Database Engines on Multicores Scale: A Practical Approach

João Soares
NOVA-LINCS/DI-FCT
Universidade Nova de Lisboa

Nuno Preguiça
NOVA-LINCS/DI-FCT
Universidade Nova de Lisboa

ABSTRACT

Multicore processors are available for over a decade, being the norm for current computer systems, but general purpose database management systems (DBMS) still cannot fully explore the computational resources of these platforms. We focus on In-Memory DBMS since these are becoming widely adopted, due to the increasing amount of memory installed in today's systems, and are expected to scale on multicore machines, by not incurring in I/O bottlenecks. This paper presents a practical study on In-Memory DBMS and shows that contention imposed by concurrency control mechanisms, such as locking, are limiting factors for both performance and scalability of these systems on multicores. Additionally, we discuss a simple database engine modification that allows an almost 10 fold performance improvement, over the original engine, also allowing databases to scale on multicores.

1. INTRODUCTION

The current processor evolution, to multicore processors, has forced software designers and developers to parallelize code in order to improve application performance on multicore platforms. While multicore processors are now available for over a decade, being the norm for current computer systems, these still pose challenges to the design of database management systems (DBMS) [12, 17, 23, 5, 9]. Existing studies show that current database engines can spend more than 30% of time in synchronization-related operations (e.g. locking and latching), even when only a single client thread is running [13]. Additionally, running two concurrent database operations in parallel can be slower than running them in sequence [25], due to workload interference. This is a limiting factor for the scalability of DBMS in current multicore platforms [18].

Several research solutions have been proposed to improve resource usage of multicore machines. Some of these solutions aim at using multiple threads to execute query plans in parallel, or using new algorithms to parallelize single steps of the plan, or effectively parallelizing multiple steps [24, 25, 9, 8, 5]. Other solutions try to reuse part of the work done during the execution of multiple queries [10], or using additional threads to prefetch data that can be needed in the future [17]. Although some of these solutions start to appear in niche markets, general purpose DBMSs have been slower to adopt them, since implementing such solutions requires significant design modifications.

A widely used approach for improving DBMS performance is weakening/relaxing the database isolation level. This aims at improving throughput by preventing read only transactions from aborting/delaying update ones, allowing more transactions to execute concurrently. Although theoretically sound, this relaxation does not translate into considerable benefits, as presented latter.

In this paper we present a study on the performance and scalability of In-Memory databases (IMDB). IMDBs provide high performance by not incurring in disk I/O overhead, when compared to disk backed DBMS. Also, the increasing amount of main memory used in current computer systems, in conjunction with the ease of embedding IMDBs in applications have made these systems increasingly popular, being used by a large number of applications and high-performance transaction processing systems, such as Sprint [6] and H-Store [15].

Focusing on two open source IMDBs: HSQLDB [2] and H2 [1], and using TPC-C, a well established OLTP benchmark, we show how both engines perform on a 16 core Sun Fire X4600 with 32 GBytes of RAM. Additionally, we discuss and investigate how the different sub-components of the database engine affect the performance of these systems. We show that contention on the underlying data structures is the main cause for their lack of scalability on single machines multicore systems. Additionally we propose a simple modification that allows both database engines to scale on multicore machines, achieving an almost 10 fold performance improvement over their unmodified siblings.

The major contributions of this paper are: *i*) a practical performance study of in-memory database engines on single multicore machines; *ii*) the identification and measurement of the performance overhead imposed by the sub-components of these engines; and *iii*) the proposal of a simple modification that allows both in-memory database engines to scale on multicores.

The remainder of this paper is organized as follows: Section 2 provides an overview of DBMS sub-components, their interactions and performance impact; Sections 3 and 4 present a practical performance study, and propose some modifications for improving their performance on a single machine multicores; Section 5 presents the evaluation results of our proposal; Section 6 discusses the most relevant related work; and Section 7 concludes this paper.

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695793

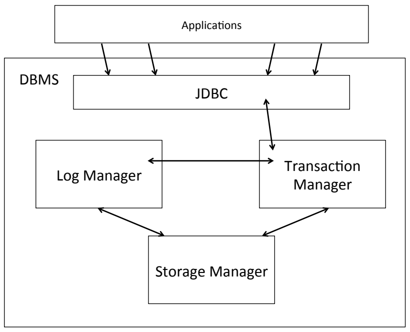


Figure 1: IMBMDS architecture.

