Poor Man's Social Network: Consistently Trade Freshness For Scalability

Zhiwu Xie
Virginia Polytechnic Institute and State University
Jinyang Liu
Howard Hughes Medical Institute
Herbert Van de Sompel
Los Alamos National Laboratory
Johann van Reenen and Ramiro Jordan
University of New Mexico

Abstract

Typical social networking functionalities such as feed following are known to be hard to scale. Different from the popular approach that sacrifices consistency for scalability, in this paper we describe, implement, and evaluate a method that can simultaneously achieve scalability and consistency in feed following applications built on shared-nothing distributed systems. Timing and client-side processing are the keys to this approach. Assuming global time is available at all the clients and servers, the distributed servers publish a pre-agreed upon schedule based on which the continuously committed updates are periodically released for read. This opens up opportunities for caching and client-side processing, and leads to scalability improvements. This approach trades freshness for scalability.

Following this approach, we build a twitter-style feed following application and evaluate it on a following network with about 200,000 users under synthetic workloads. The resulting system exhibits linear scalability in our experiment. With 6 low-end cloud instances costing a total of no more than $1.2 per hour, we recorded a peak timeline query rate at about 10 million requests per day, under a fixed update rate of 1.6 million new tweets per day. The maximum staleness of the responses is 5 seconds. The performance achieved sufficiently verifies the feasibility of this approach, and provides an alternative to build small to medium size social networking applications on the cheap.

1. Introduction

Scalability has emerged as a significant challenge for the social networking web applications. If built with the traditional web application framework, even a small social network can be easily strained by a modest level of user interaction. The performance bottleneck usually locates at the back-end persistent data store, which is typically a relational database. Following the examples of Twitter and Facebook, many social networking web sites start to migrate their relational databases to various key-value stores, collectively branded as being "NoSQL". This approach indeed can scale to a larger workload, but always at the expense of a deliberate void of the consistency guarantee.

What does consistency mean and entail in this context? Since many NoSQL advocates cite the Brewer's Conjecture [8], also known as the CAP theorem, as the theoretical foundation to justify this trade-off, we naturally adopt the consistency definition used in its formal proof [18]. Used elsewhere, this type of consistency is also known as Atomicity [23], Linearizability [20], or One-copy Serializability (1SR) [6]. This is different from the consistency as referred in the ACID properties of the database. To avoid further ambiguity, in this paper we regularly use the abbreviation 1SR to denote such consistency, and unless noted otherwise, consistency always refers to 1SR in this paper.

1SR provides the clients of a distributed system with an equivalent single processor view that allows them to reason the system behavior regardless of how many distributed servers are used to run the service, how geographically far apart they are from each other and from the clients, and how they are synchronized. Without 1SR, the distributed system may exhibit odd behaviors that confuse the users. We therefore are interested in exploring the possibility of scaling the social networking functionalities, especially the feed following applications, without violating 1SR.

Given the formal CAP theorem proof this may seem impossible. But the proof itself is strictly preconditioned on the asynchronous network model, where the only way to coordinate the distributed nodes is to pass messages across the network. Practical distributed systems usually have more tools in hand, and one of the tools is a reasonably synchronized and approximated global time. Indeed, the authors of the CAP theorem

proof used the second half of their paper to show that under a partially synchronous, or timing-based, distributed model [24], where global time is assumed to be available, CAP may indeed be simultaneously achievable most of the time, although in return we may have to give up some freshness, but not necessarily the latency.

Unfortunately, this aspect of the CAP has not attracted sufficient attention from the industry nor from academic researchers. As early as 2008, Roy Fielding proposed a RESTful approach [15], which is distinctively timing-based, for the known hard-to-scale feed following problem. But we are not aware of many real-world social networking web applications that are built this way, and to the best of our knowledge, no empirical or experimental data are publicly available to verify its scalability properties.

In this paper we take Roy Fielding's proposal as a starting point, extrapolate it to shared-nothing distributed systems, and fine-tune its timing method for replication control. We then provide a formal description of the algorithm, prove its consistency property, and analyze its trade-offs. In order to gain insights on how much freshness we must give up to gain the level of scalability currently provided by the NoSQL approach, we build a system and test its performance with the workloads similar to that used in a Yahoo! PNUTS based Twitter-like feed following experiment [27]. The server side of the system is fully implemented, but instead of writing and delivering client-side code to real browsers, we implement an emulated browser on the client side to facilitate performance testing. In our experiment, we set a fixed staleness limit of 5 seconds and an update rate of 19 new tweets per second. Due to the limitation of the test facility we used, we were not able to generate a query workload exceeding 40% of the PNUTS experiment. But within this limitation our system exhibits linear scalability, up to 6 servers.

The main contributions of this paper include:

- A formal description of a timing-based replication control algorithm for the feed following problem
- A proof of its consistency property
- A working implementation built with lower performing commodity virtual machines in the cloud
- An experiment to thoroughly test its performance and trade-offs
- A working example that demonstrates how freshness can be exchanged for better scalability

With our approach, social networking applications are able to scale linearly to fairly high workloads with low-end machines. In our implementation we take advantage of the relatively stable and familiar commodity web server hardware and software stacks, which over the years have gained wide adoption and the prices have dropped significantly. No additional expertise and training are required to operate these stacks, potentially saving the operating costs as well. We therefore dub this approach "the poor man's social network." Nevertheless, there is no real obstacle to apply the timing-based approach to various NoSQL data stores. In our approach the relational databases are used mostly as simple key-value stores. We chose PostgreSQL in particular only to take advantage of its built-in triggers, procedural languages, and the transactional features not readily available from many key-value stores. But we anticipate that the same idea may be used by key-value stores as a base to develop more consistency features in addition to the eventual consistency.

