha \in$ {c-jdbc, s-jdbc}, and $\gamma \in \{s,u,i\}$. To simplify the general formalization, $E_{(\alpha,\gamma)}$ *([nt], [nr])* is represented by default as $E_{(\alpha,\gamma)}$. Each scenario comprises a specific goal which is known as a *task*. A task represents a particular case for the use of C-JDBC and S-JDBC. The tasks to be performed are: Read (read *[nr]* adjacent tuples from the LMS), Update (update *[nr]* adjacent rows of a LMS) and Insert (insert *[nr]* tuples into a LMS). Please remember that S-JDBC uses *[nt]* LMS while C-JDBC always uses one LMS. The assessment could also comprise a random access pattern but, by empirical evidence, the most common access pattern is the access to adjacent rows. It was also decided to create and enforce different contexts for S-JDBC and C-JDBC. The idea is to create a context to C-JDBC based on tactics aimed at decreasing its performance while executing the defined tasks while for S-JDBC we will use tactics to enforce the opposite effect. S-JDBC favorable tactics – each thread has its own LMS and rows are always sequentially selected in order to minimize the transference of rows between JDBC and SQL Server. C-JDBC unfavorable environment - two situations were implemented: 1) Each thread will auto-suspend itself after having executed one protocol: read one row, update one row or insert one row. This will give the opportunity to other thread to become the running thread, this way maximizing the number of changes in the execution contexts. 2) All threads share the LMS but each thread has its own adjacent rows. This means that when a thread becomes the running thread, its execution context will set a row that belongs to a different set of rows, this way maximizing the number of blocks to be transferred from SQL Server.

Table 2 presents the algorithms used to assess S-JDBC and C-JDBC. All scenarios, for each solution, share the same algorithm for the assessments to be carried out. C-JDBC and S-JDBC create the same number of threads (*nt*) and each thread processes the same number of rows (*nr*). The main difference is: while in S-JDBC each thread selects its own subset of rows, this way accessing its own LMS, in C-JDBC a LMS is shared by all threads containing all rows.

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 is executed in PC1 and SQL Server runs in PC2. In order to promote an ideal environment the following actions were taken: CPU were set to run with a single core, this way maximizing the influence of the implemented solutions; the running threads were given the highest priority; all non-essential processes/services were cancelled in both PCs and a direct and dedicated network cable connecting PC1 and PC2 has been used in exclusive mode and performing 100MBits of bandwidth. 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.

The sets used for the number of rows and for the number of threads were:

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

[nr]={5, 10, 25, 50, 75, 100}

25 raw measures were collected for each $E_{(\alpha,\gamma)}$*([nt],[nr])* leading to (2x3x12x6)x25=10,800 raw measures. Intermediate measures were computed from the average of the 5 best measures of each $E_{(\alpha,\gamma)}$*([nt],[nr])* leading to a total of 2x3x12x6=432 measures. The final measures used in the next charts represent the ratios between $E_{(c\text{-}jdbc,\gamma)}$ and $E_{(s\text{-}}$

$_{jdbc,\gamma)}$ for each *([nt],[nr])*. In all charts the vertical axis is for the ratios and the horizontal axis is for the *[nt]*.

A table Student with the following schema was also created to store the data being used: id (int, pk), firstName (varchar 25), lastName (varchar 25), crdId (int), regYear (int), applGrade (float).

Table 2. Algorithms for $E_{(c\text{-}jdbc,\gamma)}$ II-Algorithms for $E_{(s\text{-}jdbc,\gamma)}$.

