MacroDB: Scaling Database Engines on Multicores*

João Soares, João Lourenço and Nuno Preguiça
CITI/DI-FCT-Univ. Nova de Lisboa

Abstract. Multicore processors are available for over a decade, but general purpose database management systems (DBMS) still cannot fully explore the computational resources of these platforms. This paper explores a simple and easy to deploy approach for improving DBMS performance in multicore platforms, by maintaining multiple database engines running in parallel, rather than a single instance, thus circumventing the increase in contention due to load interactions. Unlike previous works, we focus on in-memory DBMS, exploring different design solutions that combine distributed systems and concurrent programming techniques. We show that we are able to improve performance over standalone solutions, without modifying either database or application code, by up to 3 times while minimizing response times.

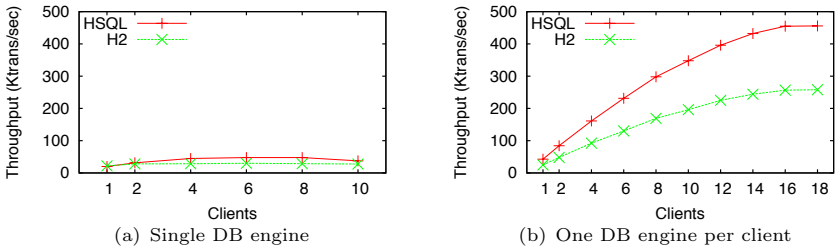
1 Introduction

Multicore processors are now available for over a decade, and still pose challenges to the design of database management systems (DBMS) [10,17,21,3,7]. Existing studies show that current DBMS 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 [11]. Additionally, running two concurrent database operations in parallel can be slower than running them in sequence [26], due to workload interference. This is a limiting factor for the scalability of DBMS in current multicore platforms [19].

Several research solutions have been proposed to improve the use of resources offered by multicore machines. Some 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 [25,26,7,6,3]. Other solutions try to reuse part of the work done during the execution of multiple queries [9], 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.

This paper addresses the problem of improving the scalability of DBMS on multicore machines, focusing on in-memory databases (IMDB). IMDBs provide high performance because they do not incur in disk I/O overhead. The high performance and ease of embedding them 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 [4] and H-Store [14].

* This work was partially support by FCT/MCT projects PEst-OE/EEI/UI0527/2011 and PTDC/EIA-EIA/108963/2008. João Soares was partially supported by FCT/MCTES research grant # SFRH/ BD/ 62306/ 2009.



(a) Single DB engine (b) One DB engine per client

Fig. 1: Scalability of read-only workload

Scalability issues of in-memory DBMS Compared to disk based databases, IMDBs incur in no overhead or contention in accessing I/O. Thus, we would expect these systems to scale with the number of cores. To verify if this was true, we have run the TPC-C benchmark with a 100% read-only workload in popular HSQL and H2 IMDBs on a 16 core Sun Fire X4600 with 32 GBytes of RAM. The results of Figure 1(a) 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 lack of resources, we concurrently ran an increasing number of pairs client/DB engine in the same machine. Figure 1(b) presents the results of such experiment, showing an increasing aggregate throughput. These results make it clear that the problem lies in the design of current IMDBs.

1.1 Proposed approach

In this paper, we explore a simple and easy to deploy mechanism for scaling IMDB, by relying on database replication and building on knowledge from distributed and replicated database systems. By treating multicore machines as an extremely low latency cluster, extended with shared memory, we deploy a middleware system, MacroDB, as a collection of coordinated IMDB replicas, for providing scalable database performance on multicores systems.

MacroDB uses a master/slave replication approach, where update transactions execute on the master replica, which holds the primary copy of the database. The slaves maintain independent secondary copies of the database, receiving read-only transactions from clients, while updates are asynchronously propagated to them upon commit on the primary. This approach minimizes contention since: *i*) Read-only transactions are fully executed on slave replicas, reducing the number of transactions each replica processes, thus distributing the load among the available replicas, and *ii*) update transactions are applied in slave replicas as sequential batches of updates, leading to no contention among them.

