Gargamel: boosting DBMS performance by parallelising write transactions

Pierpaolo Cincilla
INRIA UPMC
pierpaolo.cincilla@lip6.fr

Sébastien Monnet
INRIA UPMC
sebastien.monnet@lip6.fr

Marc Shapiro
INRIA UPMC
http://lip6.fr/Marc.Shapiro/

***Abstract*—Parallel transactions in distributed DBs incur high overhead for concurrency control and aborts. We propose an alternative approach by pre-serializing possibly conflicting transactions, and parallelizing non-conflicting update transactions to different replicas. Our system provides strong transactional guarantees. In effect, Gargamel partitions the database dynamically according to the update workload. Each database replica runs sequentially, at full bandwidth; mutual synchronisation between replicas remains minimal. Our simulations show that Gargamel improves both response time and load by an order of magnitude when contention is high (highly loaded system with bounded resources), and that otherwise slow-down is negligible.**

***Keywords*-Cloud computing; distributed DBMSes; scheduling alghorithms;**

I. Introduction

Databases often scale poorly in distributed configurations, due to the cost of concurrency control and to resource contention. The alternative of centralizing writes works well only for read-intensive workloads, whereas weakening transactional properties is problematic for application developers.

Our approach is to classify transactions according to their predicted conflict relations at a front-end to the replicated database. Non-conflicting transactions execute in parallel at separate replicas, ensuring high throughput; both read-only and update transactions are parallelised. Transactions that may conflict are submitted sequentially, ensuring that they never abort, thus optimising resource utilisation. This approach guarantees Parallel Snapshot Isolation (PSI), which is similar to SI, but allows the system to replicate transactions asynchronously [1]. It does not require (costly and slow) global synchronisation. Our system, Gargamel, operates as a front-end to an underlying database, obviating the cost of lock conflicts and aborts. Our approach also improves locality: effectively, Gargamel partitions the database dynamically according to the transaction mix. All this results in better thoughput, response times, and use of resources: our simulations show considerable performance improvement in highly-contended workloads, with negligible loss otherwise.

Our current classifier is based on a static analysis of the transaction text (stored procedures). This approach is realistic, since the business logic of many applications (e.g., e-commerce site OLTP applications) is encapsulated into a small fixed set of parameterized transaction types, and since ad-hoc access to the database is rare [2].

The static analysis is complete, i.e., there are no false negatives: if a conflict exists it will be predicted. However, false positives are allowed: a conflict may be predicted where none occurs at run time.

Our contributions are the following:

- We show how to parallelize non-conflicting transactions by augmenting a DBMS with a transaction classifier front end, and we detail the corresponding scheduling algorithm. Each replica runs sequentially, with no resource contention.
- We propose a simple prototype classifier, based on a static analysis of stored procedures.
- We demonstrate the effectiveness of our approach with extensive simulation results, varying a number of parameters, and comparing against a simple Round-Robin scheduler, the state-of-the-art Tashkent+ [3] and against a centralised-writes system.
- We conclude from the simulations that: *(i)* At high load, compared to competing systems, Gargamel improves response time and throughput of update transactions by 25% and 75% respectively, in the TPC-C benchmark. At low load, Gargamel provides no benefit, but overhead is negligible. *(ii)* The Gargamel approach requires far fewer resources, substantially reducing monetary cost.

The paper proceeds as follows. Section II overviews the Gargamel approach. The justification for its correctness is discussed in Section III. We describe our simulation model in Section IV. The experimental simulation results are detailed in Section V. We compare with related work in Section VI. Finally, we conclude and consider future work in Section VII.

II. Gargamel Architecture

A. Overview

The aim of Gargamel is to dynamically partition the workload into disjoint queues, such that no transaction in some queue conflicts with one in any other. Two queues can execute in paralle on separate replicas. To avoid aborts, transactions within a queue execute sequentially.

This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208) and by ANR project Prose (ANR-09-VERS-007-02).

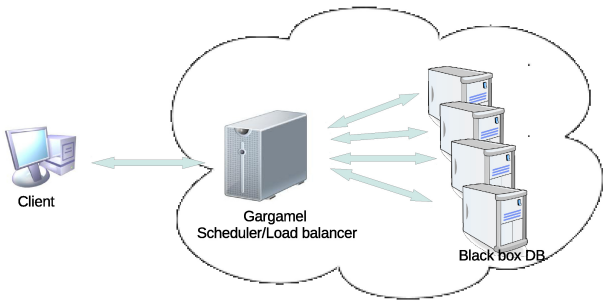


Figure 1. System architecture

The component that predicts possible conflicts between transactions is called the *transaction classifier*. The classifier checks each new transaction T against recently-committed and already-scheduled ones. T is queued after all the transactions with which it might conflict; if no conflicts are predicted, T starts a new queue.