This paper consists of the following sections: we first introduce the feed following problem and argue for a different tradeoff strategy to the popular method. We then provide a formal algorithm to implement this alternative in shared-nothing distributed systems and prove its correctness. After describing our implementation and experimental configurations, we present the results and the related work, then conclude the paper with discussions and future work.

2. Feed Following

2.1. Problem Statement

Feed following is the type of social networking functionality that layers on top of a following network consisting of large numbers of feed consumers and feed producers. Each feed consumer follows a usually large and distinctive group of feed producers, and each producer independently produces event items over the time. Now each of the consumers wants to query the n most recent event items produced by all the producers this particular consumer follows. Silberstein et al. give a more formal definition of the problem [28].

Twitter's timeline application is a typical feed following problem, where each event item is called a tweet. Many other social networking features may be modeled as variations of the feed following, and the "n most recent" predicate may also have many other flavors. But the common theme is that each feed following query can be quite personalized and distinctive from the others such that the query results for one consumer are of little or no use to another for the purpose of directly reusing the results to reduce the overall query load.

Moreover, even the newly produced query results may quickly become outdated for the same consumer. In order to provide the freshest possible response, a consumer's feed query result must be invalidated as soon as any of the followed producers posts a new event, and each producer's new event must also invalidate all the current feed following query results for each of the consumers that follow this producer. The large number of consumers also make it very expensive to maintain the materialized views for each of them. It is easy to see why the traditional scalability tools such as data partition, query and web caching, materialized view, and replication perform poorly in this type of problem. Silberstein et al. provide a more thorough analysis on the difficulties to scale the feed following problem [27].

2.2. Trade Consistency For Scalability

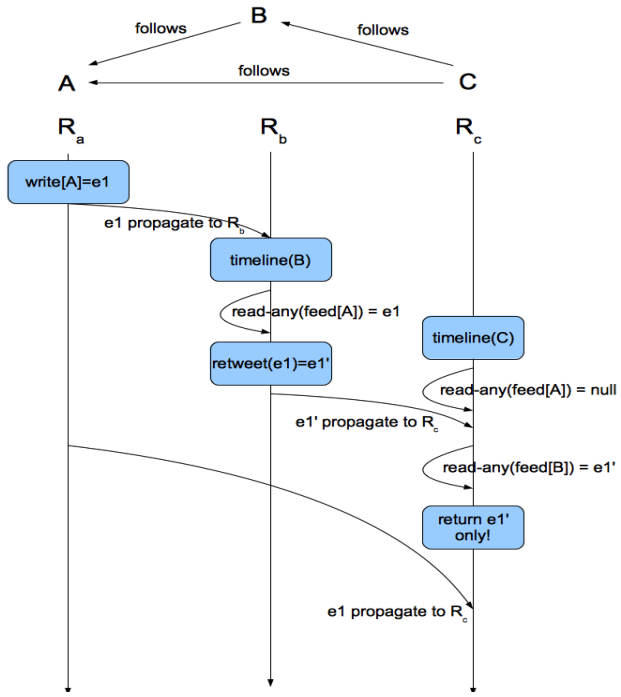
Popular NoSQL style applications are built under the assumption that consistency should be sacrificed to gain better scalability. To relax 1SR for feed following in a shared nothing, fully or partially replicated distributed environment, we may declare an event update successful as soon as one of the replicas commits it locally and before this update finishes propagating to most or all the other replicas. By eliminating the consistency locks, the replicas become more independent and can work in a more paralleled manner. But as a result, the follower's view becomes rather unpredictable. Even if a producer receives the confirmation of a successfully committed update, there is no guarantee when her followers can see this event. Some may see it shortly after, some others may need to wait for an extended period of time before this new event shows up, and even those who have already seen it once may not see it in the subsequent queries. We want to emphasize that waiting to see the most recent updates is not the real issue here. The real problem is the unpredictable nature of the wait.

Despite this, the consensus among the industry is that the users should be able to tolerate some lost events as long as they eventually show up. After all, unless the feed producers and the consumers are actively tracking and comparing their timelines, the temporarily lost events are not particularly noticeable. Moreover, many feed following applications such as Twitter do not allow editing; therefore eliminate the needs to reconcile the conflicting updates, as normally seen in an optimistic approach like this. However the inconsistency becomes apparent when the following network starts to change, e.g., a consumer decides to stop following a producer. If this relationship update is to be committed the same way as above, then it is possible for the consumer to continue receiving the event items from this producer even if this consumer has been notified of the successful unfollowing, simply because the unfollow update has not been propagated to the replica that processes the feed following query.

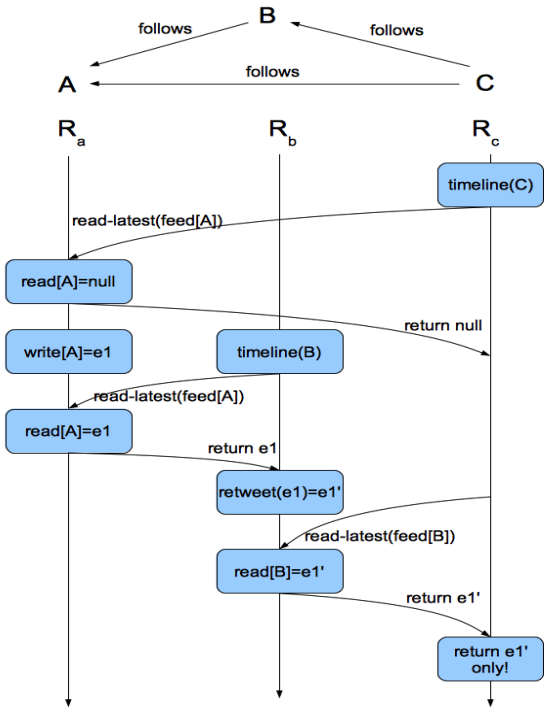
One approach to mitigate this problem is to slightly tighten up the eventual consistency model described above, to the "per-record timeline consistency", as exemplified by Yahoo!'s PNUTS [28]. This approach assigns a master replica for each record, then allows the application developer to specify what type of query to use: i.e., Read-any, Read-critical, or Read-latest. An update is not successful unless it is committed to the master copy, and a Read-latest query will always validate against that record's master copy to retrieve the most recently committed change, although a Read-any query will return any locally available data regardless of their validity. The assumption is that for those critical and consistency sensitive queries we should use Read-latest, which, although more expensive than the other two options, can at least provide some record-level consistency guarantees.

