Ganymed: Scalable and Flexible Replication

Christian Plattner Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, CH-8092 Zürich, Switzerland
{plattner, alonso}@inf.ethz.ch

Abstract

Transactional web applications play an important role in e-commerce environments and lead to significant scalability challenges to existing database architectures. Replication is an often proposed solution. Existing systems, however, often cannot guarantee consistency, scalability while supporting arbitrary access patterns. Ganymed, a middleware based replication solution, is intended to add scalability and adaptability to existing databases without having to give up consistency or having to change client software. By using a simple scheduling algorithm, RSI-GPC, the overall system footprint is small and therefore very efficient and easily extensible.

1 Introduction

Recent research in the area of replication for transactional web applications is mostly based either on middleware approaches [1, 2, 3] or on techniques that cache data at the edges of the network [4, 5, 6]. The use of caching can greatly reduce response times for large setups, however, strong consistency has to be given up. Middleware centered solutions that schedule requests over sets of replicas in turn can offer the same degree of consistency as single instance databases. Unfortunately, such systems suffer from at least one of the following problems: client software has to be modified to be able to work with the middleware (e.g., transaction access patterns have to be predeclared), efficient scheduling is only possible if the data is partitioned statically (e.g. leading to limited query patterns), certain features of the replicas can not be used (e.g., triggers are not supported), database logic is duplicated at the middleware level (e.g., conflict resolution, leading to limited scale-out due to table level locking). Middleware solutions often become the bottleneck of the system due to their complexity.

Our approach, Ganymed, is also a middleware based approach. However, we avoid all of those problem. Ganymed is tailored to the needs of typical transactional web applications. It guarantees full consistency while at the same time offering good scale out. There is no need to change client software, our current implementation features a JDBC driver that can be easily plugged into existing applications. The middleware scheduler, using our novel RSI-GPC algorithm, is able to balance transaction to a set of replicas, which can eben be different DBMS's. The data does not need to be partitioned, the system uses a fully replicated shared-nothing architecture. Update transactions that fire triggers do not impose a problem. The scheduler also does not involve SQL statement parsing or table level locking, leading to an efficient system with a small footprint.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Our TPC-W trace based evaluations show that Ganymed offers almost linear scalability for typical e-commerce workloads. Due to the graceful degradation properties of Ganymed, the availability of a database system can also be significantly improved.

2 The Ganymed Approach

Ganymed is a system which is able to execute transactions over a set of database replicas. There are two key ideas in Ganymed: The first is the separation of update and read-only transactions. Updates are handled by a master replica, queries are processed on a set of slave replicas. The second idea is to use snapshot isolation for the read-only transactions. Snapshot Isolation (SI) [7, 8] is a multiversion concurrency control mechanism used in databases. Popular database engines that use SI include Oracle [9] and PostgreSQL [10]. SI has the important property that readers are never blocked by writers, it is ideal for separating updates and queries. In Ganymed, all update transactions on the master result in writesets which have to be applied on the slave replicas. The use of snapshot isolation makes it then possible to install a stream of writesets from the master to the slaves without having conflicts with the read-only transactions on the slaves. This property of SI is a big win in comparison to systems that use traditional *two phase locking* (2PL), where many non-conflicting updates may be blocked by as simply as one reader.

However, Ganymed is not limited to database backends based on snapshot isolation. Whereas the slaves must implement snapshot isolation, this is not necessary for the master. Actually, the master replica can use any concurrency protocol that ensures that transactions do not see uncommitted data. The only requirement for a master is the ability to extract writesets of update transactions. As our evaluations have shown, this is achievable for popular database products like DB2, Oracle or PostgreSQL.

2.1 The RSI-GPC Algorithm

RSI-GC is short for *Replicated Snapshot Isolation with General Primary Copy*. A RSI-GPC based scheduler is responsible for a set of n fully replicated backend databases. One of the replicas is used as $master$, the other $n-1$ replicas are the $slaves$. The slaves must always implement snapshot isolation. The scheduler makes also a clear distinction between *read-only* and *update* transactions, which have to be marked by the client application in advance.