MacroDB provides a scalable data management solution, that does not require any changes to neither database engines or the applications. Our experiments show that MacroDB is able to provide performance benefits ranging from 40% to 180% over standalone database engines in a diverse range of benchmark workloads, such as TPC-C and TPC-W, while for write-dominated workloads, such as TPC-C, MacroDB suffers from only a 5% to 14% overhead over standalone solutions. Additionally, the memory used by the database replicas is not

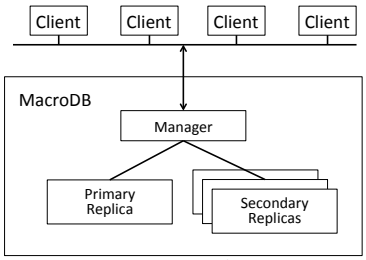


Fig. 2: MacroDB Architecture

directly proportional to the number of replicas, as replicas share immutable Java objects, thus making MacroDB practical even with large numbers of replicas.

This paper is organized as follows, section 2 describes our system and some of the prototype considerations. Section 3 presents the evaluation results. Section 4 presents some related work, and section 5 concludes this paper.

2 MacroDB

MacroDB is a middleware infrastructure for scaling IMDBs on multicore machines. It replicates the database on several engines, all running on the same machine, while offering a *single-copy serializable view* of the database to clients [2].

It works independently of the underlying database engine, acting as a transparent layer between applications and the database. Statements received from the application are passed, without modifications, to the underlying engines. This makes MacroDB easy to deploy, since it does not require any modification to existing applications or database engines. This section details the architecture and algorithms used in the system.

2.1 Architecture

The MacroDB architecture, depicted in Figure 2, is composed by two main components: the *manager*, responsible for coordinating transaction execution in the database replicas; and the *database replicas*, i.e., the engines responsible for maintaining copies of the database. Clients remain oblivious of the replicated nature of MacroDB since it offers them a standard JDBC interface, and provides them with a *single-copy serializable view* of the database [2].

Clients do not communicate directly with the database engines, instead they communicate with the MacroDB *manager*, a JDBC compliant front-end which coordinates client queries and the underlying replicas. The *manager* receives statements from clients and forwards them, without modification, to the appropriate replica, guaranteeing their ordered execution, and replying to clients the respective results. Its main function is to: *i*) route client request to the appropriated replica, *ii*) manage operation execution to guarantee that the system provides a single consistent serializable view of the replicated database to the applications, and *iii*) to detect and recover possible replica failure. In the next section, we detail transaction execution.

2.2 Transaction Execution

MacroDB uses a master-slave replication scheme [12,23]. The master maintains the primary copy of the database, while the slaves maintain secondary replicas. Update transactions, received from clients by the *manager* are executed

concurrently on the primary copy, being asynchronously propagated to the secondary replicas upon commit. This means that secondary replicas might not be completely up to date at a given moment. Read-only transactions execute concurrently on the secondaries.

Each secondary replica maintains: *i*) an associated *version*, that maintains the number of update transactions committed in the replica. This version is kept in shared memory, as an atomic *commit counter*, and can be accessed by any thread running in MacroDB; *ii*) a list for pending update batches, and *iii*) a thread responsible for executing these batches.

We will now detail the steps for executing update and read-only transactions. We assume the setReadOnly method of the JDBC interface is used for defining read-only and update transactions. The code for transaction execution is presented in Figure 3. For simplicity, we omit the code for error handling and present an explicit *begin* transaction operation - in the prototype, the code for begin transaction is executed when the first query or update operation is called after a commit or rollback.

Update Transactions For each client connection, when an update transaction begins, a newly associated batch is created. All statements executed in this context are executed by the master replica, using the context of the caller thread, and their results returned to the client. Additionally, if the operation was an update, it is added to the batch for that transaction.

If the client decides to commit the transaction, the commit is executed in the master replica, using the caller thread. If the commit succeeds, the version number associated with the master replica is incremented and the update batch, stamped with that version number, is inserted into the lists of pending batches for the secondary replicas. For correct transaction ordering, MacroDB needs to guarantee that no new transaction starts and commits between the commit of a transaction and its ordering, thus this it the only operation that requires coordination with other threads. For guaranteeing that the commit does not block, we require the underlying database engines to use two-phase locking (instead of commit time certification strategies), which is the case in most in-memory database systems. If the client decides to rollback or if the underlying engine is unable to commit the transaction, a rollback is executed in the master replica and the associated batch is discarded.