Nonetheless, this approach cannot prevent the inconsistency exposed by the retweets. A retweet is a standalone new tweet produced by a user who follows the original tweet producer. The retweet either includes the original tweet content by value or links to the original by reference. By logic a retweet can only commit after the original tweet is committed, because there exists a conflicting feed query between these two updates. It is therefore rather confusing for a feed consumer to observe only the retweet but not the original if this consumer follows both the original producer and the retweeter, which is fairly common in closely knit social groups. This arouses suspicion of either voluntary retraction or censorship, although in fact is merely a symptom of the distributed inconsistency.

As illustrated in figure 1, the retweet inconsistency may still happen in systems like PNUTS using either the Read-any or the Read-latest, as long as 1SR is not guaranteed. In the figure we assume Replica Ra is the master copy for all tweets produced by user A, et al. When using Read-any to retrieve the feeds from Replica Rc, if the retweet from Rb propagates to Rc faster than the original tweet from Ra, we may observe the inconsistency at Rc. The probability of inconsistency may be lower if we use Read-latest, in which case all the feed queries must be routed to their master replicas. But in between these large number of independent and usually



(a) Read-any



(b) Read-latest

Figure 1. Retweet Inconsistency

remote queries to the master copy, the logical sequence between the original tweet and its retweets may still be reversed, as shown in Figure 1(b). Furthermore, using Read-latest on all feed queries carries a steep performance penalty, because we lose the benefits of replication and caching, leaving data partition the only performance booster.

The Read-critical query, which "returns a version of the record that is strictly newer than, or the same as the required version" [28], provides little help in the above scenario. This is because in the per-record timeline consistency, the record's version is specified locally by its master copy. It is not a global version, therefore affords no meaningfully comparison between the versions of two different records, e.g., a tweet and its retweet. It is tempting to devise a sophisticated global versioning scheme to determine the sequence of all the updates, but this already implies 1SR. From the CAP theorem proof we already know that if a distributed system relies solely on the message passing to implement 1SR, then it is hard bounded by the CAP compromise and cannot scale well reliably. A more promising approach would be to explore beyond the asynchronous network model and more specifically, to exploit global time, which does not depend on the message passing exclusively. In the next section we discuss how this approach trades freshness for scalability.

2.3. Trade Freshness For Scalability

Freshness is oftentimes unnecessarily entangled with 1SR. For example, Vogels defines the strong consistency as being always guaranteeing the freshness and 1SR. But in its first degree of relaxation, the weak consistency allows a period of "inconsistency window" during which an update is not guaranteed to be always available to all queries [30], "not always" being the keyword. During the "inconsistency window" such weak consistency fails not only the freshness test but also the 1SR test. Such a categorization overlooks an intermediate level of distributed database system behavior which guarantees to return the 1SR responses, although they may be stale. Such systems indeed exist, e.g., a log-shipping based master-slave replication system where all the updates are processed at the master but all the queries go to the slave. The query results always lag behind the freshest state at the master, but the system is nevertheless 1SR.

We further argue that absolute freshness is not even worth pursuing in a web based system, because even the freshest query results still need to be transported

across the web to the clients, yet the web latency is not negligible. When the clients receive the results, they may have already turned stale, therefore from the holistic system view it seems rather unnecessary to guarantee the absolute freshness within the boundary of the database servers. Indeed, users of the web applications intuitively feel the latency and understand its effects. When using high-volume transactional systems such as the online bidding or stock exchange web applications, the users are acutely aware that the quote prices shown on the screens are not real-time but with delays built in. Moreover, not every system demands high levels of freshness anyway. Most social networking applications are not meant to be real-time point-to-point messaging systems, therefore some delays is tolerable and even expected.

We also note the differences between staleness and latency. Latency characterizes the speed of the request-response process, while staleness characterizes the recency quality of the data carried by the response. From the end user's point of view, latency always adds further staleness to the response, but not vice versa. Web users with short attention spans have fairly low tolerance for unresponsive web services, but not necessarily for staleness. Given a choice, faster responses carrying slightly more stale information should be much preferred than the opposite.

We now explain why freshness may be traded for scalability. We draw an analogy between this tradeoff and the mass transit system. When driving our own cars, we can freely choose the departure times and the destinations. But when using the mass transit systems, we must time our activities according to the published schedules, travel only to the vicinities of the bus stops, and make transfers between different transit lines by ourselves. Bus riders lose the flexibility to travel at-will, but gain overall efficiency and economic benefits by sharing resources. Such benefits are especially significant in metropolitan areas where not only the opportunity for sharing is higher but also the transportation resources are under much heavier loads and are much more congested.

Caching is the web's way to share resources. The web is built with the caching facilities at its core to address the scalability issues. But as explained in section 2.1, the current way of building feed following applications is not attuned for taking full advantage of the web caches. This is because such a system is built to accommodate the private-car style of usage, striving to provide the personalized response accurately and consistently at the time when the system executes that particular request. Caching is less effective because the queries are not only highly personalized, but also extremely ephemeral.