The conflict definition is determined by the isolation level. Our current classifier implements PSI, i.e., transactions conflict iff their write-sets intersect. The classifier implementation for the TPC-C benchmark is based on a static analysis of the workload.

The Gargamel load balancer is interposed between clients and database replicas (called *workers* hereafter), as illustrated in Figure 1.

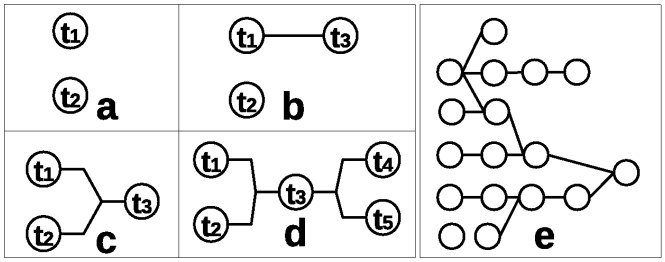
When it receives a new transaction, Gargamel checks for conflicts against already-scheduled transactions, and assigns it to a queue accordingly; more details later.

B. Scheduling Algorithm

As transactions enter and exit the system, conflict relations appear and disappear in complex ways. To keep track of this, Gargamel maintains generalized queues called *chains*. Conflicting transactions execute sequentially at a single worker; non-conflicting transactions are assigned to parallel queues, which execute, without mutual synchronisation, at different workers.

For some incoming transaction t, if t is classified as conflicting with t', then t is queued after t'. If t conflicts with transactions t' and t'' that are in two distinct queues, then the transaction is scheduled in a chain *merging* the two queues. Similarly, if two transactions t' and t'' both conflict with transaction t, but t' and t'' do not conflict with each other, then they will be put into two distinct queues, both after t. We call this case a *split*.

Figure 2 presents some examples. Clients submit transactions t_1, t_2, etc. Initially, the system is empty: t_1 does not conflict with any transaction; therefore the scheduler allocates its first chain, containing only t_1. When the scheduler receives t_2, it compares it with the only other transaction, t_1. If they conflict, the scheduler will append t_2 at the end of the queue containing t_1; otherwise, the scheduler assigns



An initially empty system receives t_1, t_2, etc.: *(a)* t_1 and t_2 do not conflict. *(b)* t_1 and t_2 do not conflict, and t_3 conflicts with t_1 but not with t_2. *(c)* t_3 conflicts with both t_1 and t_2. *(d)* Merge and Split. *(e)* General chains.

Figure 2. Example scheduling scenario.

t_2 to a new queue (Figure 2(*a*)).

Consider the latter case: transaction t_3 arrives; the scheduler compares it to t_1 and t_2. If t_3 conflicts with neither of them, it is placed into a new queue. If t_3 conflicts with a single one, it is queued after it (Figure 2(*b*)). If it conflicts with both, the two queues merge (Figure 2(*c*)). If transactions t_4 and t_5 both conflict with t_3 but not with each other, they will be on parallel queues but both after t_3 (the queue splits (Figure 2(*d*))). Repeating the algorithm, Gargamel computes chains that extend, merge or split according to conflicts between pairs of transactions (Figure 2(*e*)).

The number of classifications is, in the worst case, equal to the number of transactions queued and not yet committed plus the last transactions that update items affected by the incoming transaction.

Once a worker has finished a transaction it propagates the write-set to all other workers using causal order broadcast [4]. The write-set propagation is done in the background. Workers wait for propagation of write-sets of conflicting transactions to proceed before starting execution. For example to execute a transaction after a merge, the worker waits to receive the write-sets of transactions queued before the merging point.

III. Correctness

Gargamel ensures the Parallel Snapshot Isolation (PSI) property [1] thanks to the following features:

- *(i)* Gargamel ensures that conflicting transactions do not execute concurrently. This implies *a fortiori* that a transaction commits only if its updates do not conflict with a concurrent transaction.
- *(ii)* Workers propagate their updates using causal order broadcast. When a replica receives the write-set of some transaction, it is guaranteed to have already received the write-sets of preceding transactions.
- *(iii)* Each replica's database engine ensures SI locally. Causal order propagation of updates ensures that a

Parameter	Value		
Default number of workers	100		
Default incoming rate	150 trans/s		
Default load	150 trans/s for ~ 28 s		
Warehouses (TPC-C)	10		
	(ms)	**Mean**	**Variance**
Site-worker msg latency	.06	1	.005
Consensus latency	180	1	.01
Client-site msg latency	0		
Apply write-set	0		
TPC-C transactions [5], [6], [7]	**Duration** (ms)	**Mean**	**Variance**
New Order	700	1	.025
Payment	660	1	.028
Order Status	680	1	.028
Delivery	660	1	.035
Stock Level	1010	1	.022

Transaction durations are taken from a Gaussian distribution with the indicated mean and variance. The numerical parameters of TPC-C are taken from the referenced measurements [6].