Update transactions: SQL statements of any arriving update transaction are directly forwarded to the master, they are never delayed. The scheduler takes notice of the order in which update transactions commit on the master. After a successful commit of an update transaction, the scheduler makes sure that the corresponding writeset is sent to the $n-1$ slaves and that every slave applies the writesets of different transactions in the same order as the corresponding commit occurred on the master replica. The scheduler uses also a global database version number. Whenever an update transaction commits on the master, the global database version number is increased by one and the client gets notified about the commit. Writesets get tagged by the version number which was created by the corresponding update transaction.

Read-Only Transactions: read-only transactions can be processed by any slave replica mode. The scheduler is free to decide on which replica to execute such a transaction. If on a chosen replica the latest produced global database version is not yet available, the scheduler must delay the creation of the read-only snapshot until all needed writesets have been applied to the replica. For clients that are not willing to accept any delays for read-only transactions there are two choices: either their queries are sent to the master replica, therefore reducing the available capacity for updates, or the client can set a staleness threshold. A staleness threshold is for example a maximum age of the requested snapshot in seconds or the condition that the client sees its own updates. The scheduler can then use this threshold to choose a replica. Once the scheduler has chosen a replica and the replica created a snapshot, all consecutive operations of the read-only transaction will be performed using that snapshot. Due to the nature of SI, the application of further writesets on this replica will not conflict with this or any other running read-only transaction.

Scheduler Performance: Due to its simplicity, there is no risk of a RSI-GPC scheduler becoming the bottleneck in the system. In contrast to other middleware based schedulers, like the ones used in [3, 1], this scheduling algorithm does not involve any SQL statement parsing or concurrency control operations. Also, no row or table level locking is done at the scheduler level. The detection of conflicts, which can only happen during updates, is left to the master replica. Moreover, unlike [3, 11], RSI-GPC does not make any assumptions about the data partition, organization of the schema, or answerable and unanswerable queries.

Fault Tolerance: Since only a small amount of state information must be kept by a RSI-GPC scheduler, it is even possible to construct parallel working schedulers. This helps to improve the overall fault tolerance. In contrast to traditional eager update-everywhere systems, where every replica has its own scheduler that is aware of the global state, the exchange of status information between a small number of RSI-GPC schedulers can be done very efficiently. Even in the case that all schedulers fail, it is possible to reconstruct the overall database state: a replacement scheduler can be used and its state initialized by inspecting all available replicas. In the case of a failing slave replica, the scheduler simply ignores it until it has been repaired by an administrator. However, in the case of a failing master, things are a little bit more complicated. By just electing a new master the problem is only halfway solved. First, the new master replica must feature the same concurrency control protocols as the old one for a seamless changeover. Second, the scheduler must also make sure that no updates from committed transactions get lost, thereby guaranteeing ACID durability. This goal can be achieved by sending commit notifications to clients after the writesets of update transactions have successfully been applied on a certain, user defined amount of replicas.

3 Prototype Architecture

The Ganymed system is already a full working system. Its main component is a lightweight middleware scheduler that implements the RSI-GPC algorithm to balance transactions from clients over a set of database replicas. Clients are typically application servers in e-commerce environments. From the viewpoint of such clients, the Ganymed scheduler behaves like a single database which offers SI for read-only transactions and (depending on the master replica) a set of concurrency control options for update transactions. Our working system prototype is entirely based on Java.