A mass transit style of feed-following system may improve the situation on two fronts. First, it may address the ephemeral issue by only executing queries with accuracy and consistency guarantee by a pre-agreed upon schedule, e.g., every 5 seconds. In a mass transit system all the passengers arriving at the station before the scheduled departure time must wait for the next bus. By the same logic, if enhanced with this improvement, all queries submitted to the servers between 1:05:30PM and 1:05:35PM will be immediately responded, but with the results that are accurate and consistent only as of 1:05:30PM. Conceptually this enhancement allows the queries received within this period all be executed against the same database snapshot taken at 1:05:30PM rather than against a moving target. We have built in no more than 5 seconds' staleness in all responses, yet the latencies are not necessarily higher. Due to effective caching and reusing, this approach may even significantly lower the response latency. However, the system works differently on updates. For example, if an update is received at 1:05:33PM, it will be committed as soon as the system permits, but the committed result will not be available for queries until the next scheduled time point, e.g., 1:05:35PM.

On the other hand, much like the mass transit system that will not board and drop off riders at any location, the server may also decline to execute personalized feed following queries. Instead, it may analyze and reorganize these queries, break them down into multiple steps, and only execute the commonly shared queries on the server. In case of the feed following, one exemplary common query useful for all users is the "time map" query, which tells us which feed producers have created new events during the past scheduled intervals. With such information at hand, the feed followers themselves can combine and match to produce their own personalized event lists on the client side. This is analogous to bus riders making transfers by themselves. Note that such an approach is only feasible when the queries are against the same database snapshot. Querying against changing database states cannot be reorganized correctly in this manner. In other words, this enhancement is preconditioned on the prior one.

In our approach a large portion of the processing is therefore offloaded to the clients, shifting the system from a thin client system to a fat client system. This is quite different from the NoSQL approach where the distributed servers still attempt to process all query

loads from the start to the end albeit abandoning the 1SR guarantee. In the next section we will formalize the algorithm and prove its correctness with respect to 1SR.

3. Replication Control and Its Correctness

We adopt the lazy-master style partial replication strategy for this distributed system. In particular, a conceptual centralized master database is naively and horizontally partitioned into multiple smaller physical database servers based on the producer id. A new tweet is routed to its own partition server according to the producer id and then committed there. Each partition server maintains a "time map". This is a materialized view that documents which producers allocated to this server have tweeted in the current scheduled interval. This view must be synchronously maintained within the same atomic update transaction boundary for a new tweet. In practice we also maintain multiple combined views that cover larger granularity of the time intervals.

A client, upon receiving a timeline query, first checks global time and then determines by itself the most recently scheduled release time. Imagine we have taken a conceptual full database snapshot at this particular release time. This is the database state against which this particular timeline query needs to be executed. The client then must make sure it has synchronized its local partial data with this snapshot before executing the timeline queries. Note the web architecture mandates that a server cannot initiate connection to the clients. This determines the lazy nature of our replication strategy. That is, an update is committed at the master copy but not atomically propagated to all the caches and the clients. We allow the partial database replicas to lag behind the master until they are used for queries, by which time they must catch up to the scheduled snapshot.

We now give the formal definition of the scheduled releasing mechanism. For any given time $t > t_0$, where t_0 is the initial database time, there exists one and only one time period $[t_i, t_i+\Delta t_i)$, 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 or a tolerable time interval, any query issued at t will be responded with the same result as if the query is executed at t_i. We require the staleness limit to be much larger than the network latency, because from the user's point of view the network latency is automatically added to the staleness of every response, therefore we cannot promise the staleness to be less than that. Larger staleness limit will also have positive effects on the scalability.

We require the releasing schedules be 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 in line with the partially synchronous distributed model and eliminates the unnecessary web and database operations.

The replication control algorithm is described in the following. The pseudo code is depicted in Figure 2. In our algorithm, an update is routed to the corresponding master database partition server allocated for that producer, and executed in an ACID manner, e.g., using the strict two-phase locking (2PL) protocol. The queries executed at the clients are against their explicitly scheduled snapshot, and we require the clients to synchronize to that snapshot with the master database before the execution. This is similar to the multi-version mixed method described in [6] or the snapshot isolation protocols [4] with one important distinction. In our method the snapshots are chosen a priori and independently from the database states and the timing of the queries.

```
Upon: submit of a read-only transaction T to client at time t
1:  assign T the timestamp t_i, the starting time of its scheduled time interval
2:  if local database is not synced to the snapshot at t_i:
3:    request from all the master partitions the writesets up to [t_{i-1}, t_{i-1}+Δt_{i-1})
4:    sync local database the snapshot 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 a master partition
8:  atomically request necessary shared and exclusive locks
9:  wait until all locks are granted
10: execute T at master partition, record commit time t
11: for all the materialized views V_i covering t:
12:   update V_i to reflect the writeset of T
13: release locks of T
14: return ok

Upon: submit of a request to a master partition for writeset for time interval [t_n, t_n+Δt_n)
15: for all the materialized views V_i on this partition:
12:   if V_i is for [t_n, t_n+Δt_n):
13:     return writeset in V_i
```

Figure 2. Replication control algorithm.

Conceptually the scheduled releasing introduced here enforces a new transactional state to the database. Traditionally we assume once an update is committed its changes are immediately visible to all the other active transactions. We revoke this assumption and define a "QUERY VISIBLE" state after "COMMITTED", as shown in Figure 3. The changes made by an update are still immediately visible to other updates once commit-

ted, but are visible to active queries only when they reach the "QUERY VISIBLE" state.

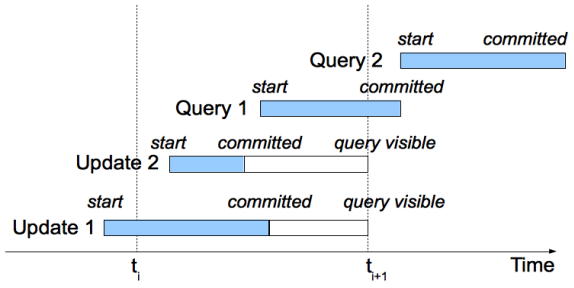


Figure 3. Update visibility and serial execution.