Table I
SIMULATION PARAMETERS

transaction t can't commit after t' at any site if t' has observed the updates of t.

(iv) A worker does not start an update transaction until it has received the updates of preceding transactions.

The conjunction of these properties ensures that concurrent transactions commute.

To explain property *(iv)* in more detail, note that, since communication between the scheduler and replicas is asynchronous, a replica might receive an update transaction from the scheduler before it is ready. The replica delays the execution of the new transaction until it has received the updates from its predecessors. Read only transactions are never delayed, they can read stale values.

The scheduler maintains the identifier of the last transaction that updated any given data item. The classifier maintains a list of data items affected by any given transaction type. Therefore it is easy to extract the identifier of the latest transaction that conflict with any new transaction.

When the scheduler sends a new transaction to some replica, it piggy-backs the list of recent conflicting transactions executed at a different replica (this may occur, for instance, when chains merge). When it receives this, the replica checks that the corresponding updates have been received; if not, the new transaction is delayed.

IV. SIMULATION MODEL

To evaluate the Gargamel approach, we have implemented a discrete event simulator. In this section we present the simulation model and describe the benchmark we use. The experiments and results are discussed in Section V.

A simulation has two phases. In the first phase, clients submit transactions to the scheduler at the specified incoming rate. During the second phase, no more transactions are submitted. It lasts until every transaction has finished executing. The first phase lasts approximately 28 s.

Unless specified otherwise, all simulations use the parameters presented in Table I.

A. TPC-C simulation

TPC-C [7] is a standard benchmark for OLTP transactions. The numeric simulation parameters of Table I are taken from actual TPC-C measurements [6]. We simulate TPC-C with ten warehouses.

It is composed of five kinds of transactions: New-Order (NO), Payment (P), Order-Status (OS), Delivery (D), and Stock-Level (SL). 92% of the workload consists of update read-write transactions; the remaining 8% are read-only (OS and SL).

A $NO(w, d, I)$ transaction adds an order to a warehouse. Its parameters are a warehouse w, a district d, and a list of items I. Each item $i(w', d') \in I$ has two parameters: its warehouse ID w' and the item ID d'. An I list contains between 5 and 15 elements. NO transactions occur with high frequency and relatively costly.

The parameters of a Payment transaction $P(w, c, cw, cd)$ are a warehouse ID w, a customer c, a customer warehouse cw, and a customer district cd. The customer c is selected 60% of the time by name, and 40% of time by unique identifier. Homonyms are possible in the former case. P and NO transactions dominate the workload.

The single parameter of a Delivery transaction D_w is the warehouse ID w.

Our classifier is based on a very simple static analysis: two transactions are considered to conflict: (i) if they update the same column of a table, and (ii) unless it is clear from the analysis that they never update the same row. In the case of TPC-C, conflicts may happen between pairs of the same transaction type (NO and NO, P and P, D and D) and between Payment and Delivery transactions. Table II shows which transactions conflict according to their parameters.

Because homonyms cannot be checked statically, transaction classification admits false positives between two Payment transactions and between a Payment and a Delivery transaction. If the customer is identified by name, the classifier conservatively assumes that a conflict is possible. Around 0.04% of classifications return a false positive, i.e. they predict a conflict that will not appear at run-time.

Delivery transaction pairs do not cause false positives because the customer selection here is based on the lowest ID (representing the oldest NO), which is unique for a given snapshot.

B. Round Robin, Centralised-Writes and Tashkent+ simulation

Our simulations compare Gargamel with a simple Round Robin scheduler, and with the state-of-the-art Tashkent+ [3].

Transaction pairs	Conflict condition
$NO(w_1, d_1, I_1) \times NO(w_2, d_2, I_2)$	$(w_1 = w_2 \wedge d_1 = d_2) \vee I_1 \cap I_2 \neq \emptyset$
$P(w_1, c_1, cw_1, cd_1) \times P(w_2, c_2, cw_2, cd_2)$	$(w_1 = w_2) \vee ((cw_1 = cw_2 \wedge cd_1 = cd_2) \wedge (c_1 = c_2))$
$D(w_1) \times D(w_2)$	$w_1 = w_2$
$D(w_1) \times P(w_2, c_2, cw_2, cd_2)$	$w_1 = cw_2$

The subscripts represent two concurrent transactions. Please refer to Section IV-A for an explanation of variable names.

Table II
CONFLICTS OF TPC-C UNDER SNAPSHOT ISOLATION

We also compare Gargamel with a centralised-writes system when possible.

Round-Robin aims to maximise throughput by running as many transactions in parallel as possible. It works as follows. Transactions are assigned to each worker in equal portions and in circular order. Because concurrent transactions proceed in parallel, Round-Robin suffers from a lot of abort-restarts, i.e., wasted work.