The thread associated with each secondary replica waits for the next update batch to be inserted into the associated list, and executes it. Then, it atomically commits the transaction and advances the version associated with the replica. Since these updates are performed sequentially, we guarantee that no deadlock will occur on the secondaries, thus all update batches will commit successfully. By sequentially executing each commit in the master replica, and advancing the version counter, we define a correct serialization order for update transactions, without forcing an *a priori* commit order. Executing update batches in secondary replicas in the same order as in the primary guarantees that all replicas evolve to the same consistent state

```
var global: atomic int version[0..num replicas]
Map pendingTx[1..num replicas]
Connection connB[1..num replicas]

var per client: Connection conn[0..num replicas]
Batch txOps
int txReplica

function begin( boolean readOnly)
    active = true
    if readOnly then
        txVrs = version[0]
        txReplica = SelectReplica()
        wait until version[txReplica] >= txVrs
    else
        txReplica = 0
        txOps = new Batch

function execQuery( Statement query)
    conn[txReplica].execQuery( query)

function execUpdate( Statement update)
    conn[0].execQuery( query)
    txOps.add( update)

function commit()
    if readOnly then conn[txReplica].commit()
    else
        LOCK REPLICA 0
            result = conn[0].commit()
            if NOT result then throw CommitFailed
            newVrs = ++version[0]
        for i:= 1 to num secondary replicas
            pendingTx[i].put( newVrx, txOps)

function threadLoop( int num)
    vrs = version[ num]
    forever
        batch = pendingTx.blockingGetRemove( vrs + 1)
        connB.execBatch( batch)
        LOCK REPLICA num
            connB[num].commit()
            vrs = ++version[num]
```

Fig. 3: MacroDB code.

Read-only transactions All queries from read-only transactions execute directly on secondary replicas, being executed in the context of the caller thread. For providing a single consistent view of the replicated database, MacroDB enforces that a read-only transaction will only execute on a secondary that is up-to-date, i.e., when its version is, at least, equal to the version of the primary when that transaction started. On the beginning of a read-only transaction, the manager reads the current version of the primary replica. This defines the version for the secondary replica in which the transaction will execute, waiting, if necessary, until the selected secondary is up-to-date, i.e., it waits until the version on the selected secondary is, at least, equal to the version read from the primary. This guarantees that the selected secondary is not in an old state, which could potentially lead to a violation of causality for the client. Thus, MacroDB provides clients with a *single copy serializable view* of the replicated database.

Fault Handling When a replica fault is detected, an immediate recovery process is initiated. If the master fails, all currently executing update transactions abort,

and all new update transactions are postponed until a new master is active. A new master is then selected from the set of secondary replicas, replacing the previous one after successfully executing all pending update batches. This guarantees that no update transaction is lost, and that all replicas have the same consistent state, since all update batches have been executed in all of them. At the moment of this selection, new read-only transactions are only forwarded to the remaining secondaries. At this moment the new master becomes active and the system behaves as if a secondary replica had failed. Whenever a secondary replica fails, its current transactions abort, and new read-only transactions are forwarded to the remaining replicas. A new secondary replica is then created and recovered from a non-faulty one.

2.3 Correctness

For the correctness of the system, it is necessary to guarantee that all replicas evolve to the same state after executing the same set of transactions. Also, for guaranteeing that MacroDB provides a single consistent view of the replicated database, it is necessary to guarantee that a transaction is always serialized after all update transactions that may precede it commit. This is achieved because the system enforce the following properties.

Theorem 1. *All replicas commit all update transactions in the same, serializable, order.*

Proof. At the primary, as commits execute atomically in isolation, the serializable order is defined by the order of each commit. Since secondary replicas execute update transactions in a single thread, i.e., sequentially, by the same order, all replicas commit all update transactions in the same order.

Theorem 2. *A transaction is serialized after all update transactions that precede it commit.*

Proof. For update transactions, this is guaranteed by the database engine at the primary. For read-only transactions, MacroDB enforces this property by delaying the beginning of a transaction until the secondary replica has executed all transactions committed at the moment the begin transaction was called.