In comparison, if we use the mixed protocol or snapshot isolation, Query 1 in Figure 3 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. The scheduled release time interval is independent from the query load. When the query load becomes heavier, more updates become visible at each scheduled release, but the number of snapshots to be propagated per time unit remains the same. On the other hand, if both updates in Figure 3 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 being committed, the clients don't care to know. Further, since we did not relax the irrevocability aspect of the COMMITTED state, the ACID property of the database still holds.

Proving the 1SR property of this protocol is a two-step process. We first show that there exists a one-copy equivalence of this protocol. We then show this one-copy equivalence is serializable. Due to the scheduled release and the master-replica differentiation between the update-query transactions, executing queries on the client-side replicas introduces no data contention, and the network latency is masked by the timestamps. We can easily see that any query executed on the client 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 ACID manner on each partition, there exists a serial execution of the updates on the same partition, and they are serialized by the sequence of their commit times. Since global time exists across all partitions, there also exists a global serialization, ordered by the global time and segmented by the scheduled releasing. 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.

4. Implementation

We implemented a twitter-like feed following application prototype. The server side was fully implemented with Python/Django and PostgreSQL. We chose PostgreSQL as the backend database because of its mature support for the time travel functionalities, which goes back to its origin. The scheduled releasing, time map, and related functionalities were implemented with triggers and programmed in PL/pgSQL. Nonetheless, the database was queried primarily as a key-value store.

The client-side functionalities could have been fully implemented in Javascript and client side database, but we were concerned about how to evaluate the system performance. At the time of the experiment, our Amazon account only allowed up to 100 instances running at the same time, but we potentially needed thousands of real browsers running in parallel to generate the desired workload. We eventually decided to first implement a simplified emulated browser in Python/Django and PostgreSQL.

We picked the maximum staleness level at 5 seconds. But for those followers who didn't follow many active producers, it would have taken them numerous 5-second time map queries to gather the 20 most recent tweets. We therefore also implemented time maps for the following larger granularities to speed up the time map query: 30 seconds, 3 minutes, 15 minutes, 1 hour, 4 hours. Accordingly, we also added slightly more materialized view maintenance work when committing each new tweet. If a client is not able to gather sufficient tweets from a time map query, it will attempt an earlier time map at either the same or larger granularity level if it's available.

5. Evaluation

To better evaluate our implementation we must generate more realistic social networking workloads. Attempting to compare our implementation with the Yahoo! PNUTS system, we used the same zipfan parame-

ters as used in that experiment [28]. Both the social network and the synthetic workload were generated with Yahoo! Cloud Serving Benchmark [10], and Table 1 lists the parameters used in both experiments.

Since our implementation relies heavily on the clients being able to programmatically interpret the workload and the server responses and do local calculations as well as database operations, we need large number of machines and computing power to emulate the client-side processing. The only viable test environment seemed to be the computing cloud. We therefore set up our test environment in Amazon EC2.

Table 1. Comparing workload parameters with the Yahoo! PNUTS experiment [25]

		PNUTS	This
Number of producers		67,921	67882
Number of consumers		200,000	196,283
Consumers per producer	Average	15.0	13.38
	Zipf parameter	0.39	0.39
Producers per consumer	Average	5.1	4.63
	Zipf parameter	0.62	0.62
Per-producer rate	Average	1/hour	1/hour
	Zipf parameter	0.57	0.57
Per-consumer rate	Average	5.8/hour	varied
	Zipf parameter	0.62	0.62

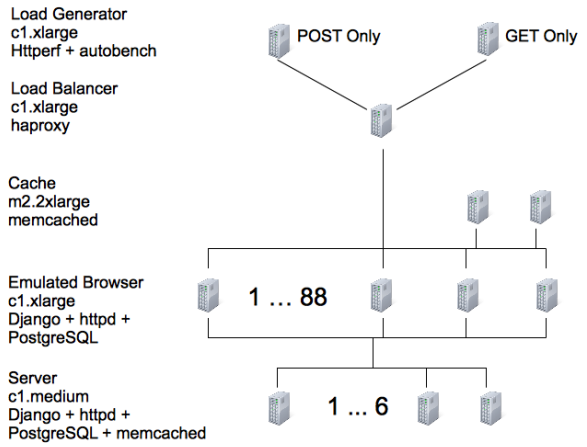


Figure 4. Experiment configuration

Figure 4 schematically shows the multi-layer server configuration deployed to conduct the experiment. On the bottom layer we deployed a small number of low-end servers, initially up to 20 small instances (m1.small) then upgraded to 3 to 6 high-CPU medium instances (c1.medium). The reason we upgraded was because even at the maximum number (100) of total instances we still were not able to generate sufficient client-side processing power to drive up to 40% of the query load in the Yahoo! PNUTS experiment. We therefore decided to move more virtual machines to simulate the clients rather than further increasing the server numbers. These low-end servers also run their local memcached service. Directly above the servers were up to 88 high performance instances (high-CPU extra large instance, or c1.xlarge) used to simulate the client-side processing. These emulated browser machines then shared up to 2 high-performance instances (m2.2xlarge, or high-memory double extra large instance) running standalone memcached server to simulate the web caching. We then ran HAProxy on one c1.xlarge instance to evenly distribute the workloads to these emulated browsers, and had two c1.xlarge instances, both running httperf and autobench, to drive the workload and run benchmarking, one for the update load and another for the query load. All these instances also had collectd installed and had various statistics reported back to our cloud service control panel.

Much like the other web applications, social networking applications' query load vastly dominates their update load. If the Yahoo! PNUTS experiment workload is any indication, the query load exceeds 99% of the total requests.

