Improving Scalability by Self-Archiving

Zhiwu Xie[1, 2], Jinyang Liu[3], Herbert Van de Sompel[4], Johann van Reenen[1], Ramiro Jordan[1]

[1]University of New Mexico
Albuquerque, NM 87131
{zxie,jreenen,rjordan}@unm.edu

[2]George Mason University
Fairfax, VA 22030
zxie2@gmu.edu

[3]Howard Hughes Medical Institute
Ashburn, VA 20147
liuj@janelia.hhmi.org

[4]Los Alamos National Laboratory
Los Alamos, NM 87544
herbertv@lanl.gov

ABSTRACT

The newer generation of web browsers supports the client-side database, making it possible to run web applications entirely in the web clients. Still, the server side database is indispensable as the central hub for exchanging persistent data between the web clients. Assuming this characterization, we propose a novel web application framework in which the server archives its database states at predefined intervals then makes them available on the web. The clients then use these archives to synchronize their local databases with the server. Although the main purpose is to reduce the database scalability bottleneck, this approach also facilitates the long-term preservation and can be used for time traveling. We discuss the consistency and coherency properties provided by this framework, as well as the tradeoffs imposed.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Online Information Services – *Web-based services;* H.2.4 [**Database Management**]: Systems – *Concurrency, Distributed databases, Transaction processing.*

General Terms

Design, Performance.

Keywords

Database scalability, replication control, freshness, archive.

1. INTRODUCTION

Web applications may preserve their own dynamic resources for long-term use by timely archiving the historic data versions from which the transient representations are generated. Unless it is part of the core functionalities, however, most web applications do not self-archive. Even if they do, the archived data are rarely made available on the web because they are not typically considered mission-critical. We therefore attempt to provide a more pertinent incentive for self-archiving. We propose a novel web framework in which the archived data are tightly integrated into the core web functionalities in order to improve scalability. Self-archiving of this kind can therefore claim the mission-critical benefits to the organizations, such as more robust service under load surges and deferring expensive hardware upgrades.

In this paper we present the theoretical aspects of this framework. Experimental verification and quantitative measurements will be reported in our follow-up work.

JCDL'11, June 13–17, 2011, Ottawa, Canada.

2. DATABASE SCALABILITY

For data driven dynamic web applications, the backend database is known to be "the" scalability bottleneck. Caching [1] and database replication [2] are commonly used to alleviate the pain but each comes with its own set of challenges.

Caching performs best for static contents. Without pre-scheduled expiration time, cached dynamic contents such as database query results must be validated before reusing. Such validation may require extra database queries, defeating the purpose of caching. The "NoSQL" approach avoids this problem by intentionally sacrificing the strong database consistency, a price often regarded as "too steep to pay". Furthermore, the total size of the materialized views can be many orders of magnitude larger than the size of the database, leading to lower reuse rates and the deterioration of the cache efficiency.

When the database is replicated, the replicas must be timely and consistently updated, a known complication by its own right [3]. This is particularly hard when a large number of replicas are located geographically far from each other. Lazy replication scales better than eager protocols such as two-phase commit, because it allows the updates to propagate after being committed. But it is also more difficult to ensure consistency. The strongest form of consistency, known as 1-copy serializability [4], guarantees the execution of the transactions on replicated databases is equal to that on a single copy database.

Much research and development work has been done on both fronts but they typically assume the database code may be modified [5, 6], a scheduler or middleware layer may be added [7, 8, 9], or a proprietary high-speed link or mass storage infrastructure is at hand. To date few options are available for web applications to consistently scale well using the existing infrastructure and the commodity technologies.

A recent web browser feature shed a new light. Today, all major web browsers support in-browser local databases and provide standard APIs to access them [10, 11]. This forms the foundation for a lazy-master style replication structure that distributes part of the database query load to the end users who directly benefit from the web applications. Even better, it's free.