Figure 1 shows the main components of the architecture. Applications servers connect to the Ganymed scheduler through a custom JDBC driver. The Ganymed scheduler then distributes incoming transactions over the master and slave replicas. Writesets, denoted as WS, are extracted from the master and sent to the slaves. Replicas can be added and removed from the system at runtime. The master role can be assigned dynamically, for example when the master replica fails. The current prototype does not support parallel working schedulers, yet it is not vulnerable to failures of the scheduler. If a Ganymed scheduler fails, it will immediately be replaced by a standby scheduler. The decision for a scheduler to be replaced by a backup has to be made by the *manager component*. The manager component, running on a dedicated machine, constantly monitors the system. The manager component is also responsible for reconfigurations. It is used, e.g., by the database administrator to add and remove replicas. Interaction with the manager component takes place through a graphical interface.

Client Interface: Clients connect to the scheduler through the Ganymed JDBC 3.0 database driver. The availability of such a standard database interface makes it straightforward to connect Java based application

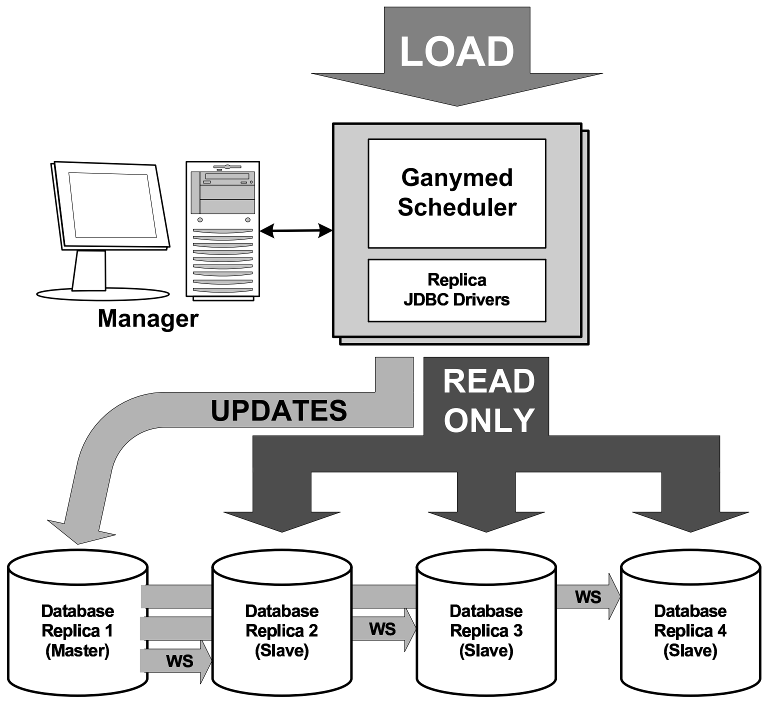


Figure 1: Ganymed Overview.

servers to Ganymed. The migration from a centralized database to a Ganymed environment is very simple, only the JDBC driver component in the application server has to be reconfigured, there is no change in the application code. Our driver also supports a certain level of fault tolerance. If a configured Ganymed scheduler is not reachable, the driver automatically tries to connect to an alternate scheduler.

Since the scheduler needs to know if a transaction is an update or read-only, the application code has to communicate this to the Ganymed JDBC driver. This mechanism is already included in the JDBC standard. Application code that wants to start a read-only transaction simply calls the *Connection.setReadonly()* method.

Writeset Extraction and Replica Support: On the replica side, Ganymed currently supports PostgreSQL and Oracle data-base engines. This allows to build heterogenous setups, where replicas run different database engines. Also, our approach is not limited to a specific operating system. As in the client side, the communication with the replicas is done through JDBC drivers. In a heterogenous configuration, the Ganymed scheduler has therefore to load for every different type of replica the corresponding JDBC driver. Since this can be done dynamically at run time, on startup the scheduler does not need to be aware of the type of replicas added at runtime.

Unfortunately, the JDBC interface has no support for writeset extraction, which is needed on the master replica. In the case of PostgreSQL, we implemented an extension of the database software. PostgreSQL is very flexible, it supports the loading of additional functionality during runtime. Our extension consists of a shared library written in C which holds the necessary logic to collect changes of update transactions in the database. Internally, the tracking of updates is done using triggers. To avoid another interface especially for the writeset handling, the extension was designed to be controlled over the normal JDBC interface. For instance, the extraction of a writeset can be performed with a ”*SELECT writeset()*” SQL query. The extracted writesets are table row based, they do not contain full disk blocks. This ensures that they can be applied on a replica which uses another low level disk block layout than the master replica.