In our experiment, however, we decided to use a fixed update load at 19 requests per second, only slightly higher than the average update load in the PNUTS experiment. We then slowly drove up the query load until any server returned a 500 code. Figure 5 shows the linear scalability observed when the number of the servers was increased from 3 to 6. Figure 6 depicts the latency-load relations under different server configurations. Beyond 6 servers, the client-side simulation became the bottleneck and no meaningful data could be obtained. The linear scalability property is further supported by the following observations:

First, we observed extremely high cache hit rates, oftentimes exceeded several thousands to one, at the standalone memcached servers deployed to simulate the web caches, as shown in Figure 7. This indicated the success of the mass transit style approach we employed to build the feed following applications. The changes of the cache hit rate corresponded nicely to the scheduled releasing times, increased during the intervals, and had sudden dips at the release time points when newly released data caused cache misses.

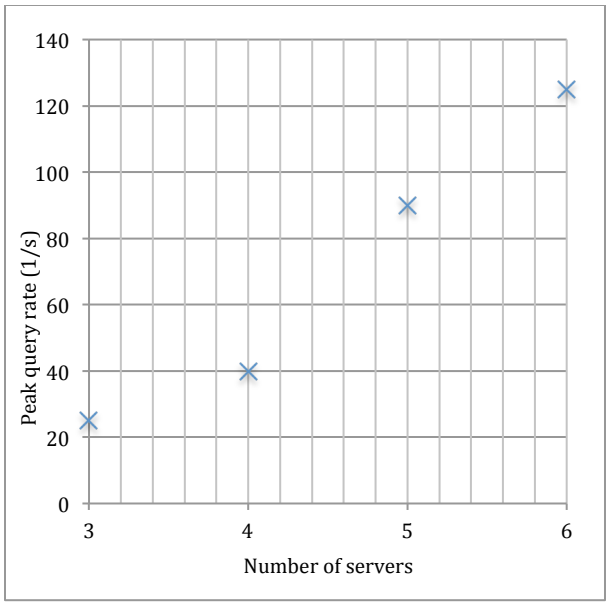


Figure 5. Peak query rates vs. number of servers under the query/update combined workload

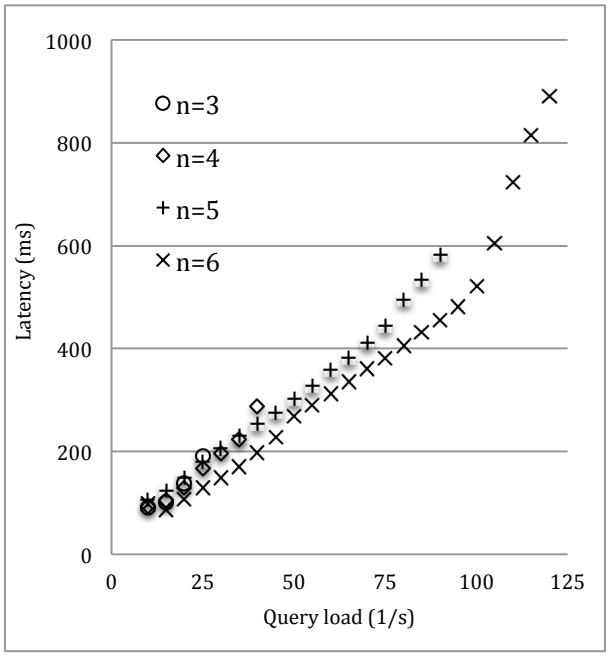


Figure 6. Latency vs. query load, with constant update load at 19 updates per second.

Secondly, at the local memcached service inside the server, we also observed fairly good, but not much as high, cache hit rates, at approximately 2 to 1 to 4 to 1. This indicated that the distributed system overall fared pretty well in not bothering to contact the origin server unless absolutely necessary. This is in line with the principle of the web architecture and the web caching.

Finally, we observed that when the query load increased, the CPU load increased much faster at the emulated browsers than at the servers, as shown in Figure 8 and 9. This illustrated the strength of this approach, which distributed a larger portion of the increasing processing load to the clients than absorbed by the servers. This was the source of the favorable scalability properties at display in this experiment.

6. Related Work

Partitioning, replication, and caching are time-tested and battle-hardened strategies to scale web applications as well as the databases that drive these applications. Unwilling to voluntarily give up on consistency, researchers typically rely on consistency guaranteeing network communication protocols to implement these strategies. Examples include the multicast total ordering as used in Postgres-R [21], RSI-PC as used in Ganymed [26], the total ordering certifier as used in Tashkent and related designs [13][22][7], Pub/Sub as used in Ferdinand [17], and the deterministic total pre-ordering [29]. Their performance is in theory upper-bounded by the centralized service implementing these protocols, and in some cases also by that of the snapshot isolation [4]. In contrast, our design differs fundamentally from these systems in terms of the distributed system model. Relying on global time, we managed to eliminate all centralized services and more efficiently implement the partitioning, replication, and caching. At the partition level, our design may also be considered a much simplified special case of the snapshot isolation, where not matter how high the query load is, only one snapshot is taken for each scheduled release interval.

Recognizing the CAP tradeoff [18], NoSQL systems like Dynamo [12], MongoDB [25], CouchDB [11], and Cassandra [2] etc. conscientiously sacrifice 1SR for better scalability, but the inconsistency exposed thereafter is undesirable even for non-critical web services such as social networks. More recent NoSQL systems such as PNUTS [9] slightly tighten this up but are still lacking. Different from these systems, our design guarantees 1SR by design. Performance wise, since our design pushes a larger portion of the increased load to the clients, we anticipate performance advantages under higher load. Such advantages are inherently true even compared to much faster systems that move all data to the in-memory distributed cache, for the same reasons that more highways and faster cars do not diminish the advantages of the mass transit systems.