A few limitations exist in this structure that make many earlier lazy-master replication control protocols unsuitable, mostly due to the web architecture [12, 13]. First, all communications are pull-based, initiated by the web clients towards the server only. The

master database on the server does not track the client-side replicas and their states, nor does it initiate replication requests to the clients. The clients are not aware of each other's existence, lest directly talk to each other. Second, the replicas are synchronized over the general web operations, via standard HTTP, and by "materializing" the master database states or writesets into payloads then re-applying them to the client-side replicas. Despite these, a cached copy of the above payload may be reused for speeding up the replications of the other web clients, although the validation problem described above persists.

Our goal, therefore, is to build a scalable as well as 1-copy serializable web application framework on top of this replication structure. The replication control protocol is fully implemented in Javascript in the web application layer, downloaded then executed in the web browsers and fully independent from the master database and the network.

3. AN ARCHIVE BASED SOLUTION

3.1 Trading Data Freshness for Scalability

Ideally all database replicas should immediately synchronize with any update that occurs on the master, and queries processed on any replica would indistinguishably result in the latest available information. However to achieve this goal the database will run a deadlock probability of at least the third power of the transaction size [3]. To get better scalability we may relax the data freshness in a controlled way [14, 15, 16, 17]. As a thought experiment, imagine a trivial case where a multiversion database that always responds to queries from its initial state no matter how its actual state is changed by the committed updates. The query results received by the users would always be the most stale possible, and the web application would be useless. But these query results are valid forever, may be infinitely cached, replicated, and reused, therefore enjoy the similar cache efficiency and replication scalability as static files.

Most real world web applications assume the opposite extreme, in which the database always responds to queries with the latest database state. The usual justification for doing so is the user expectations but it's crucial to acknowledge that existing combinations of the systems do not guarantee to serve the freshest data from the user's point of view. An ACID database recognizes as legitimate any concurrent executions equivalent to any serial execution of the same set of transactions. It is therefore possible for a subsequent query to not see the earlier updates. Compounding this with the network latency the mismatch can go even further. For example, assuming traditional non-replicated three-tier web architecture, when the initially fresh response leaves the database server and reaches the remote client across the WAN latency, the database state could have already changed and the response would have become stale from the user's point of view. Therefore, it seems unnecessary to insist the database alone serving data fresher than the order of the WAN latency. In reality, staleness is always expected and well understood by the users of the heavy loaded web applications such as online auction or stock exchange sites. The users intuitively know the prices shown on the screen may have changed milliseconds ago, and no matter how fast they react, their bids may be rejected because someone else's bid commits first as seen by the database.

Many researches on the relaxed freshness assume we want to relax the consistency too. Not in our case. Serving stale data does not necessarily contradict with the database consistency and coherency. For example, all multiversion concurrency control methods produce stale query results at some point, but many of them are serializable. Even snapshot isolation can be serializable under certain conditions [6]. Indeed, if all replicas serve the same stale but consistent data to the queries, a replicated system can be 1-copy serializable too.

3.2 Replication Control and Its Correctness

There are many ways to relax the freshness constraint [18]. We propose a novel relaxation method based on a predefined absolute timestamp. The way our method works is analogous to the motion pictures. When shooting motion pictures, no matter how fast the object moves, the video camera takes snapshots at its own predefined frequencies. The states in between are not recorded, and the snapshots taken are the candidates for long-term preservation. Given the archived film, a video player understands how the frames are timed and assembled, and uses this knowledge to replay the snapshots to the clients.

More formally, for any given time $t > t_0$, where t_0 is the initial database time, there exists one and only one time interval $[t_i, t_i+\Delta t_i)$, such that for $t \in [t_i, t_i+\Delta t_i)$ and $\Delta t_i=O(\Delta t_L)$, where Δt_L is the network latency, any query arrives at the database at t will be responded with the same result as if the query is executed at t_i. A simplified example would be having time intervals of every 5 seconds starting from 2010-02-01T01:00:00Z, such that we limit the data staleness to be at most 5 seconds.