2. IN-MEMORY DBMS OVERVIEW

In this section we provide an overview on the components of a DBMS, focusing on how these are implemented on the studied systems and discussing their interactions during transaction execution and performance implications.

Traditional relational DBMS design feature disk-based storage system (indexing and heap files), a log-based transaction manager, and a concurrency control mechanism. Most current in-memory DBMS have evolved from this traditional design, only abandoning the disk-based storage systems to in-memory ones [21].

Applications interact with DBMSs by a well defined API, such as ODBC/JDBC, as presented in Figure 1. Operations, performed by each client, are grouped into *transactions*. Transactions are a series of query and/or update operations delimited by a commit or rollback operation. DBMS allow concurrent transactions to execute in isolation from one another, in accordance to the specified isolation level [4]. The isolation level, defined by the application, is provided by a *Transaction Manager* in conjunction with a *Log Manager*, while data is maintained by a *Storage Manager*.

Transaction Manager.

The transaction manager provides transaction isolation. Generally, it uses a locking scheme to allow concurrent execution of non conflicting transactions, and preventing conflicting ones from executing. It offers different isolation level to increase concurrency.

Log Manager.

The log manager is used to preserve and guarantee a consistent database state, by logging operations. It also allows databases to recover from possible faults by redoing unfinished logged operations.

Storage Manager.

The storage manager maintains the data structures that preserve the data of the database. Tables are commonly mapped by one or more index structure, used to maintain the respective table rows. B-Trees and Hash tables are commonly used to implement these indexes.

Although additional sub-components impact the overall performance of DBMSs, such as parsers and query optimizers, they have been extensively covered by their respective research community [11, 8, 16] and as such several proposals exist to improve their performance on multicore systems. Additionally, since these cannot be simply turned off or swapped, we have no means to study their impact on the overall system performance, thus will not be addressed in this discussion. Next we will detail how HSQLDB and H2 engines behave during client interaction.

```
 1  var global:
 2  Transaction_Manager tx_mngr
 3  Storage_Manager storage_mngr
 4  Log_Manager log_mngr
 5
 6  var per client:
 7  Session session
 8  Result result
 9  Command command
10
11  function executeCommon ( statement )
12    if( NOT valid_connection ( session ) )
13      throw DBError
14    if( NOT validate_syntax ( statement ) )
15      throw SyntaxError
16    valid_statement = parse_and_compile ( statement )
17    result = create_result_set ( valid_statement )
18    command = optimize ( valid_statement )
19    tx_manager.aquire_table_locks ( session , command )
20
21  function executeQuery ( statement )
22    executeCommon ( statement )
23    storage_mngr.read_data ( command, result )
24    return result
25
26  function executeUpdate ( statement )
27    executeCommon ( statement )
28    log_previous_data ( session , command )
29    storage_mngr.delete_row ( session , command )
30    storage_mngr.insert_row ( session , command )
31    storage_mngr.verify_integrity ( session )
32    fire_table_triggers ( )
33    return result;
34
35  function commitCommon ( )
36    tx_manager.unlock_tables ( session )
37    tx_manager.awake_awaiting_txs (  )
38
39  function commitQuery ( )
40    commitCommon ( )
41
42  function commitUpdate ( )
43    log_mngr.log_actions ( session )
44    log_mngr.log_commit ( session )
45    commitCommon ( )
```

Figure 2: Statement execution.

2.1 H2 and HSQLDB behavior

Both engines interact with client applications through a JDBC interface. Whenever a client establishes a new connection to the database engine a new *Session* is created. Sessions are used by the database engine for providing atomicity and isolation to the statements performed by different clients. A simplified algorithm of statement execution is presented in Figure 2. We omit error and conflict verification due to simplicity. During this discussion, we call *transaction* to a series of statements, queries and/or updates, executed in a session and delimited by a commit or rollback command.

Whenever a statement is executed by a client, both database engines start by validating the state of the connection, i.e., if it has not been previously closed by the client, and the syntax of the statement. If no error occurs, then a new result object for that statement is created. This object is used to maintain the statement's result and its respective metadata (e.g. the information on the tables and columns being read, the number of lines of the result and their respective data). After this, an optimization stage selects an execution plan suitable for the statement's execution, as presented in lines

11-18 of Figure 2.