In the case of Oracle, we implemented a similar extension, which is based on triggers and the JVM (Java

Virtual Machine) which is part of the Oracle database. As in the case of PostgreSQL, this extension is accessible through standard JDBC calls. Currently we are also implementing the writeset extraction feature for the IBM DB2 databases.

Implementation of RSI-GPC: Our current implementation of RSI-GPC is very strict, the scheduler always provides strong consistency. Loose consistency models are not supported in the current version, therefore read-only transactions will always see the latest snapshot of the database. The scheduler also makes a strict distinction between the master and the slave replicas. Even if there is free capacity on the master, read-only transactions are always assigned to a snapshot isolation based slave replica. This ensures that the master is not loaded by complex read-only transactions and that there is always enough capacity on the master for sudden bursts of updates. However, if there is no slave replica present, the scheduler is forced to assign all transactions to the master replica, acting as a relay.

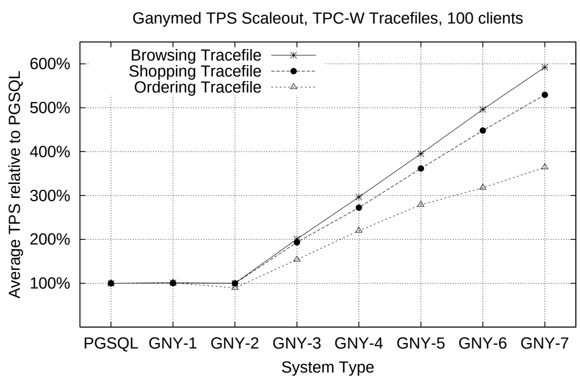
Read-only transactions are assigned to a valid replica according to the LPRF (*least pending requests first*) rule. Valid means in this context, that the replica must contain the latest produced writeset. If no such replica exists, the start of the transaction is delayed. Once a transaction is assigned to a replica, be it of type update or a read-only, this assignment will not change. Also, unlike other replication solutions, Ganymed does not require that transactions are submitted as a block, i.e., the entire transaction must be present for it to be scheduled. In Ganymed different SQL statements of the same transaction are progressively scheduled as they arrive, with the scheduler ensuring that the result is always consistent.

To achieve a consistent state between all replicas, the scheduler must make sure that writesets of update transactions get applied on all replicas in the same order. This already imposes a problem when writesets are generated on the master, since the scheduler must be sure about the correct commit order of transactions. Ganymed solves this problem by sending COMMIT operations to the master in a serialized fashion. The distribution of writesets is handled by having a FIFO update queue for every replica. There is also for every replica a thread in the scheduler software that applies constantly the contents of that queue to its assigned replica.

4 Behavior for Typical TWA Loads

To be able to verify the validity of our approach we have performed extensive experiments with Ganymed. To ease the interpretation of the results, a homogenous setup was chosen: all the used replicas used the same database product, namely PostgreSQL, and all participating machines had the same hardware configuration. All replicas were initialized with the data set from a database of a TPC-W installation (scale factor: 10.000 items, 288.000 customers). The TPC benchmark W (TPC-W) is a transactional web benchmark from the Transaction Processing Council [12]. TPC-W defines an internet commerce environment that resembles real world, business oriented, transactional web applications. The benchmark also defines different types of workloads which are intended to stress different components in such applications. The TPC-W workloads are as follows: primarily *shopping*, *browsing* and web-based *ordering*. The difference between the different workloads is the ratio of browse to buy: browsing consists of 95% read-only interactions, for shopping the ratio is 80% and for ordering the ratio is 50%. Shopping, being the primary workload, is considered the most representative one. To be able to perform repeatable experiments with Ganymed, we generated a tracefile for each of the three workloads using a TPC-W installation.