We require the time intervals defined a priori. At any given time after a web client initializes itself, it should already know the corresponding time intervals without having to contact the server again to find out. We also require the time intervals defined in absolute time and all the web clients reasonably synchronized to a NTP server to guarantee limited time skews among the replicas and the server. This is important because it eliminates the expensive web and database operations to just determine to which state a database replica should be synchronized.

```
Upon: submit of a read-only transaction T to client at time t
1:  assign T the timestamp t_i, the start of its time interval
2:  if local database is not synced to the state at t_i:
3:    request from the master the writesets up to [t_i-1, t_i-1+Δt_i-1)
4:    sync local database with the master to its state at t_i
5:  execute T at the local database
6:  return result

Upon: submit of an update transaction T to client
7:  forward T to the master

Upon: submit of an update transaction T to master
8:  atomically request necessary shared and exclusive locks for T
9:  wait until all locks are granted
10: execute T at the master database, record the commit time t
11: release locks of T
12: calculate n such that t ∈ [t_n, t_n+Δt_n)
13: read m where the writeset for time interval [t_m, t_m+Δt_m) was
last archived. If no writeset has ever been archived, m = -1
14: if m < (n-1):
15:   for each i ∈ (m, n):
16:     archive the writeset for time interval [t_i, t_i+Δt_i)
17: return ok

Upon: submit of a request to master for writeset for time interval
[t_n, t_n+Δt_n)
18: if n = 0:
19:   return the initial database state
20: read m where the writeset for time interval [t_m, t_m+Δt_m) was
last archived. If no writeset has ever been archived, m = 0
21: if m < n:
22:   for each i ∈ (m, n]:
23:     archive the writeset for time interval [t_i, t_i+Δt_i)
24: return writeset for time interval [t_n, t_n+Δt_n)
```

Figure 1. Archive based replication control algorithm.

We now describe the replication control algorithm. The pseudo code is depicted in Figure 1. In our algorithm, all updates are sent to the database master at the server and executed using the strict two-phase locking (2PL) protocol. The queries are executed at the

clients based on their explicitly declared timestamps, and we require the clients to synchronize with the master database version at that particular timestamp before the execution. This is similar to the multiversion mixed method described in [4] or the snapshot isolation protocols [19, 6] with one important distinction. In our method the timestamps are chosen independently from the database state and the timing of the queries. The technique we use is inspired by that devised in the archived feed protocol [20], which converts the sliding windows in the algorithm into fixed time intervals, therefore the name "archive based replication control".

Conceptually the time intervals we introduced enforce a new transactional state to the database. Traditionally we assume once an update is committed its changes are immediately visible to all other active transactions. We revoke this assumption and define a "QUERY VISIBLE" state after "COMMITTED", as shown in Figure 2. The changes made by an update are still immediately visible to other updates once committed, but are visible to active queries only when they mature through the predefined time interval.

As an illustration, if we use the mixed protocol or snapshot isolation, Query 1 in Figure 2 would be able to see Update 2 because it starts after Update 2 commits. But this would require a new version or snapshot being created for each committed update, and each of them may need to be individually propagated to all the replicas. In our protocol, Update 2 is invisible to Query 1, because it becomes visible to queries after the latter starts. Query 2, on the other hand, can see both updates. Since the time interval does not depend on the transaction size, when the service becomes busier, even though there may be more update transactions become visible at once at each timestamp, the number of snapshots to be propagated per time unit does not increase proportionally. On the other hand, if both updates in Figure 2 write to the same data item, the data written by Update 2 is lost in our protocol because no query ever sees it. We simply assume these updates represent the intermediate states which, despite committed, the clients don't care to know.

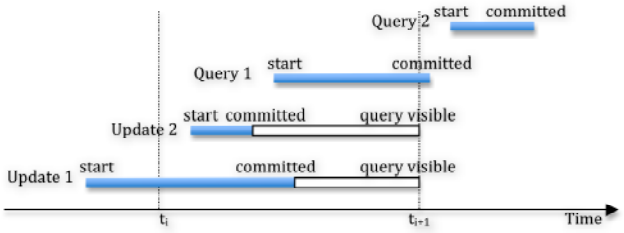

Figure 2. Update visibility and serial execution.