Sessions proceed by interacting with the *transaction manager* for executing the statement in isolation (line 19 of Figure 2). Both engines provide different isolation levels, offering lock based or multi-version schemes. For simplicity reasons we will focus on lock-based ones.

For lock schemes, both engines implement standard two-phase locking mechanism, using shared and exclusive locks at a table level. Sessions are only allowed to execute each operation after acquiring the necessary table locks. If a session fails to acquire a table lock, due to a conflicting concurrent session, then it will wait until the necessary locks are released. Each session maintains the set of table locks acquired during its execution, while the transaction manager maintains a global map of sessions and their associated table locks to detect possible conflicts. All these steps are common to both queries and update statements.

After acquiring table locks, sessions proceed by interacting with the *storage manager*. For query statements, the corresponding data is copied to the session's result set and is then returned to the client (lines 23 and 24 of Figure 2). For update statements, the corresponding data is first read and logged for state recovery purposes, i.e., in case the transaction aborts due to a conflict or a rollback is issued by the client (line 28 of Figure 2), only then will the *storage manager* update the necessary table indexes (lines 29-32 of Figure 2). In both engines sessions maintain *logged* data until a commit or rollback operation. Also, both engines implement index using AVL-trees, and updates are executed as an index delete followed by an insert operation.

All commit operations release acquired locks and awaken existing waiting concurrent sessions. For update transactions sessions first interact with the *log manager*, logging all performed actions (lines 43 and 44 of Figure 2). It logs data updated during the transaction execution, old and new values, followed by the commit operation it self. Rollback operations undo the necessary changes before releasing locks.

Next we present a performance study of two different IMDB engines, H2 [1] and HSQLDB [2], and discuss the implications of the previously described components on the performance of those DBMS on multicores, and propose a simple, but effective improvement.

3. PERFORMANCE OF IMDB

Compared to disk based DBMS, IMDBs incur in no overhead or contention in accessing I/O. Thus, one would expect these systems to scale with the number of cores. To verify the veracity of this assumption, we measured the throughput obtained when running the TPC-C benchmark, on HSQLDB and H2 IMDBs, with a read-only workload. This experiment ran on a 16 core Sun Fire X4600 with 32 GBytes of RAM. The results, presented in Figure 3, show that these engines do not scale, even when transactions do not conflict with each other.

For understanding if the lack of scalability was due to the lack of computational resources, we ran an increasing number of pairs client/DB engine concurrently, i.e., one DB engine per client, on the same machine. The results of this experiment, presented in Figure 3 as *HSQLDB (aggr)* and *H2 (aggr)*, show an increase in the aggregate throughput near linear scalability up to 8 clients for both HSQLDB and H2. These results show that the general lack of performance

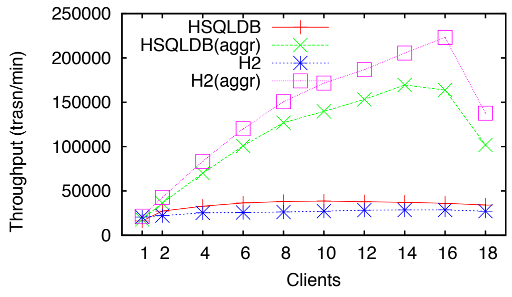


Figure 3: Scalability of TPC-C read-only workload.

and scalability is related with the design of current IMDBs. We further investigated the reasons for this in this section.

Transactional semantics.

We started by investigating how different transactional semantics, i.e., isolation levels, impact the performance of these systems. The main purpose of weaker isolation levels is to achieve higher concurrency among transactions - e.g. the relaxation from serializability to snapshot isolation should increase throughput of read-only transactions since these can execute on a different database snapshot from update transactions, thus are not aborted nor delayed by concurrent updates.