The system was set up in different configurations and loaded with those traces. On one hand, the achievable throughput and the resulting response times were measured and compared to a single PostgreSQL instance. To do this, a load generator was attached to the database (either the single instance database or the scheduler, depending on the experiment). During a measurement interval of 100 seconds, a trace was then fed into the system over 100 parallel client connections and at the same time average throughput and response times were



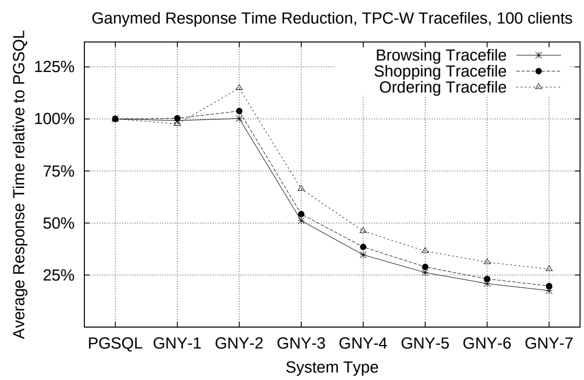


Figure 2: Ganymed Scaleout for TPC-W Tracefiles.

measured. On the other hand, the reaction of the system to failures was investigated. For this, the load generator was attached to the scheduler and then the change in throughput was monitored while disconnecting replicas.

Scalability: We measured the performance of the Ganymed scheduler in different configurations, from 1 up to 7 replicas. Including the single instance measurements, this results in a total of 8 experimental setups (called PGSQL and GNY-n, $1 \leq n \leq 7$). Figure 2 shows the results for the achieved throughput (transactions per second) and average transaction response times for the three TPC-W traces. The rate of aborted transactions was below 0.5 percent. As already noted, the TPC-W shopping workload is regarded as the most representative one. Clearly, a nearly linear scale-out in terms of throughput was achieved and response times were reduced. Obviously, in case of the ordering workload the system scales, but not linearly. This is due to the large amount of updates in this workload which have to handled by the master replica.

Fault Tolerance: As an example of the fault tolerance of Ganymed, we denote the results from an experiment were the master replica fails. The setup is a GNY-4 system, fed with a TPC-W shopping trace. During the measurements, a SIGKILL signal was used to stop the PostgreSQL database system software running on the master replica. The scheduler reacted by reassigning the master role to a different, still working slave replica. It is important to note that the reaction to failing replicas can be done by the scheduler without intervention from the manager console. Even with a failed or otherwise unavailable manager console the scheduler can still disable failed replicas and, if needed, move the master role autonomously. Figure 3 shows the resulting throughput histogram for this experiment. Transaction processing is normal until in second 45 the master replica stops working. The immediately move of the master role to a slave replica leaves a GNY-3 configuration with one master and two slave replicas. The failure of the master replica led to an abort of 2 update transactions, no other transactions were aborted during the experiment. The arrows in the graph show the change of the average transaction throughput per second.

5 Conclusions

Despite its simplicity, the Ganymed system based on RSI-GPC tries to avoid common drawbacks of existing solutions. For instance, in comparison to other middleware based approaches like [1, 2, 3], Ganymed does not involve any SQL statement parsing or table level locking. Due to the use of SI and the separation of updates and read-only transactions, conflicts can only happen at the master replica. Therefore, this task has not to be duplicated at the scheduler level and can be done efficiently by the master replica. SQL parsing and table level