Since we did not relax the irrevocability aspect of the COMMITTED state, the ACID property of the 2PL still holds.

The correctness of this protocol is easy to prove. Due to the timestamps and the master-replica differentiation between the update-query transactions, executing queries on the replicas introduces no data contention, and the network latency is masked by the timestamps. We can easily see that any distributed execution is equal to the execution of the same transactions on a single copy master database. As for the single copy execution equivalence, since the updates are executed in strict 2PL, there exists a serial execution of these updates, and they are serialized by the sequence of their commit time. By definition all queries can be moved to the start of their time intervals, and their relative ordering does not matter because there's no update transaction in between. Therefore the single copy execution equivalence is serializable, and the replication control protocol is 1-copy serializable. As an example, the transactions shown in Figure 2 can be serialized in this order: Query 1 $<_t$ Update 2 $<_t$ Update 1 $<_t$ Query2.

3.3 What to Archive

An archive stores the frozen, static snapshots of the changing web resources, each associated with an unambiguous, historic point of time or period. A historic database snapshot produced in section 3.2 may be archived and made a web resource by its own right, also called a Memento [21]. By definition the representation of a Memento is time invariant, therefore its cached copy never needs validation. Due to the timestamp nature, most web clients will need the same Memento in approximately the same period of time, providing good temporal locality for caching.

However, archiving full database states may be expensive in terms of storage and bandwidth usage. In many cases we only need to know the data changes, or the writesets, since the last archive [22]. But if all archives are in the form of writeset for just a single time interval we'd need to download all the archives up till the initial state before we can reconstruct a Memento from scratch. Inspired by [20], we use an archive format that contains not only the writeset of the current time interval, but also pointers to larger combinations of these archives. For example, we may archive the sum of the writesets for two consecutive time intervals in every two time intervals, then do the same for four consecutive time intervals and so on, then embed their URIs in the later archives. Now the clients must know how to arrive at a Memento. In the Memento architecture, this is achieved by leveraging a TimeGate resource. A TimeGate [21] supports HTTP content negotiation in the datetime dimension to obtain access to prior versions of a resource. In our architecture, the TimeGate functionality is implemented in Javascript, downloaded to the web clients, and executed in a distributed manner. This Javascript code parses the downloaded archive, calculates the most efficient combination of archived resources based on the current database state and the current time, then downloads these archives and applies their writesets in the correct order to rebuild the Memento.

It's easy to prove that if we archive the database states in this way, the total archive size is linear to the total database size, and the number of downloads necessary to reconstruct the full Memento from scratch is no more than the order of the logarithm of the number of time intervals. This provides a better balance between the cache efficiency, network usage, and latency.

3.4 Partial Replication and Privacy

Privacy poses a major concern in replication control and caching [23, 24]. No sensible web application would allow the important personal information be cached or replicated outside of its direct control, let alone at the client-side of a random web user.

Partial replication may be used to mitigate this problem, in which the tables containing sensitive data are not replicated. However many queries need data across sensitive and insensitive tables, such as the order details page that shows the customer's billing address as well as the book title and the author information. If this type of transactions must all be executed on the server, we'd have more read locks that interfere with the update transactions, leading to scalability issues.

The archive-based solution described in this paper provides an easy and straightforward solution. We may break up these queries into multiple sub-queries, some concern the sensitive data, others not, then execute them separately on different places. Under normal circumstances synthesizing these query results together would lead to inconsistent state, because the database state may have changed in between the execution of the sub-queries. But in our case this is not a problem because both the original query and all its sub-queries carry the same timestamp that points to an archived, frozen database state.