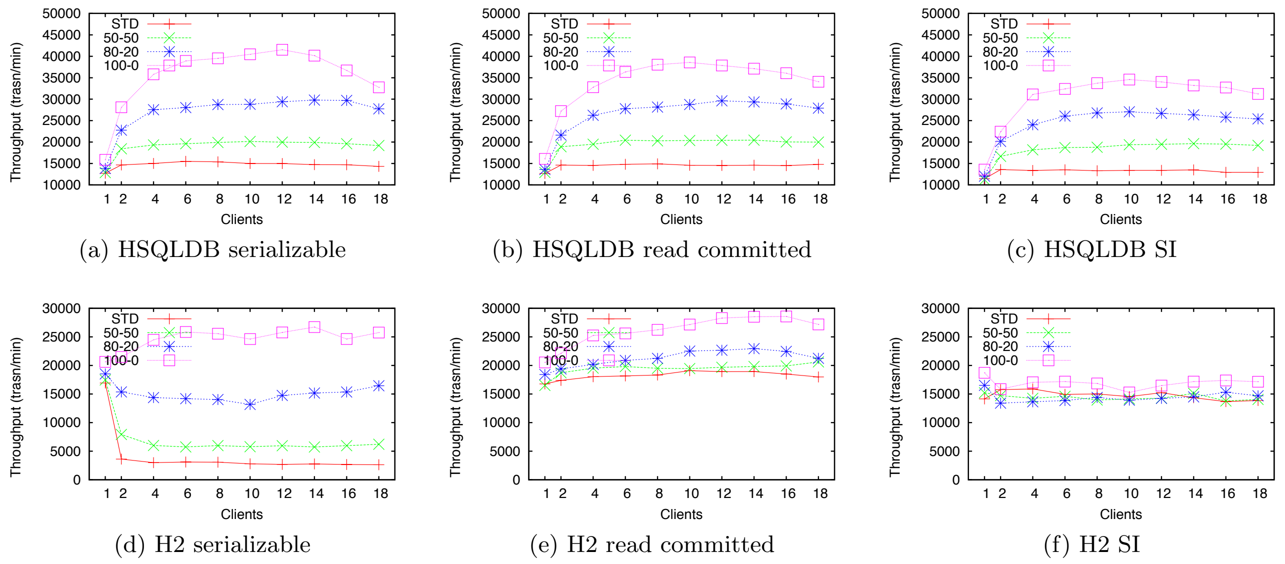
We measured the throughput of different workload mixes of TPC-C transactions on both HSQLDB and H2 under different isolation levels: *i*) *serializable*, relying on two-phase locking; *ii*) *read committed*, relying on two-phase locking with early release of read locks; and *iii*) *snapshot isolation*, relying on a multi-version concurrency control solution. The workloads varied from update intensive ones: *standard* with 92% updates and *50-50* with 50% updates; to read intensive ones: *80-20* with 20% updates and *100-0* with 0%updates.

The results, presented in figure 4, show that, while both engines show performance variations between the different isolation semantics (more considerable for H2 and very small for HSQL), they do not scale on multicores, even for read-only workloads. As expected, the higher the ratio of updates the lower the performance. The results for *snapshot isolation* show no benefit over *serializability* in read intensive workloads, while for update intensive workloads there is a performance improvement on H2. Overall, these results show that although different transactional semantics may lead to different result, there is no significative impact on scalability.

Logging impact.

While both IMDBs log transaction updates for recovery, results do not suggest that this is the main bottleneck for scalability. If this were the case, *read-only* workloads would scale much better, as they log no information. Unlike traditional databases, these two IMDBs implement transaction concurrency control by relying on table-level locks. This is a pragmatic approach that allows to avoid the complexity and overhead of fine-grain and semantic locks. Again, if this was the main reason for the lack of scalability, *read-only* transactions would have to scale better, especially for the read-committed and SI isolation levels. Next, we investigate the impact of the *storage manager* component.

Storage contention.



(a) HSQLDB serializable (b) HSQLDB read committed (c) HSQLDB SI

(d) H2 serializable (e) H2 read committed (f) H2 SI

Figure 4: Database Performance for TPC-C.

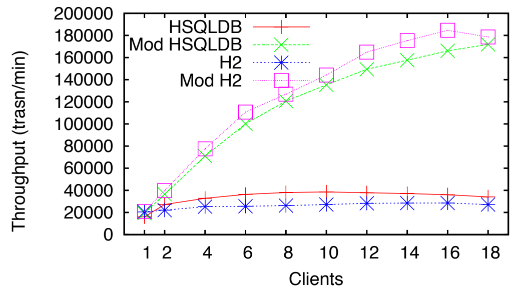


Figure 5: Modified engines for TPC-C read-only workload.

From the previous result, we believe IMDB performance is compromised due to contention, created by concurrency control mechanisms, in the *storage manager* component. To understand this impact, we removed all locks from index data structures on both engines. Figure 5 presents the results of the original and modified version of HSQLDB and H2, presented as *Mod HSQLDB* and *Mod H2* respectively, with a *read-only* workload under the *read committed* isolation level.