2.4 Minimizing Contention for Efficient Execution

As presented earlier, the master-slave replication approach used in MacroDB executes update and read-only transactions in different replicas. Thus, read-only transactions never block update transactions and vice-versa, since these execute in distinct replicas. The execution of update transactions in secondary replicas may interfere with read-only transactions, depending on the concurrency control scheme used in the underlying database. Both H2 and HSQL support multi-version concurrency control that allows read-only transaction to not interfere with update transactions. Since only a single update transaction executes at a time, in secondary replicas, this approach guarantee serializable semantics.

Read-only transactions still need to wait until the secondary replica is up-to-date before starting. Our approach, of executing update transaction as a single

batch of updates minimizes the execution time for these transactions, thus also minimizing waiting time.

We can infer, from the results presented in the introduction, that there is contention among multiple threads inside the database engine even when transactions do not conflict. We minimize this contention by reducing the number of transactions that execute in the same replica at the same time - by executing only a fraction of the read-only transactions in each secondary replica and by executing update transactions quickly in a single database operation.

3 Evaluation

In this section we evaluate MacroDB performance, comparing it with a single uncoordinated instance of the database engines (standalone versions), by measuring the throughput of each system. For this comparison we used the TPC-C benchmark, varying the number of clients and workloads. We also evaluated the performance impact of varying the number of secondary replicas of MacroDB. Additionally we also used the TPC-W benchmark.

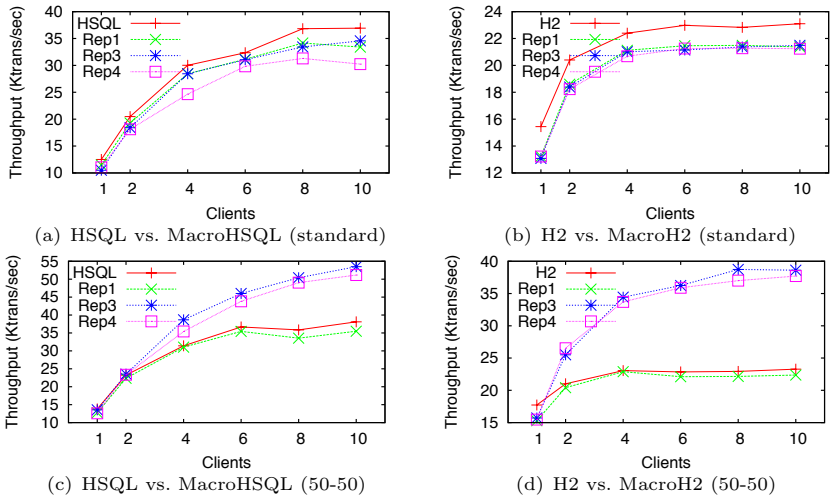
Prototype Considerations Our current MacroDB prototype is built in Java, and includes the necessary runtime system, as well as a custom JDBC driver. By using this simple approach, developers are able to integrate MacroDB into their applications by simply adding its library and modifying the URL used to connect to the database engine, without additional changes to the application code. The number of replicas and underlying database engines used are defined in the connecting URL. When the first client connects to the database, replicas are instantiated and the runtime system is started.

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 2.2.9, and OpenJDK version 1.6. All MacroDB configurations use a full database replication scheme.

3.1 TPC-C

We ran the TPC-C benchmark using 4 different 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), for 2 minutes, on a 4 gigabyte database. The number of clients varied between 1 and 10. The results presented are the average of 5 runs, performed on fresh database copies, disregarding the best and the worst results, and were obtained from the standalone uncoordinated versions of HSQL and H2, and MacroDB using HSQL (MacroHSQL) and H2 (MacroH2), configured with 1, 3 and 4 replicas (Rep1, Rep3 and Rep4, respectively).

Standard workload Figures 4(a) and 4(b) present the results obtained running TPC-C with a standard workload. As expected, under update intensive workloads, our system is unable to benefit from the additional replicas for load balancing, since all updates must be executed on the same replica. Thus, the standalone versions of the database engines outperforms the MacroDB versions. These results also show an important aspect of MacroDB, its overhead. As put in evidence, our system is able to impose a fairly reduced overhead, compared to the standalone versions, ranging between 5% and 14%, even in update intensive workloads.