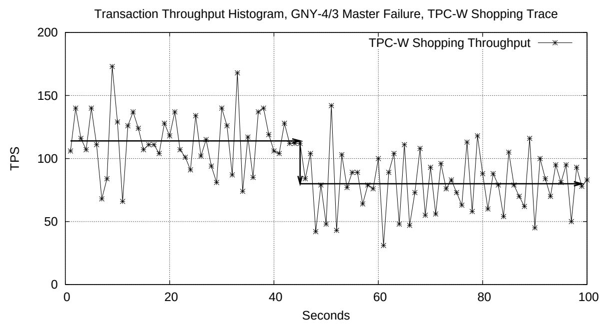


Figure 3: Ganymed Fault Tolerance.

locking at the scheduler both severely limit concurrency and prevent the use of certain features in the database replicas. Triggers, for example, cannot be used with such approaches: the scheduler cannot easily decide by looking at SQL statements if they fire triggers (and eventually update other tables), the only solution would be to lock even more conservatively. In the Ganymed approach, triggers do not impose a problem. They can be installed and changed on the master replica at any time and the scheduler does not have to be informed about their existence.

To circumvent these problems, some systems force the client to predeclare all the objects a transactions is going to access. However, this is very inconvenient and inflexible. It also complicates the adaptation of already existing client software. Solutions where the client has to send full transactions as a block impose similar, even worse problems.

Another problem of common middle schedulers solutions is the usage of write-all approaches on SQL statement level: updates are applied to replicas by broadcasting the client's SQL update statements. This can be very inefficient, since SQL processing work is duplicated at all replicas. In the case of complex update statements which update just a few rows, it is much more efficient to work with writesets. In the case of Ganymed, SQL update statements have only to be evaluated at the master; slave replicas just apply the resulting changes, leaving capacity for read-only transactions.

Systems that are distributed over the network and support caching at the edge level, like [4, 5, 6], can dramatically improve response times. Also, part of these systems are very flexible and can react to changes in the load and adapt the used cache structures. Unfortunately, in all cases full consistency has to be given up to achieve scalability.

The goal of Ganymed is to be as flexible as possible without having to give up full consistency or disallowing certain access patterns. Ganymed imposes no data organization, structuring of the load, or particular arrangements of the schema. Applications that want to make use of Ganymed do not have to be modified, the JDBC driver approach guarantees the generality of the interface that Ganymed offers. Thanks to the minimal infrastructure needed, Ganymed provides excellent scalability and reliability for typical transaction web applications. For typical loads, Ganymed scales almost linearly. Even if replicas fail, Ganymed is able to continue working with proper performance levels thanks to the simple mechanisms involved in recovery.

As part of future work, we are exploring the use of specialized indexes in the read-only replicas to speed up query processing. We are also studying the possibility of autonomic behavior in the creation of such indexes or even whole replicas. For instance, the manager console could adapt dynamically the replica setup as a result of inspecting the current load. Given the low overhead of the infrastructure, we can invest in such optimizations without worrying about the impact of the extra computations on performance. In the medium term, the implementation of complete autonomous behavior is an important goal.

References

[1] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*, 2003.

[2] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[3] Emmanuel Cecchet, Julie Marguerite, Mathieu Peltier, and Nicolas Modrzyk. C-JDBC: Clustered JDBC, http://c-jdbc.objectweb.org.

[4] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *Proceedings of the 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, 2003.

[5] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, 2003.

[6] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Transparent Mid-tier Database Caching in SQL Server. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 661–661. ACM Press, 2003.

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1–10, May 1995.

[8] Ralf Schenkel and Gerhard Weikum. Integrating Snapshot Isolation into Transactional Federation. In Opher Etzion and Peter Scheuermann, editors, *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*, volume 1901 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2000.

[9] Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7. Oracle White Paper, July 1995.

[10] PostgreSQL Global Development Group. PostgreSQL: The most advanced Open Source Database System in the World. http://www.postgresql.org.

[11] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *IEEE 22nd Int. Conf. on Distributed Computing Systems, ICDCS'02, Vienna, Austria*, pages 477–484, July 2002.

[12] The Transaction Processing Performance Council. TPC-W, a Transactional Web E-Commerce Benchmark. http://www.tpc.org/tpcw/.