The results show the significative benefits of removing all index locks. With this modification we were able to obtain an almost 10 fold performance increase compared to the unmodified engines. Moreover, both engines were able to scale almost up to the number of cores. Thus, one can conclude that scalability of DBMS on multicores is greatly restricted due to the concurrency control implemented in the data structures of the storage manager. Next we will discuss how to leverage this insight for improving the performance of IMDBs on multicore machines.

4. UNLOCKING SCALABILITY

As put in evidence by the results presented in the previous section, performance wise there is no real advantage for in-memory databases to offer relaxed isolation levels. The increase in concurrency, attained from relaxing isolation levels, increases contention on the data structures, and does not translate into performance benefits. This has been confirmed by removing concurrency control (CC) mechanisms used at the storage level, which allowed both modified engines to scale with the number of cores.

Multi-version support.

Data structures use CC mechanisms to guarantee data consistency when accessed concurrently. Removing such mechanisms may result in state corruption due to concurrent write/write operations, or may expose data inconsistencies under concurrent read/write operations. While database transaction managers prevent conflicting update transactions from concurrently executing, thus preventing concurrent updates on the data structures, multi-version concurrency control, such as SI, allow read transactions to execute concurrently with update ones without any concurrency control. These phenomena also occur when using the read uncommitted isolation level. Since, under these isolation levels, read and write operations may execute concurrently at the storage level, data structures must use CC mechanisms to provide data integrity.

Locking and 2PL benefits.

Isolation levels based on locking schemes acquire read or write locks, i.e., shared or exclusive locks, before reading or updating a table, respectively. This prevents read transactions from concurrently executing with conflicting update ones. Compared to the previous protocols, locking not only prevents concurrent execution of write/write operations at the storage level, but also prevents read operations on the data structures to be executed concurrently with updates. Since locking schemes only allow read operations to execute concurrently with other read operations, at the storage level, it prevents both scenarios that may cause data structure corruption (assuming that data structures are not modified on reads). This way, locking schemes provide healthy data structure states even when using data structures without CC mechanisms. As a particular case of a locking scheme, 2PL divides locking into 2 separate phases: lock acquiring and lock releasing, with the restriction that no new lock is acquired after the release of a lock. This allows 2PL to offer the same benefits as general locking, adding higher isolation levels, such as serializability. Thus, 2PL prevents concurrent read/write and write/write conflicts from executing at the storage level, while offering serializable transaction isolation levels.

4.1 Proposed Solution

As previously stated, the use of 2PL schemes prevents conflicting operations, i.e., read/write and write/write operations, from concurrently executing at the storage level, while offering serializable isolation level. This allows removing all concurrency control from the data structures.

Thus, our proposal is removing support for multi-version isolation schemes and row level locking, in favor of 2PL schemes at the table level, also removing support for the read uncommitted isolation level, and combine these restrictions with the removal of all CC mechanisms used at the storage level.

Assuming that the DBMS implement table level locking, which most IMDB implementations do, and write locks are kept until the end of each transaction, then there is no need for concurrency control at the storage level. Using table level locking enforces that read operations only execute concurrently with each other, while maintaining write locks until the end of the transaction enforces that update transactions execute in mutual exclusion and prevent inconsistent states from being exposed, thus providing data consistency.

Although this is a simple modification the achieved performance improvements are considerable, allowing IMDB to scale on multicores, as put in evidence next.

5. EVALUATION

In this section we evaluate the proposed modifications to the IMDB engines, and compare them to their unmodified siblings, by measuring throughput gains and scalability, compared to the unmodified engines.

Setup.

All experiments were performed on a Sun Fire X4600 M2 x86-64 server machine, with eight dual-core AMD Opteron Model 8220 processors and 32GByte of RAM, running Debian 5 (Lenny) operating system, H2 database engine version 1.3.169 and HSQL engine version 2.3.1, and OpenJDK version 1.6. All IMDB used a read committed isolation level and table level locking.

For this evaluation we used the TPC-C benchmark, varying the number of clients, form 1 to 18, and workloads: standard (8% reads and 92% writes); 50-50 (50% reads and 50% writes); 80-20 (80% reads and 20%writes); and 100-0 (100% reads). The benchmark ran for 2 minutes, and a 4 gigabyte database was used. The presented results are the average of 5 runs, performed on a fresh database, disregarding the best and the worst results, and were obtained from the unmodified versions of *HSQLDB* and *H2* engines, and the respective modified versions, named *Mod HSQLDB* and *Mod H2* respectively.