(a) HSQL vs. MacroHSQL (standard)

(b) H2 vs. MacroH2 (standard)

(c) HSQL vs. MacroHSQL (50-50)

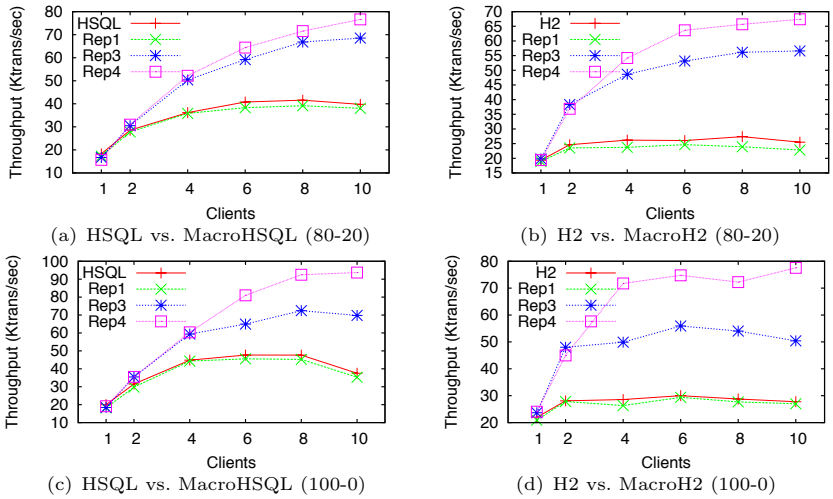
(d) H2 vs. MacroH2 (50-50)

Fig. 4: TPC-C standard and 50-50 workload results

50-50 workload Figures 4(c) and 4(d) present the results obtained running TPC-C with a 50% read and 50% write workload. In moderate update workloads, MacroDB versions of both HSQL and H2 are able to achieve higher throughput than the standalone versions, offering up to 40% and 70% improvements over HSQL and H2 respectively. Although MacroDB is able to scale better than the standalone engines, its scalability is still limited by the nature of the workload. The secondary replicas are able to balance read-only transactions, but the moderate update nature of this workload still imposes great stress at the master replica, thus limiting scalability. This is put in evidence by the, almost negligible, performance difference when MacroDB is configured with 3 or 4 replicas, i.e., 2 or 3 secondary replicas. These results show that, even at considerable update rates, MacroDB is able to achieve a 70% improvement over standalone engines.

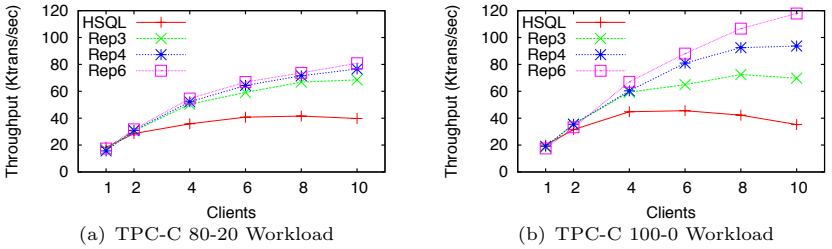
80-20 and 100-0 workloads The nature of read intensive workloads allows MacroDB to take full advantage of its replicated architecture. Both MacroDB versions achieve higher throughput than their standalone siblings, with an increased performance of 93% and 165%, and 166% and 176% performance increase over HSQL and H2, under an 80% (Figures 5(a) and 5(b)) and 100% (Figures 5(c) and 5(d)) read workloads, respectively.

These results put into evidence the benefits of load balancing in reducing contention, since both MacroDB systems achieve higher performances with additional secondary replicas. The performance benefits of MacroDB is only limited by the number of replicas. As presented next, increasing the number of replicas allows MacroDB to further scale, achieving even higher performance figures. It also put into evidence that running 4 database replicas (1 primary and 3 secondary) is not sufficient to fully explore the processing power of our current system. Since current processors offer up to 20 threads per CPU chip [13], current engines considerably underutilize such platforms.



(a) HSQL vs. MacroHSQL (80-20)
(b) H2 vs. MacroH2 (80-20)
(c) HSQL vs. MacroHSQL (100-0)
(d) H2 vs. MacroH2 (100-0)

Fig. 5: TPC-C 80-20 and 100-0 workload results



(a) TPC-C 80-20 Workload
(b) TPC-C 100-0 Workload

Fig. 6: MacroHSQL with 6 replicas

Also, the overhead measured by running a MacroDB with a single replica are consistent in all experiments, with a maximum value of 8%, independently of the nature of the workload, when compared to the standalone DBMS. It is also important to note the lack of scalability of the standalone versions of both database engines. These IMDBs scale fairly well up to 2 or 4 clients, but above that point performance improvements are not significant, in the majority of the experiments, thus showing that a major redesign is needed to improve DBMS performance on current multicore processors.