I	1. Delete all rows from *Student* 2. Fill *Student* with *[nr]*[nt]* rows (zero rows for insert) 3. Start counter 4. Select all rows from *Student* into one single ResultSet 5. Create all threads. Each thread (ψ tuples) 5.1 for each row 5.1.1 read/update/insert (row) 5.1.2 suspend thread 5.2 dies 6. Wait all threads to die 7. Stop counter
II	1. Delete all rows from *Student* 2. Fill *Student* with *[nr]*[nt]* rows (zero rows for insert) 3. Start counter 4. Create all threads. Each thread: 4.1 select ψ trow into its own ResultSet 4.2 for each row 4.2.1 read/update/insert a tuple 4.3 dies 5. Wait all threads to die 6. Stop counter

Select scenario

Figure 4 presents the ratio between the measures collected for $E_{(s\text{-}jdbc,s)}$) and $E_{(c\text{-}jdbc,s)}$. The chart shows that there are some situations where the gain is very significant. The most significant situation occurs for 5 tuples and 10 to 25 threads reaching a gain above 3.5 times. The ratio decreases when the number of rows increases and also when the number of threads increases. The reasons for this behavior is that when either *nt* or *nr* increases, the probability of a thread of C-JDBC to access a row not contained in the LMS increases, this way requiring a block transfer from the server dataset to the LMS. Please remember that in C-JDBC all threads share the same LMS and each thread is reading a different block of adjacent rows. This means that when *thread 1* reads row 1, *thread n* is reading row *nr*(n-1)* which eventually may not be at that moment contained in the local LMS.

Figure 5 presents a tabular view of the chart presented in

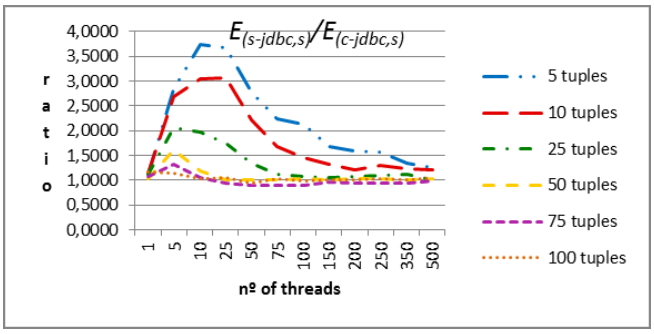

Figure 4. Select scenario: ratio between S-JDBC and C-JDBC.

NR/NT	5	10	25	50	75	100
1	1,13	1,11	1,13	1,13	1,09	1,14
5	2,68	2,47	1,99	1,50	1,32	1,16
10	3,54	2,69	1,63	1,17	1,01	1,04
25	3,48	2,73	1,59	1,02	0,96	1,04
50	2,61	1,67	1,21	0,90	0,91	0,97
75	1,76	1,26	0,92	0,94	0,91	0,97
100	1,96	1,42	0,87	0,95	0,93	0,98
150	1,38	1,04	0,95	0,94	0,96	0,98
200	1,37	1,19	0,97	0,97	0,95	1,00
250	1,42	1,10	1,01	0,95	0,94	0,99
350	1,14	1,04	1,03	0,93	0,98	1,03
500	1,23	1,00	1,02	0,94	0,94	1,03

Figure 5. Tabular view for the ratio between S-JDBC and C-JDBC for the Select scenario.

Figure 4. It shows that in some cases the ratio is not greater than one which means that measures of $E_{(c\text{-}jdbc,s)}$ are not always better than measures of $E_{(s\text{-}jdbc,s)}$. But the key issue is that when compared with ratios obtained with CTSA, C-JDBC has a mean improvement around 6%. Additionally, in this assessment only about 25 ratios are under 1.00, in opposite to 40 in CTSA. This improvement is due to the embedded thread-safe mechanisms. CTSA was based on a wrapper and, therefore, an overhead is an unavoidable issue which was solved in the approach herein presented. The results here obtained show that $E_{(c\text{-}jdbc,s)}$ has achieved outstanding results even when compared with the CTSA.