Standard Workload.

Under update intensive workloads the modified engines offer some performance benefits over their unmodified siblings, as presented in Figures 6 and 7. Although our modified solutions are able to offer a 15% and 25% performance improvement, compared to the unmodified HSQLDB and H2 engines respectively, these are still unable to scale. This is an expected situation, since all TPC-C update transactions interfere, thus restricting concurrency.

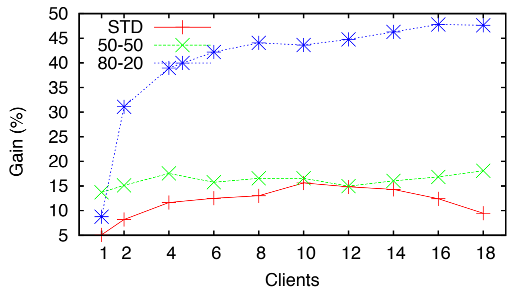


Figure 6: Mod HSQLDB TPC-C performance gain.

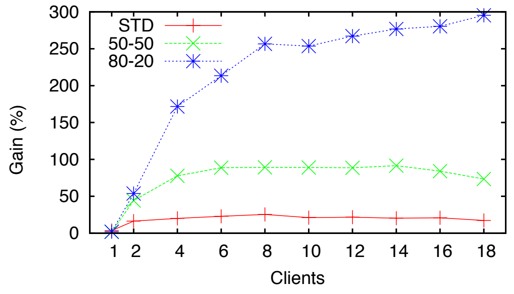


Figure 7: Mod H2 TPC-C performance gain.

50-50 Workload.

In moderate update workloads, our modified versions of both HSQL and H2 are able to achieve higher throughput than the standalone versions. As presented in Figures 6 and 7, these offer up to 20% and 90% performance improvements over the unmodified HSQL and H2 database engines respectively. Once again, the conflicting nature of the TPC-C workload has a considerable impact on scalability on both modified engines.

80-20 and 100-0 Workloads.

The nature of read intensive workloads is favorable to the proposed modifications, with both modified versions achieving higher throughput than their unmodified siblings. The modifications offer an increase in performance of approximately 50% and 300%, over the unmodified HSQL and H2 engines respectively, for the 80-20 workload (Figures 6 and 7). And an approximately 400% and 550% performance increase over the unmodified HSQL and H2 engines respectively, for the 100% read workload, as presented in Figure 8.

Additionally, read intensive workloads allow both modified database engines to scale on multicore machines. Above all, for read-only workloads, we can see that the obtained results for both modified engines are fairly close to their aggregate maximum achievable by the same machine, presented in Figures 8 as *HSQLDB(aggr)* and *H2(aggr)*, respectively. These workloads also allow the modified versions to scale near linearly with the number of cores, thus taking advantage of the computational resources offered by current multicore systems.

6. RELATED WORK

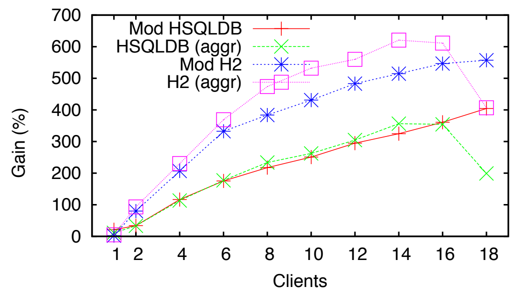


Figure 8: TPC-C performance gain for 100-0 Workload.

While several works have addressed database scalability on multicores, many approaches follow the path of distributing the database engine [7, 3, 18, 20, 19]. These works are complementary to ours, since these rely on replication techniques for reducing contention and improving database performance and scalability.

In Shore [13] and Shore MT [14], the authors also focus on the sub-component of database engine and their impact on performance. While isolating database components, the authors focus on traditional disk based database engines. Thus, since these systems differ from in-memory databases most of the proposed solutions are inadequate for such systems.

In Silo [22], the authors propose the use of concurrent data structures in conjunction with optimistic concurrency control to provide scalability. Contrarily to our solution, Silo requires applications to be re-written to take advantage of their system, since clients issue single-shot requests to the database instead of variable length transactions.