Additional Replicas To further explore the computational power offered by our setup, we ran TPC-C, using the 80-20 and 100-0 workloads, on a MacroDB with 6 replicas (1 primary and 5 secondaries). The obtained results, presented in Figures 6 for MacroHSQL, show the benefits of increasing the number of replicas on a MacroDB. This increase allows MacroDB to offer performance improvements of up to 234% over standalone engines. These results also put into evidence how current chips are underutilized by current IMDB engines, since MacroDB was able to successfully improve performance, over standalone engines, even when running 6 engines on a single machine.

Memory Usage To measure the practicality of our proposal, we measured the memory overhead imposed by MacroDB, over the standalone database engines

	Replicas		
MacroDB	2	3	4
H2	1.53×	1.56×	1.76×
HSQL	1.64×	2.29×	2.56×

Fig. 7: Memory overhead

Workload	Throughput(WIPS)	
	H2	MacroDB(Rep3)
Browsing Mix	261.6	458.4
Shopping Mix	202	428.6

Fig. 8: TPC-W results

(Figure 7), varying the number of replicas. Contrarily to what may be expected, the memory used by MacroDB is not directly proportional to the number of replicas. This is due to the fact that replicas share immutable Java objects, such as Strings. The obtained results show that, a MacroDB configured with HSQL replicas, uses at most *2.5 times* more memory than the standalone engine, while a MacroDB configure with H2 replicas, uses at most *1.7 times* more memory than the standalone engine, when using a 4 replica configurations. This makes deploying MacroDB practical on single machine multicores, even with large numbers of replicas.

3.2 TPC-W

As an additional experiment, we compared the results obtained running TPC-W benchmark on a single, uncoordinated, H2 engine and a MacroDB using three H2 replicas (Rep3). The results obtained, presented in Figure 8, show the throughput, in web interactions per second (WIPS), obtained running TPC-W browsing and shopping mix, on the machine previously described with a database of 2 gigabytes, for 20 minutes and using 128 emulated browsers, with no thinking time. The performance improvements of MacroDB over the standalone version of H2 ranges from 75% to 112%, thus showing the benefits of our system.

4 Related Work

Several works have addressed the issues of database scalability on multicores. Most of these proposals focus on an engine redesign; on reuse of previous engine work; or on the addition of threads to automate specific procedures or to prefetch data [17,21,3,7,26,9]. These works are complementary to ours, since our focus is to allow existing engines to scale on current hardware without modification.

MacroDB, an example of a Macro-Component [15], follows the path that multicores should be seen as extremely low latency distributed systems [5,1,19,20], extended with shared memory. Thus, techniques previously developed for distributed systems are suitable for re-engineering and deploying on these platforms.

Many database replication studies have proposed solutions for improving service availability and performance [18,24,8,16]. Although complementary to our work, MacroDB builds on some of the techniques from these systems, applying them to multicore systems.

Multimed [19], an adaptation of Ganymed [18] for multicores, has previously explored database replication in single multicore machines. Although similar to our work, MacroDB presents differences that make it unique. First, unlike Multimed, we focus on in-memory databases, which presents different challenges for providing scalability, by not incurring in I/O overhead. Second, our solution aims at providing a single-copy serializable view of the database, instead of relying on weaker snapshot isolation semantics. Finally, by considering a multicore

system as a distributed system *extended with shared memory*, we explore the shared memory for efficient communication between replicas and to expose data for efficient consistency management, load balancing and transaction routing.

5 Final Remarks

In this paper we presented MacroDB, a tool for scaling database systems on multicore platforms. Designed as a transparent middleware platform, it integrates replicas of existing unmodified database engines to offer increased concurrency and performance over standalone DBMS engines. MacroDB is transparent to applications, offering a single serializable view of the database, without need of rewriting the application code, while reducing contention and minimizing response times, by dividing and routing transactions according to their nature.