Update and Insert scenarios

The update scenario updates rows contained by LMS and the insert scenario inserts rows in empty LMS. The measures collected for the Update and also for the Insert scenario were very close to ones collected for CTSA. The basic reason for these results is that these protocols are much heavier than the Select protocol and, therefore, the achieved gains, in C-JDBC when compared with CTSA, have a much lower impact. Figure 6 presents the graphic for the Update scenario and Figure 7 presents the graphic for the Insert scenario for the ratios S-JDBC/C-JDBC.

The chart for Update scenario shows that the gain is always greater than 1 and it increases when the number of rows decreases. The number of threads seems to not have a significant impact. This behavior is understandable if we remind that the update protocol is very heavy and its weight can be much more influent than the weight associated with

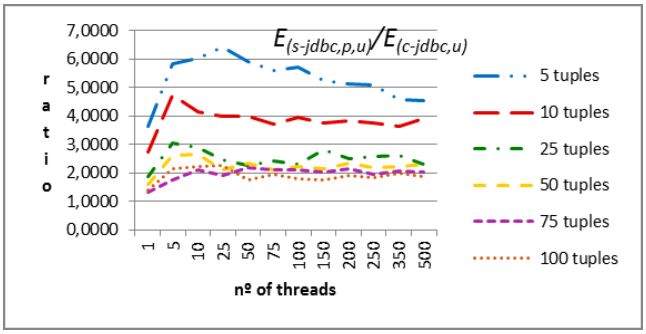

Figure 6. Update scenario: ratio between S-JDBC and C-JDBC.

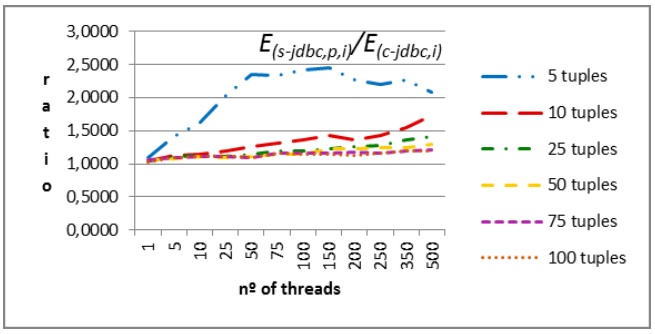

Figure 7. Insert scenario: ratio between S-JDBC and C-JDBC.

transferring blocks from dataset servers to LMS. The maximum gain (above 6) is reached for 5 tuples and 25 threads. The chart for the Insert scenario shows that the gain is always greater than 1 and that it increases when the number of tuples decreases and also when the number of threads increases. This last behavior is curious. It derives from the fact that server datasets are empty and there are no transferences of blocks from dataset servers to LMS. This way, as the number of server datasets increases, performance of S-JDBC decreases more significantly than performance of C-JDBC.

VII. CONCLUSION

Performance has become increasingly a key concern in intensive database applications. Recently is has reached a major importance with the advent of Big Data and IoT. Among many issues that can influence the overall performance, the middleware that connects business logics to RDBMS and NoSQL servers is certainly a key component, in our case, CLI. CLI are composed by two main types of components: client-side components and server side components. These components were designed to perform in environments where concurrency is not a major concern. Basically, both types of components are not prepared to work on client-side environments where several threads need to access to the same memory structures, especially LMS. To overcome this drawback, we propose a new design for CLI where LMS are natively thread-safe, in opposite to the work done with CTSA, The collected results show that the improvement in performance is noticeable in the Select scenario even when compared with the results collected with CTSA. The ratios have been improved in a mean of 6% and the number of ratios < 1.0 felt from 40 to 25. In the remaining scenarios, Update and Insert, the collected results are very similar. This is due to the fact that Update and Insert protocols are much heavier than the Select, leading to a much lower percentage impact in the overall performance.

As a future work, we are already working on an extension of the C-JDBC which will provide an additional functionality. Basically, besides the single thread-safe LMS already implemented, it will also provide replicas of the same LMS. This way, each thread will own its own LMS but the server will only need a single server cursor. The replication process can bring many advantages in many situations (now threads can interact with LMS without any locking mechanism) but the update and insert processes need additional processing to keep data consistency in all LMS. Anyway, the preliminary results are very encouraging.