A Centralised-Write system runs all read-only transactions concurrently, but serialises all update transactions at a single worker in order to avoid wasted work. It can be considered as an idealized version of Ganymed [8]. Centralised-Write was simulated quite simply by classifying all update transactions as mutually-conflicting; thus all update transactions are put into a single queue, executed by a single worker. Our simulations show that Centralised-Write is overly conservative on standard benchmarks.

Like Round-Robin, Tashkent+ aims to maximise throughput, but optimises the assignment of transactions to workers by ensuring that the working set of the transactions sent to a worker fits into the worker's main memory. Workers are assembled in groups executing the same set of transaction types. To balance the load, Tashkent+ monitors each worker's CPU and disk usage, and rearranges groups, by moving workers from the least loaded group to the most loaded group.

Tashkent+ estimates the working set of an incoming transaction by examining the database execution plan. Our simulated Tashkent+ extracts the execution plan from TPCC-UVA [9], an open source TPC-C implementation.

Our simulator implements the Tashkent+ group allocation/re-allocation algorithm as described in the literature [3]. Since CPU and disk usage are not significant in this simulation, we estimate the load by the ratio of busy to available workers. Replica allocation and re-allocation are implemented in such a way that the balance remains optimal all the time. Our simulations are favorable to Tashkent+ because we assume that re-allocation has no cost.

As the literature shows that Tashkent+ improves performance by reducing disk access, our simulation takes this into account by reducing the duration of every transaction by 10% under Tashkent+. Nevertheless, our simulations hereafter show that Tashkent+ suffers from lost work under TPC-C.

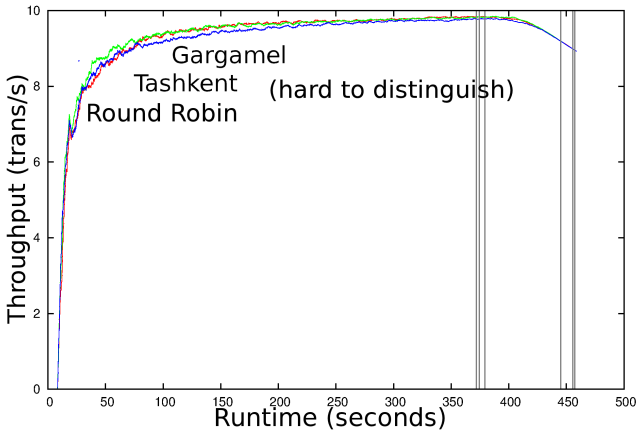


Figure 3. Throughput (TPC-C, 10 trans/s).

All systems perform transactions at a single replica and then diffuse the write-set to other replicas.

V. EXPERIMENTAL RESULTS

We measure transaction throughput, response time, and amount of resources consumed, comparing Gargamel, Tashkent+ and Round-Robin. When relevant, we compare also with Centralised-Write. This set of experiments is based on TPC-C, varying the transaction incoming rate.

A. Performance

When the incoming rate is low, parallelism is low, therefore conflicts are not an issue. In such a situation, all schedulers are basically equivalent. For instance, Gargamel will schedule each incoming transaction in a new chain and execute it immediately. Figure 3 confirms that at 10 trans/s (transactions per second) Gargamel, Tashkent+ and Round-Robin have indistinguishable throughput. Vertical lines show, for each system, when the first and the last client stop submit transactions.

Things get more interesting at high incoming rates. Figure 4 compares the throughput of the three systems at 200 trans/s. Figure 5 shows maximum throughput, varying the incoming rate. The two figures show that Gargamel exhibit a significant improvement compared to the other systems during the first phase (while clients submit transactions). In the second phase (when no more transactions are

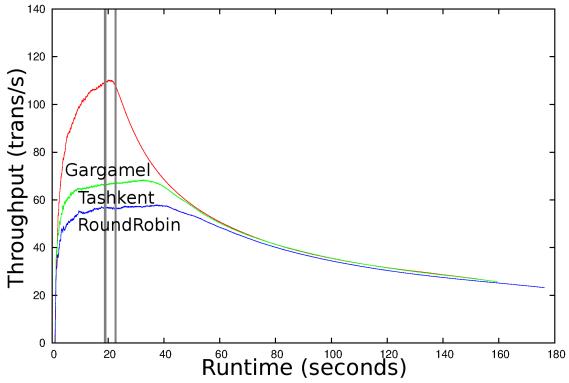


Figure 4. Throughput (TPC-C, 200 trans/s).

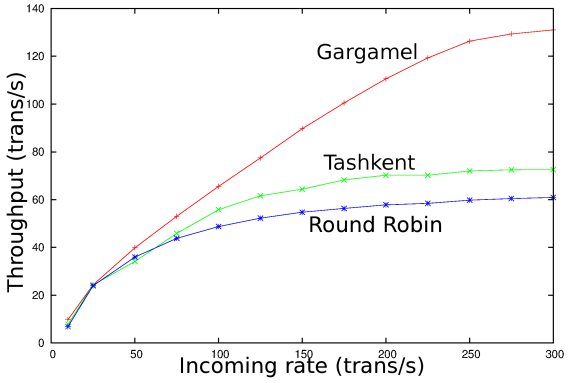


Figure 5. Maximal throughput (TPC-C).