4. CONCLUSION

In this paper we present the theoretical aspects of an archive based database replication control protocol for web applications. It uses the client-side databases as the replicas of the master database on the server. This protocol ensures 1-copy serializability and improves scalability. It also provides an easy-to-use method to partition privacy related queries between the master and the replicas. We also provide a novel archiving method based on which time travel can be efficiently implemented.

We will report the experimental verification and quantitative measurements in the follow-up work.

5. REFERENCES

[1] Labrinidis, A., Luo, Q., Xu, J. and Xue, W. 2009. Caching and Materialization for Web Databases. *Foundations & Trends in Databases.* 2, 3 (Jul. 2009), 169-266.

[2] Kemme, B., Peris, R.J. and Patino-Martinez, M. 2010. Database Replication. *Synthesis Lectures on Data Management.* 2, 1 (2010), 1-153.

[3] Gray, J., Helland, P., O'Neil, P. and Shasha, D. 1996. The dangers of replication and a solution. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (Montreal, Quebec, Canada, 1996), 173-182.

[4] Bernstein, P.A., Hadzilacos, V. and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publishing Company.

[5] Bettina, K. and Alonso, G. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. *Proceedings of the 26th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2000), 134–143.

[6] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P. and Shasha, D. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492-528.

[7] Amza, C., Cox, A.L. and Zwaenepoel, W. 2003. Conflict-aware scheduling for dynamic content applications. *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Berkeley, CA, USA, 2003), 6–6.

[8] Plattner, C. and Alonso, G. 2004. Ganymed: scalable replication for transactional web applications. *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware* (Toronto, Canada, 2004), 155-174.

[9] Thomson, A. and Abadi, D. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* 3, 1 (2010).

[10] Hickson, I. Web SQL Database. http://www.w3.org/TR/webdatabase/

[11] Mehta, N., Sicking, J., Graff, E. and Popescu, A. Indexed Database API. http://www.w3.org/TR/IndexedDB/

[12] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. 1999. Hypertext transfer protocol–HTTP/1.1. RFC 2616, June 1999.

[13] Fielding, R.T. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* University of California.

[14] Alonso, R., Barbara, D. and Garcia-Molina, H. 1990. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems.* 15, 3 (1990), 359-384.

[15] Gallersdörfer, R. and Nicola, M. 1995. Improving Performance in Replicated Databases through Relaxed Coherency. *Proceedings of the 21th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1995), 445–456.

[16] Röhm, U., Böhm, K., Schek, H. and Schuldt, H. 2002. FAS: a freshness-sensitive coordination middleware for a cluster of OLAP components. *Proceedings of the 28th international conference on Very Large Data Bases* (2002), 754–765.

[17] Guo, H., Larson, P. and Ramakrishnan, R. 2005. Caching with "good enough" currency, consistency, and completeness. *Proceedings of the 31st international conference on Very large data bases* (Trondheim, Norway, 2005), 457-468.

[18] Bouzeghoub, M. 2004. A framework for analysis of data freshness. *Proceedings of the 2004 international workshop on Information quality in informational systems - IQIS '04* (Paris, France, 2004), 59.

[19] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* (New York, NY, USA, May 1995), 1–10.

[20] Nottingham, Mark. Feed Paging and Archiving. http://www.ietf.org/rfc/rfc5005

[21] Van de Sompel, H., Nelson, M. and Sanderson, R. HTTP framework for time-based access to resource states -- Memento. http://tools.ietf.org/html/draft-vandesompel-memento-00

[22] Benson, E., Marcus, A., Karger, D. and Madden, S. 2010. Sync Kit: A persistent client-side database caching toolkit for data intensive websites. *19th International World Wide Web Conference*, WWW2010, April 26, 2010 - April 30, 2010 (Raleigh, NC, United states, 2010), 121-130.

[23] Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L. and Zheng, X. 2009. Building secure web applications with automatic partitioning. *Commun. ACM.* 52, 2 (2009), 79-87.

[24] Manjhi, A. 2008. *Increasing the scalability of dynamic web applications.* Carnegie Mellon University.