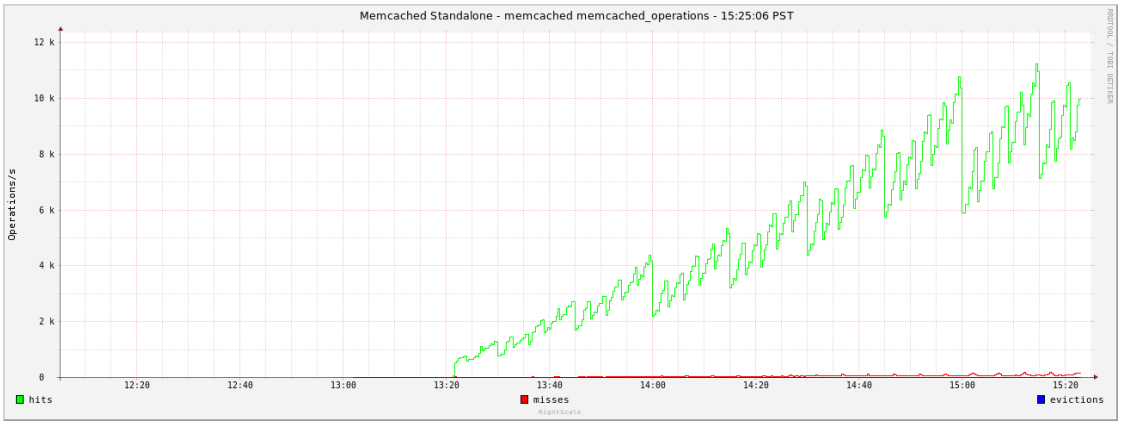


Figure 7. Cache operations at the standalone memcached server

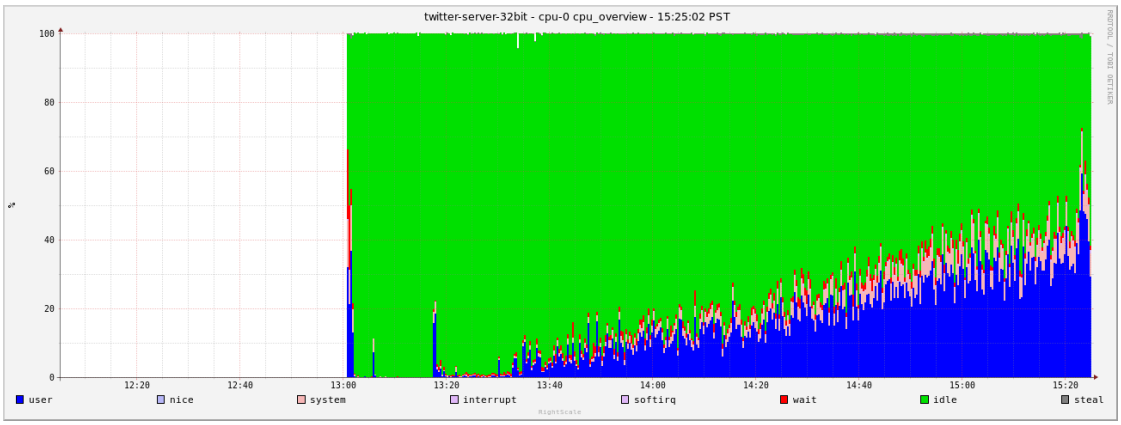


Figure 8. CPU load of a server

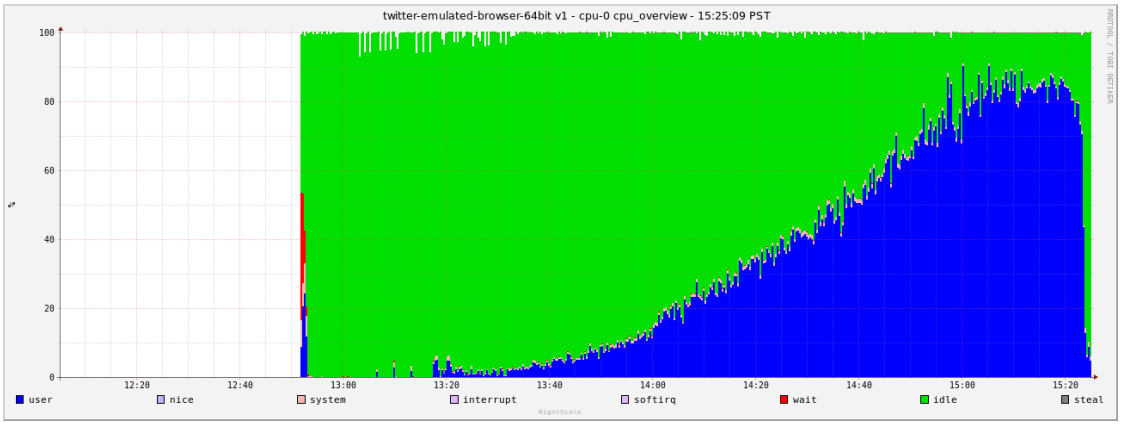


Figure 9. CPU load of an emulated browser

Trading staleness for scalability isn't a new idea, but the previous systems didn't attempt to preserve 1SR [1][16][19][5]. Our approach is the first known to us that guarantees 1SR under this tradeoff.

This paper is primarily inspired by Roy Fielding's blog post on RESTful feed following [15], and much of the experimental verification is adopted from the Yahoo! PNUTS experiment [27]. Fielding's approach is distinctively timing-based, but he did not elaborate on the the-

oretical foundation, the replication control algorithm, the freshness-scalability tradeoff involved, and the database partitioning, replication, and caching details. His implementation is based on a single centralized server, while ours is based on shared-nothing distributed systems. Nevertheless, his emphasis on the RESTful design [14] and the web architecture [3] reminds us that scalability is not a pure database endeavor but requires a holistic view of the system.