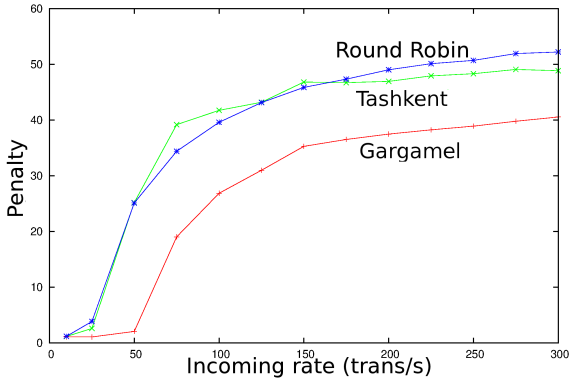


Figure 6. Penalty ratio (TPC-C).

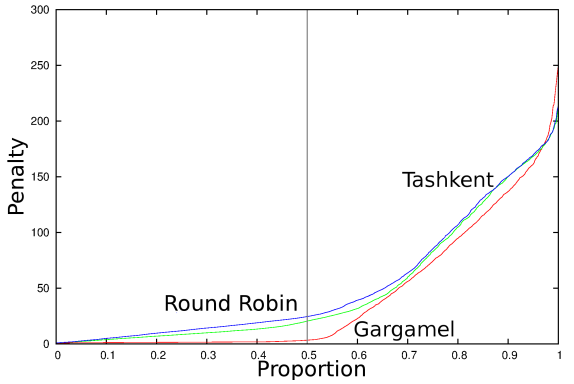


Figure 7. Cumulative distribution function of penalty ratio (TPC-C).

submitted), parallelism decreases and the throughput of all three systems decreases consequently.

Figure 5 shows that the improvement of Gargamel over Tashkent+ and Round-Robin grows with the incoming rate. Tashkent+ and Round-Robin both flatten out at $\simeq$ 150 trans/s. Gargamel's maximum throughput is twice that of Tashkent+ and Round-Robin.

At 250 trans/s, Gargamel saturates the available workers, and, at constant number of workers, any increase in incoming rate does not provide any improvement.

We estimate the response time by measuring the "penalty ratio," i.e., response time (time elapsed between transaction submission and transaction commitment) divided by transaction duration. The lower the scheduling delays, the lower the penalty ratio suffered by clients. Figures 6 and 7 show the penalty ratio and its CDF, comparing Gargamel, Tashkent+ and Round-Robin. Figure 6 shows the average penalty ratio of ten runs, Figure 7 shows the CDF of one run. Gargamel's penalty is approximately 20% lower than Tashkent+ and Round-Robin. 51% of the transactions in Gargamel suffer a penalty of 4 or less, whereas this is the case of only 10% of transactions (approximately) in the competing systems.

The speedup is estimated by dividing the sequential execution time by the total execution time. The speed-up improvement is low (Figure 8): while most transactions execute with little delay (as shown by the penalty ratio CDF) the longest-waiting transaction is delayed almost identically in all three systems. This is due to the fact that conflicting transactions must be serialised anyway; the TPC-C workload is dominated by the New Order and Payment transactions, and two New Order transactions can conflict, as well as two Payment transactions. The serialization of conflicting New Order and Payment transactions dominate the speedup. The speed-up of all three systems flatten at an incoming rate of around 50 transactions per second. For higher incoming rates even if the throughput is higher (see Figure 5) the time to execute serially conflicting New Order and Payment transactions remain the same.

B. Resource Utilisation, Bounded Workers

Our next experiments examine resource utilisation and queue size in our systems.

The bottom part of Figure 9 shows the number of busy workers as the simulation advances in time. The top part shows the number of queued transactions (note that for readability the scale of the ordinate differs between the bottom and the top).

For the default number of workers (100), at the default incoming rate (150 trans/s), Gargamel always finds an available worker; queue size is close to zero. This means that the number of parallelizable transactions remains lower than the number of workers, at all times.