MacroDB is implemented using a custom JDBC driver and a self contained runtime, and can be used with any JDBC compatible database engine. Thus, performance improvements are obtained without modification to the database engine or the application. It is also easy to configure, allowing database engines and configurations to be specified by the JDBC driver URL.

Our evaluation shows that MacroDB offers 40% to 180% performance improvements over standalone in-memory DBMS, for various TPC-C workloads. Under update intensive workloads (92% update transactions), MacroDB has a reduced overhead of less than 14% when compared to standalone database engines. For TPC-W workloads, MacroDB is able to achieve improvements of up to 112%, over standalone in-memory DBMS.

The memory used by the database replicas is not directly proportional to the number of replicas, as replicas share immutable Java objects, thus making MacroDB practical even with large numbers of replicas.

References

1. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new os architecture for scalable multicore systems. In: Proc. SOSP'09. (2009)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman. (1986)
3. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proc. SIGMOD '11. (2011)
4. Camargos, L., Pedone, F., Wieloch, M.: Sprint: a middleware for high-performance transaction processing. In: Proc. EuroSys '07. (2007)
5. Cecchet, E., Candea, G., Ailamaki, A.: Middleware-based database replication: the gaps between theory and practice. In: Proc. SIGMOD '08. (2008)
6. Chekuri, C., Hasan, W., Motwani, R.: Scheduling problems in parallel query optimization. In: Proc. PODS'95. (1995)
7. Cieslewicz, J., Ross, K.A., Satsumi, K., Ye, Y.: Automatic contention detection and amelioration for data-intensive operations. In: Proc. SIGMOD'10. (2010)
8. Elnikety, S., Dropsho, S., Pedone, F.: Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In: Proc. EuroSys'06. (2006)

9. Giannikis, G., Alonso, G., Kossmann, D.: Shareddb: killing one thousand queries with one stone. In: Proc. VLDB'12. (2012)
10. Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N., Ailamaki, A., Falsafi, B.: Database servers on chip multiprocessors: Limitations and opportunities. In: Proc. CIDR'07. (2007)
11. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: Oltp through the looking glass, and what we found there. In: Proc. SIGMOD '08. (2008)
12. Helal, A.A., Bhargava, B.K., Heddaya, A.A.: Replication Techniques in Distributed Systems. Kluwer Academic Publishers. (1996)
13. Intel: Xenon processor e7 family (2012), http://www.intel.com/
14. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J.: H-store: a high-performance, distributed main memory transaction processing system. In: Proc. VLDB'08. (2008)
15. Mariano, P., Soares, J., Preguiça, N.: Replicated software components for improved performance. In: Proc. InForum'10 (2010)
16. Mishima, T., Nakamura, H.: Pangea: an eager database replication middleware guaranteeing snapshot isolation without modification of database servers. In: Proc. VLDB'09. (August 2009)
17. Papadopoulos, K., Stavrou, K., Trancoso, P.: Helpercoredb: Exploiting multicore technology for databases. In: Proc. PACT '07. (2007)
18. Plattner, C., Alonso, G.: Ganymed: scalable replication for transactional web applications. In: Proc. Middleware'04. (2004)
19. Salomie, T.I., Subasu, I.E., Giceva, J., Alonso, G.: Database engines on multicores, why parallelize when you can distribute? In: Proc. EuroSys '11 (2011)
20. Song, X., Chen, H., Chen, R., Wang, Y., Zang, B.: A case for scaling applications to many-core with os clustering. In: Proc. EuroSys'11. (2011)
21. Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., Kossmann, D.: Predictable performance for unpredictable workloads. In Proc. VLDB'09 (2009)
22. Vandiver, B., Balakrishnan, H., Liskov, B., Madden, S.: Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In: Proc. SOSP'07. (2007)
23. Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: Proc. SRDS 2000. (2000)
24. Wiesmann, M., Schiper, A.: Comparison of database replication techniques based on total order broadcast. IEEE Trans. on Knowledge and Data Engineering 17 (2005)
25. Ye, Y., Ross, K.A., Vesdapunt, N.: Scalable aggregation on multicore processors. In: Proc. DaMoN'11. (2011)
26. Zhou, J., Cieslewicz, J., Ross, K.A., Shah, M.: Improving database performance on simultaneous multithreading processors. In: Proc. VLDB '05. (2005)