7. Discussions and Future Work

Because the server side scalability is known to be the primary bottleneck for the current social networking applications, in this paper we evaluate this new approach mainly from the server's perspective. We have not delved into its detailed effects on the real web browsers. But as an indicator, we have observed a peak client load of approximately 2 queries per second per client with cloud instances running the emulated browsers. When running a real browser on a physical machine the result may be slightly different due to various factors including the absence of the hypervisor isolation and the cloud CPU throttling, the real browsers' performance differences, and their implementations of the client-side database. This will be the topic for further research, but in general we are optimistic that the clients' processing capability should not pose an intractable bottleneck. This is because in our approach the same-client queries beyond the rate of once per 5 seconds do not require additional client and server processing. The browser will simply respond with the same, client cached response generated for an earlier request. If the higher query rate is caused by more clients initiating queries concurrently, these additional clients also bring in more processing power to counterbalance the increased client side processing needs.

Another practical concern for this approach is the wide adoption of the mobile devices as the social networking clients. While the processing capabilities of these devices may be continuously improving, the network bandwidth provided for these devices is harder to reach a satisfying level. For these non-performing clients, it is still possible to tier the web services such that many emulated browsers we used in the experiment can be deployed at the edge of the web (e.g., CDN) and repurposed as proxy servers for them. At least in theory this does not alter the linear scalability property of this approach.

The third practical concern is the implementation of global time. In our experiment all clients and servers frequently synchronized their local times with authoritative time sources using ntpd. The small time skews resulting in this approach did not pose a problem for the duration of our experiment. Even if the clients lag far behind the servers, the system performance should not decline significantly if sufficient web cache is provided. Nevertheless, in practice the assumption may be too strong for all the clients and servers to always maintain global time. In a follow-up research we relax this condition to only requiring all the servers to be properly synchronized. The client must send one extra request to detect the global server time before any timeline request can be processed. Under such relaxation, we can still show that the consistency guarantee and the linear scalability property are largely maintained.

Finally, we also noticed a potential problem with the load balancing. We employed a naïve database partitioning strategy but have not built any load elasticity and protection. Under such circumstances, even if all the other servers were way below their capacities, if one partition server encountered aberrant load spike and crashed, the whole system would crash. This issue may be addressed in the future work. We also expect to further this research by experimenting with different staleness levels to investigate their impacts on scalability, and extending load generation capabilities to further verify the linear scalability property.

To summarize, in this paper we describe, implement, and evaluate a novel method that can simultaneously achieve scalability and consistency in feed following applications built on shared-nothing distributed systems. In our experiments the servers scaled linearly, and sustained sufficiently high workloads to be of practical use for small to medium size social networks. The cost of running 6 low-end servers was fairly reasonable for the performance and the capacity they delivered.

We also demonstrate for the first time the feasibility of a new design pattern that consistently trades freshness for better scalability. This is achieved by assuming the availability of global time in a shared nothing distributed system, timing the queries with a pre-published release schedule, pushing much of the personalized query workload to the clients, and more efficiently partitioning, replicating, and caching.

References

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

[2] Apache Cassandra, http://cassandra.apache.org/.

[3] Architecture of the World Wide Web, Volume One. http://www.w3.org/TR/webarch/.

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

[5] Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R. and Tamma, P. 2006. Relaxed-currency serializability for middle-tier caching and replication. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (Chicago, IL, USA, 2006), 599–610.

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

[7] Bornea, M.A., Hodson, O., Elnikety, S. and Fekete, A. 2011. One-copy serializability with snapshot isolation under the hood. *2011 IEEE 27th International Conference on Data Engineering (ICDE)* (Apr. 2011), 625–636.

[8] Brewer, E.A. 2000. Towards robust distributed systems (abstract). *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2000), 7–.

[9] Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D. and Yerneni, R. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1, 2 (2008), 1277–1288.

[10] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing*. (2010), 143–154.

[11] CouchDB, http://couchdb.apacheorg/.

[12] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. 2007. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220.

[13] Elnikety, S., Zwaenepoel, W. and Pedone, F. 2005. Database Replication Using Generalized Snapshot Isolation. *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems* (2005), 73–84.

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

[15] Fielding, R.T. 2008. Paper tigers and hidden dragons. http://roy.gbiv.com/untangled/2008/paper-tigers-and-hidden-dragons.

[16] 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). 1995, 445–456.

[17] Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C. and Tomasic, A. 2008. Scalable query result caching for web applications. *Proc. VLDB Endow.* 1, 1 (2008), 550–561.

[18] Gilbert, S. and Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*. 33, 2 (2002), 51-59.

[19] 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* (2005), 457–468.

[20] Herlihy, M.P. and Wing, J.M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (Jul. 1990), 463–492.

[21] Kemme, B. 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.

[22] Krikellas, K., Elnikety, S., Vagena, Z. and Hodson, O. 2010. Strongly consistent replication for a bargain. *2010 IEEE 26th International Conference on Data Engineering (ICDE)* (Mar. 2010), 52–63.

[23] Lamport, L. 1986. On interprocess communication. *Distributed Computing*. 1, 2 (Jun. 1986), 86-101.

[24] Lynch, N.A. 1996. *Distributed Algorithms*. Morgan Kaufmann.

[25] MongoDB, http://www.mongodb.org/.

[26] 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.

[27] Silberstein, A., Machanavajjhala, A. and Ramakrishnan, R. 2011. Feed following: the big data challenge in social applications. *Databases and Social Networks* (New York, NY, USA, 2011), 1–6.

[28] Silberstein, A., Terrace, J., Cooper, B.F. and Ramakrishnan, R. 2010. Feeding frenzy: selectively materializing users' event feeds. *2010 International Conference on Management of Data* (Indianapolis, IN, United states, 2010), 831-842.

[29] Thomson, A. and Abadi, D.J. 2010. The case for determinism in database systems. *Proceedings of the VLDB Endowment*. 3, (Sep. 2010), 70–80.

[30] Vogels, W. 2009. Eventually consistent. *Communications of the ACM*. 52, (Jan. 2009), 40–44.