To our knowledge this is the first practical performance study for in-memory database systems that focus on the different subcomponents of the systems and their performance implications.

7. FINAL REMARKS

In this paper we presented a study on the performance and scalability of two in-memory DBMS on multicore machines, HSQLDB [2] and H2 [1].

While IMDBs are expected to provide high performance, since these do not incur in disk I/O overhead when compared to traditional databases, our tests showed that both implementations are unable to scale on multicores. We also investigated the performance impact of the different subcomponents of these systems, and concluded that contention on the underlying data structures are the main bottleneck for their lack of scalability.

Additionally we proposed a simple and effective modification that allowed both engines to scale on multicore systems. Our evaluations showed that the modified engines out perform their siblings up to 300% to 500% when running the TPC-C benchmark with read-intensive workloads.

Acknowledgments

This research was partially supported by FCT/MCT projects PEst-OE/EEI/UI0527/2014 and PTDC/EEI-SCR/1837/2012 and EU FP7 project SyncFree (grant no 609551). João Soares was partially supported by FCT/MCTES research grant # SFRH/BD/62306/2009.

8. REFERENCES

[1] H2 database engine, http://www.h2database.com, (2012).

[2] HyperSQLDB, http://hsqldb.org, (2012).

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania: The multikernel: a new os architecture for scalable multicore systems. In Proc. SOSP'09 (2009).

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil: A critique of ansi sql isolation levels. In Proc. SIGMOD'95 (1995).

[5] S. Blanas, Y. Li, and J. M. Patel: Design and evaluation of main memory hash join algorithms for multi-core cpus. In Proc. SIGMOD'11, (2011).

[6] L. Camargos, F. Pedone, and M. Wieloch: Sprint: a middleware for high-performance transaction processing. In Proc. EuroSys'07 (2007).

[7] E. Cecchet, G. Candea, and A. Ailamaki: Middleware-based database replication: the gaps between theory and practice. In Proc. SIGMOD'08 (2008).

[8] C. Chekuri, W. Hasan, and R. Motwani: Scheduling problems in parallel query optimization. In Proc. PODS'95 (1995).

[9] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye: Automatic contention detection and amelioration for data-intensive operations. In Proc. SIGMOD'10 (2010).

[10] G. Giannikis, G. Alonso, and D. Kossmann: Shareddb: killing one thousand queries with one stone. In Proc. VLDB'12 (2012).

[11] W.-S. Han and J. Lee: Dependency-aware reordering for parallelizing query optimization in multi-core cpus: In Proc. SIGMOD'09 (2009).

[12] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi: Database servers on chip multiprocessors: Limitations and opportunities. In CIDR'07 (2007).

[13] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker: OLTP through the looking glass, and what we found there. In Proc. SIGMOD'08 (2008).

[14] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki: A new look at the roles of spinning and blocking. In Proc. DaMoN'09 (2009).

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi: H-store: a high-performance, distributed main memory transaction processing system. Proc. VLDB'08 (2008).

[16] K. Krikellas, M. Cintra, and S. Viglas: Multithreaded query execution on multicore processors. Technical report, The University of Edinburgh School of Informatics, (2009).

[17] K. Papadopoulos, K. Stavrou, and P. Trancoso: HelperCoreDB: Exploiting multicore technology for databases. In Proc. PACT'07 (2007).

[18] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso: Database engines on multicores, why parallelize when you can distribute? In Proc. EuroSys'11 (2011).

[19] J. Soares, J. Lourenço, and N. Preguiça: MacroDB: Scaling database engines on multicores. In Proc. Euro-Par'13 (2013).

[20] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang: A case for scaling applications to many-core with os clustering. In Proc. EuroSys'11 (2011).

[21] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland: The end of an architectural era: (it's time for a complete rewrite). In Proc. VLDB'07 (2007).

[22] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden: Speedy transactions in multicore in-memory databases. In Proc. SOSP'13 (2013).

[23] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann: Predictable performance for unpredictable workloads. Proc. VLDB'09 (2009).

[24] Y. Ye, K. A. Ross, and N. Vesdapunt: Scalable aggregation on multicore processors. In Proc. DaMoN'11 (2011).

[25] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah: Improving database performance on simultaneous multithreading processors. In Proc. VLDB'05 (2005).