As Gargamel, Centralised-Write always finds an available worker, but it paralelise mutch less than other systems, then

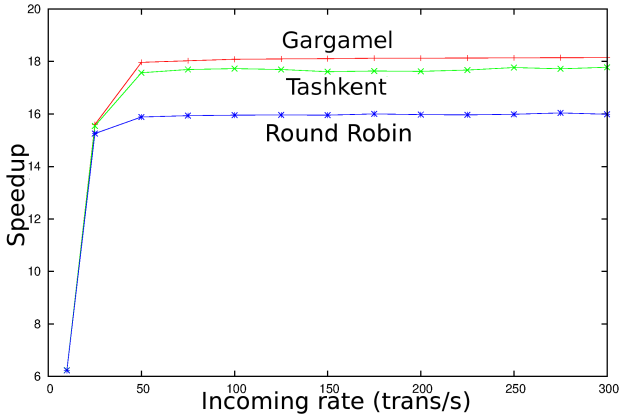


Figure 8. Speedup for TPC-C

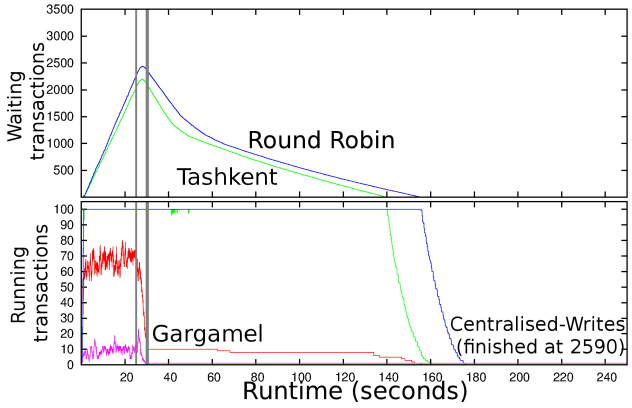


Figure 9. Resource utilisation for TPC-C at 150 trans/s

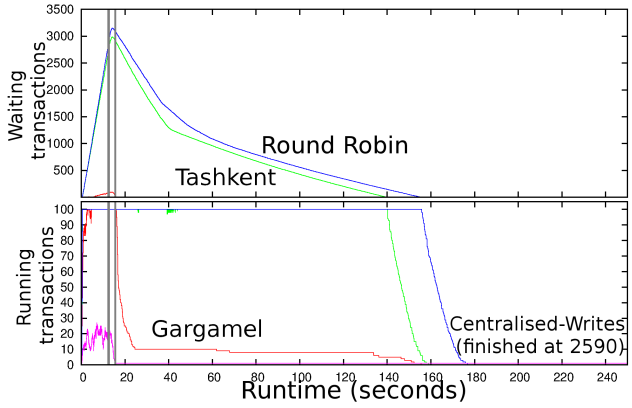


Figure 10. Resource utilisation for TPC-C at 300 trans/s

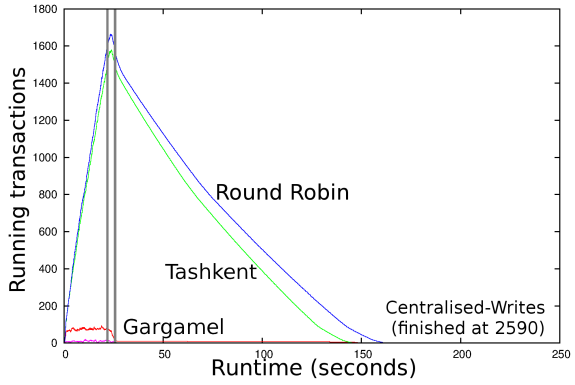


Figure 11. Resource utilisation, unbounded workers (TPC-C, 175 trans/s).

its execution time is mutch longher.

In contrast, the policy of both Tashkent+ and Round-Robin is to execute as many transactions as possible in parallel, as soon as they arrive. However, since many of those transactions conflict, there are many aborts, and they do not make progress. They quickly saturate the number of workers, incoming transactions are delayed, and queue size grows rapidly.

In the second phase, after transactions stop arriving (incoming rate goes to zero), Gargamel frees most of the workers. Indeed, at this point, all the read-only and non-conflicting transactions finished executing; Gargamel only needs a few workers for the remaining chains of conflicting transactions.

In contrast, Tashkent+ and Round-Robin continue to saturate the workers by attempting to parallelise conflicting transactions. At some point during the second phase, Tashkent+ re-assigns groups and continues to empty the queue of waiting transactions more slowly. This is because, at this point, all the read-only and non-conflicting transactions have terminated. The remaining transactions conflict with higher probability.

We have also simulated a rate of 300 trans/s (Figure 10). Even for Gargamel the load is too high for the default number of workers, and Gargamel builds a (very small) queue during the first phase.

C. Resource Utilisation, Unbounded Workers

We now consider a system where the number of workers is unbounded, e.g., an elastic cloud computing environment. In this case, both Tashkent+ and Round-Robin mobilise a much higher amount of resources than Gargamel. Figure 11 shows that, at the end of the first phase, Tashkent+ needs 1500 concurrent workers, whereas Gargamel needs less than 100.