ACKNOWLEDGEMENTS

This work is funded by National Funds through FCT - Fundação para a Ciência e a Tecnologia under the project UID/EEA/50008/2013.

REFERENCES

[1] ISO, "ISO/IEC 9075-3:2003," 2003. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134.
[2] Microsoft, "Microsoft Open Database Connectivity," 1992. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx.
[3] M. Parsian, *JDBC Recipes: A Problem-Solution Approach*. NY, USA: Apress, 2005.
[4] B. Christian and K. Gavin, *Hibernate in Action*. Manning Publications Co., 2004.
[5] D. Yang, *Java Persistence with JPA*. Outskirts Press, 2010.
[6] D. Kulkarni, L. Bolognese, M. Warren, A. Hejlsberg, and K. George, "LINQ to SQL: .NET Language-Integrated Query for Relational Data." Microsoft.
[7] Oracle, "ResultSet," 2012. [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/sql/ResultSet.html.
[8] Microsoft, "RecordSet (ODBC)," *Microsoft*. [Online]. Available: http://msdn.microsoft.com/en-us/library/5sbfs6f1.aspx. [Accessed: 16-Nov-2016].
[9] W. Cook and A. Ibrahim, "Integrating programming languages and databases: what is the problem?," 2005. [Online]. Available: http://www.odbms.org/experts.aspx#article10.
[10] Ó. M. Pereira, R. Aguiar, and M. Santos, "A Concurrent Tuple Set Architecture for Call Level Interfaces," in *Computer and Information Science*, vol. 493, R. Lee, Ed. Springer International Publishing, 2013, pp. 143–158.
[11] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "CTSA: Concurrent Tuple Set Architecture Extending Concurrency to Call Level Interfaces," *IJSI - Int. J. Softw. Innov.*, vol. 1, no. 3, pp. 12–33, 2013.
[12] C. J. Fidge, "Real-Time Schedulability Tests for Preemptive Multitasking," *Real-Time Syst.*, vol. 14, no. 1, pp. 61–93, 1998.
[13] Spring, "Spring." [Online]. Available: http://www.springsource.org/.
[14] Oracle, "Oracle TopLink," 2011. [Online]. Available: http://www.oracle.com/technetwork/middleware/toplink/overview/index.html.
[15] 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*, 2006, p. 706.
[16] A. M. Russell and H. K. Ingolf, "SQL DOM: compile time checking of dynamic SQL statements," in *27th Int. Conf. on Software Engineering*, 2005, pp. 88–96.
[17] R. C. William and R. Siddhartha, "Safe query objects: statically typed objects as remotely executable queries," in *27th Int. Conf. on Software Engineering*, 2005, pp. 97–106.
[18] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "Assessment of a Enhanced ResultSet Component for Accessing Relational Databases," in *ICSTE-Int. Conf. on Software Technology and Engineering*, 2010, p. V1:194-201.
[19] J. L. Gregor Kiczales Anurag Mendhekar, Chris Maeda, Cristina Lopes Videira, Jean-Marc Loingtier, Joh Irwin, "Aspect-Oriented Programming," in *ECOOP*, 1997, pp. 220–242.
[20] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich,CT,USA: Manning Publications, 2003.
[21] J. Fabry and T. D'Hondt, "KALA: Kernel Aspect Language for Advanced Transactions," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006, pp. 1615–1620.
[22] T. Dinkelaker, "AO4SQL: Towards an Aspect-Oriented Extension for SQL," in *8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'11)*, 2011, pp. 1–5.
[23] D. Gomes, Ó. M. Pereira, and W. Santos, "JDBC (Java DB connectivity) concorrente," University of Aveiro, ria - institutional repository, http://hdl.handle.net/10773/7359, 2011.