This can be directly translated into monetary cost. Considering that Amazon EC2 advertises a CPU cost of approximately 1 euro/hour [10], Figure 12 plots the cost of the three systems, varying the incoming rate, with both the

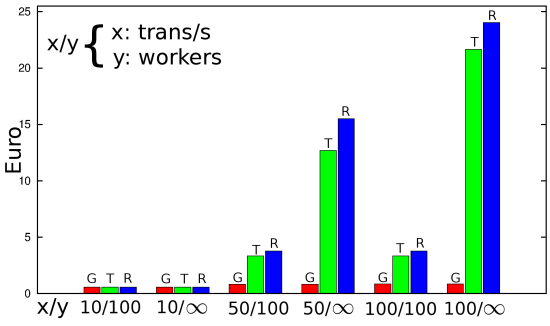


Figure 12. Cost comparison

default number of workers, and with unbounded resources. At low incoming rate, all systems use the same small amount of resources. As the rate increases, Tashkent+ and Round-Robin use as many workers as possible in order to maximise parallelism. With bounded workers, once all workers are in use, the cost of Tashkent+ and Round-Robin remains the same, even if the incoming rate increases; if the number of workers is unbounded, the resource usage of Tashkent+/Round-Robin is proportional to the incoming rate. At 100 trans/s, with unbounded workers, Gargamel is 25 times cheaper than Tashkent+.

VI. Related Work

Database replication potentially improves performance and availability, by allowing several transactions to proceed in parallel at different replicas. This works well for read-only transactions, but remains challenging in the presence of updates. Concurrency control is an expensive mechanism; it is also wasteful to execute conflicting transactions concurrently, since one of them must abort and restart. This well-known issue prevents DBMSes from making effective use of modern low-cost concurrent architectures such as multicores, clusters, grids and clouds. For instance, Salomie et al. report that the performance of both PostgreSQL and MySQL degrade on multicore architectures [11]. As we have shown in this paper, this is also the case of the state-of-the-art Tashkent+ system [3]: as it aims to make full use of all available parallelism, it suffers high abort rates on standard benchmarks.

One approach to circumvent this problem is to parallelise only read-only transactions, and to centralise all update transactions at a single replica. This is the case of both Ganymed [8] (in a cluster) and Multimed [11] (on a multicore machine). However, this conservative approach results in poor response times, as it ignores that non-conflicting transactions can be parallelised, even if they perform updates. Our simulations, hereafter, confirm this intuition. In contrast, Gargamel is capable of parallelising at least some update transactions. Therefore, it does not need a master replica, which constitutes a scalability bottleneck.

An alternative is to give up on strong transactional consistency [12]. Thus, systems such as Facebook, Zynga and Twitter make heavy use of *memcached*, a key-value store with weak consistency guarantees [13]. This approach is promising for some particular classes of application, but is bug-prone and difficult to get right for application developers, and therefore not considered in this paper.

H-Store [2] is a DBMS designed and optimized for OLTP applications. It requires the complete workload to be specified in advance as statically defined stored procedures. This advance knowledge allows H-Store to partition and to parallelise the load between different single-threaded replicas operating in a share-nothing environment. Under this assumption, H-Store improves the performance by orders of magnitude compared with other commercial databases. Gargamel also parallelises the load between single-threaded replicas, but does so above an unmodified DBMS. Furthermore, Gargamel requires only approximate advance knowledge, encapsulated in the conflict classifier, and does not require the whole workload to be specified in advance.

The system of Pacitti et al. [14] is a lazy multi-master replicated database system. It enforces a total order of transactions, by using reliable multicast and a timestamp-based message ordering mechanism. Gargamel, in contrast, can execute non-conflicting update transactions in parallel; in addition, it minimises resource utilisation. Furthermore, since Gargamel ensures proper transaction ordering in the background, it scales well to a WAN environment, in contrast to the *a priori* total order approach of Pacitti et al.

Sarr et al. [15] introduce a solution for transaction routing in a grid. Their system, like Gargamel, is conflict-aware. However, they check for conflicts only in order to propagate updates among replicas in a consistent way; they do not serialise conflicting transactions, as Gargamel does.

A conflict-aware scheduling system is proposed by Amza et al. [16], [17]. Their system ensures 1-copy-SI by executing all update transactions in all replicas in a total order. Gargamel parallelizes non-conflicting write transactions and transmits the write-sets off the critical path. Moreover Gargamel executes a given transaction only once, at a single replica, which ensures that replicas do not diverge in the presence of non-determinism.

Middle-R [18], similarly to Gargamel, serialises conflicting transactions at one replica classifying them before execution. Middle-R associates each (update) transaction to one or more *conflict class*. Each conflict class has a master copy. Transactions are executed at the master copy of the conflict they belong. After execution the write-set is diffused to other replicas. The key difference between Gargamel and Middle-R is that Middle-R statically associates replicas to transaction class and does not avoid aborts. At the opposite Gargamel partitions the database dynamically according to the transaction mix and never aborts transactions.

VII. Conclusion And Future Work

We described Gargamel, a middleware that maximizes parallelism in distributed databases while limiting the amount of wasted work. Instead of parallelising conflicting transactions, like previous systems, Gargamel uses a transaction classifier to pre-serialise possibly conflicting transactions. This improves the average response time (lowers the penalty ratio) and dramatically decreases resource utilisation with respect to competing approaches.

Since Gargamel runs at the load balancer, it does not impose any changes to existing DBMS engines.

We show, by simulation, that under high incoming rates and with a good classifier, Gargamel performs considerably better than both Tashkent+ and Round-Robin: it provides higher throughput, lower response time, and consumes fewer resources.

To provide PSI we consider only write-write conflicts. We plan to extend Gargamel to provide higher isolation levels (e.g. one-copy serializability). We expect that a higher isolation level will decrease even more the resource utilisation, with respect to certification-based approaches.

In future work, we plan to implement a distributed version of Gargamel to support the co-existence of multiple concurrent classifying load balancers, one for each site. Load balancers synchronisation can be done optimistically to maximize performance. In the distributed version of Gargamel, we plan to study a scheduling policy that favors performance over resource utilisation, executing the prefix of split chains at all sites in parallel. If a site is less loaded or has faster computers than the others, it will send its write-set early, allowing the slower site(s) to catch up.

References

[1] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Symp. on Op. Sys. Principles (SOSP)*. Cascais, Portugal: Assoc. for Computing Machinery, Oct. 2011, pp. 385–400.

[2] M. Stonebraker, S. R. 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 *VLDB*, Vienna, Austria, 2007.

[3] S. Elnikety, S. Dropsho, and W. Zwaenepoel, "Tashkent+: memory-aware load balancing and update filtering in replicated databases," in *Euro. Conf. on Comp. Sys. (EuroSys)*, vol. 41, ACM SIG on Operating Systems (SIGOPS). Lisbon, Portugal: Assoc. for Comp. Machinery, Mar. 2007, pp. 399–412.

[4] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[5] "Methods to implement the random database population," http://db.apache.org/derby/javadoc/testing/org/apache/derbyTesting/system/oe/util/OERandom.html.

[6] Hewlett-Packard, "TPC benchmark C: full disclosure report for HP ProLiant BL685c G7," 2011. [Online]. Available: http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA3-5312ENW.pdf

[7] TPC, "TPC benchmark C Standard Specification Revision 5.0," 2001. [Online]. Available: http://www.tpc.org/tpcc/spec/TPCC_v5.pdf

[8] C. Plattner and G. Alonso, "Ganymed: Scalable replication for transactional web applications," in *Int. Conf. on Middleware (MIDDLEWARE)*, 2004, pp. 155–174.

[9] D. R. Llanos, "TPCC-UVA: An open-source TPC-C implementation for global performance measurement of computer systems," *ACM SIGMOD Record*, December 2006, iSSN 0163-5808.

[10] A. Inc, *Amazon Elastic Compute Cloud (Amazon EC2)*. http://aws.amazon.com/ec2/#pricing: Amazon Inc., 2008. [Online]. Available: http://aws.amazon.com/ec2/#pricing

[11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso, "Database engines on multicores, why parallelize when you can distribute?" in *Euro. Conf. on Comp. Sys. (EuroSys)*. Salzburg, Austria: ACM, 2011, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966448

[12] W. Vogels, "Eventually consistent," *ACM Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008.

[13] J. Petrovic, "Using memcached for data distribution in industrial environment," in *ICONS*. IEEE Computer Society, 2008, pp. 368–372.

[14] E. Pacitti, M. T. Özsu, and C. Coulon, "Preventive multimaster replication in a cluster of autonomous databases." in *Euro-Par'03*, 2003, pp. 318–327.

[15] I. Sarr, H. Naacke, and S. Gançarski, "DTR: Distributed transaction routing in a large scale network," in *High Performance Computing for Computational Science (VECPAR)*, ser. Lecture Notes in Comp. Sc., J. M. Palma, P. R. Amestoy, M. Daydé, M. Mattoso, and J. a. C. Lopes, Eds. Toulouse, France: Springer-Verlag GmbH, Jun. 2008, vol. 5336, pp. 521–531.

[16] C. Amza, A. L. Cox, and W. Zwaenepoel, "Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites." in *Middleware*, ser. Lecture Notes in Computer Science, M. Endler and D. C. Schmidt, Eds., vol. 2672. Springer, 2003, pp. 282–304. [Online]. Available: http://dblp.uni-trier.de/db/conf/middleware/middleware2003.html#AmzaCZ03

[17] ——, "Conflict-aware scheduling for dynamic content applications," in *USENIX Symposium on Internet Technologies and Systems*, 2003.

[18] M. Patiño Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "MIDDLE-R: Consistent database replication at the middleware level," *Trans. on Computer Systems*, vol. 23, no. 4, pp. 375–423, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1113574.1113576