Technical Report: HybridTime - Accessible Global Consistency with High Clock Uncertainty

David Alves
Cloudera Inc./UT Austin

Todd Lipcon
Cloudera Inc.

Vijay Garg
UT Austin

This technical report was originally submitted, verbatim, to OSDI on 04/24/2014, the core ideas were submitted for a patent (app. number 20150156262) on 12/2013.

Abstract

Guaranteeing global consistency across distributed systems with uncertain physical clocks is a problem that concerns several types of distributed systems such as distributed databases. Google's Spanner database implements consistency across replicas groups by employing high precision clocks, reducing clock uncertainty, and then using *commit-wait*, a technique which ensures that causally related transactions are separated in time by more than the clock synchronization error bound. However, employing a similar technique outside of Google's data-centers may be difficult due to operational concerns or even impossible (e.g. when using cloud services like Amazon EC2).

In this paper we introduce HybridTime, a hybrid between physical and logical clocks, that can be used to implement a globally consistency database. Unlike Spanner, HybridTime does not need to wait out time error bounds for most transactions, and thus may be used with common time-synchronization systems. We have implemented both HybridTime and commit-wait in the same system and evaluate experimentally how HybridTime compares to Spanner's commit-wait in a series of practical scenarios, showing it can perform up to 1 order of magnitude better in latency.

1 Introduction

A growing class of distributed systems, e.g. distributed databases such as Spanner [7], are deployed at unprecedented scale and partitioned/replicated across multiple datacenters both to improve serving latency and fault tolerance. In such systems, transactions frequently span multiple datacenters and touch many servers, each serving different partitions of a dataset.

In order to provide good performance, many scalable distributed databases such as Amazon's Dynamo [9] and Facebook's Cassandra [13], relax the consistency model and offer only eventual consistency. Others such as HBase [1] and BigTable [4] offer strong consistency only for operations touching a single partition, but not across the database as a whole. These looser consistency models present difficulty for application developers, who are typically accustomed to programming against fully linearizable [11] single-node data structures.

Additionally, end users may be surprised and disturbed if they witness a consistency anomaly: for example, in a geo-replicated service, a user may first be routed to a local datacenter and perform some action such as sending an email message. If they then reload their browser and are routed to another datacenter backend, they would still expect to see their sent message in their email outbox. Looser consistency models such as timeline consistency [5] or eventual consistency [9] permit this kind of anomaly, which is likely unacceptable for many applications such as banking or online commerce.

The easiest way to address consistency issues would be to have perfectly accurate and synchronized clocks across all servers; however, as absolutely accurate physical timekeeping is impossible in distributed systems, the traditional approach to global consistency has been to use logical clocks. Logical clocks, such as Lamport Clocks [15] or Vector Clocks [10, 20]. Such clocks can provide global, causally consistent view of data and updates, but have two disadvantages: first, they require that all clients must propagate clock data to achieve consistent views, and second, the assigned timestamps have no relation to physical time. Therefore, they are unable to serve read requests that require a snapshot at a physical point-in-time, a feature available in commercial systems such as Oracle Flashback [17] and IBM DB2 [8] as well as academic temporal database projects such as ImmortalDB [19]. Moreover users expect that operations they know occurred far apart in the real world appear ordered as such independently of whether the system could establish a causal link between them, something that Vector Clocks of Lamport Clocks cannot provide.

It is important to mention that we refer to consistent **global** state that is not only consistent state across different replicas serving the same data, but also that state is kept consistent across different partitions of the database, possibly themselves spread out across geo-distributed datacenters. Systems such as Spinnaker [23] use Paxos [16] for the former, but fall back to pessimistic locking schemes for cross-partition operations. Many of the applications for scalable databases involve a significant analytical workload, in which read-only transactions frequently run over the entirety of the dataset. For these types of transactions, which may run for several minutes or even hours, locking-based schemes for cross-partition consistency are unacceptable, as they would prevent any other concurrent operations.

Spanner introduced *commit-wait*, a way of ensuring physical-time based consistent global state by forcing operations to wait long enough so that all participants agree that the operation's timestamp has passed based on worst case synchronization error bounds. While innovative, the system performance becomes highly dependent on the quality of the time synchronization infrastructure, and thus may have unacceptable performance absent specialized hardware such as atomic clocks and GPS receivers. Often, organizations rent computing infrastructure, e.g. Amazon EC2, and thus do not have the infrastructure control necessary to add such high precision timekeeping equipment. In this paper, we present **HybridTime(HT), a hybrid between physical and logical clocks** and we show how HT can be used to achieve the same semantics as Spanner, but with good performance even with commonly available time synchronization.

Like Spanner, our approach relies on physical time measurements with bounded error to assign HybridTime timestamps to events that occur in the system. However, unlike Spanner, our approach does not usually require the error to be waited out, thus allowing for usage in common deployment scenarios where clocks are synchronized through common protocols such as the Network Time Protocol, in which clock synchronization error is often higher than with Spanner's TrueTime. The trade-off is that, in order to avoid commit-wait, HybridTime requires that timestamps be propagated across machines to achieve the same consistency semantics as Spanner. Contrary to vector clocks, which can expand as the number of participants in the cluster grows, HybridTime timestamps have constant and small size.

HybridTime clocks follow similar update rules to Lamport clocks, but the time values are not purely logical: each time value has both a logical component, which helps in guaranteeing the same properties as a Lamport Clocks, and a physical component which allows the event to be associated with a physical point-in-time. Like logical timestamps, HT timestamps are required to be attached to every message to and from clients. HT timestamps do not suffer from the problems of Vector Clocks, which can grow over time as they accumulate state from different machines. Instead, they are more like Lamport Clock timestamps in that they have constant and small size.

Moreover, in contrast to both Lamport and Vector Clocks, when two events are separated by more than the maximum clock error, it is possible to establish a temporal ordering of events even if they are not causally related, or if their causal relationship was established through hidden channels that the system fails to capture.

In this paper we make the following contributions:

- We introduce the HybridTime clock, including update algorithms and proofs of correctness.
- We introduce a real world implementation that illustrates how the HybridTime clock can be used in the same context as TrueTime is used in Spanner.
- We evaluate the implementation in a real world scenario and demonstrate how HybridTime compares to Spanner/TrueTime in similar scenarios.
- We describe how commit-wait can be implemented with commonly available hardware and software.

The rest of this paper is organized as follows: Section 2 presents the background and related work relative to HybridTime and Spanner; In Section 3 we formalize the underlying assumptions, introduce the clock update algorithms and present proofs of correctness; In Section 4 we describe our implementation and how HybridTime can be used to provide the same guarantees as required by Spanner, in the same context; In Section 5 we evaluate the implementation and show its practical value. Finally in Section 6 we summarize the conclusions and make some final remarks.

2 Background and related work

One of Spanner's key benefits is that is **externally consistent**, which is defined as fully linearizable, *even in the presence of hidden channels*. In this paper we use the term *externally consistent* in the same sense. Additionally we use the term *globally consistent* to describe a system which provides the same linearizability semantics, provided that there are no hidden channels present.

Even though consistency within a replica group is widely implemented in large scale databases, this is not the case for inter-replica-group consistency. Dynamo [9] author's state that they use vector clocks to perform

conflict-resolution within a single row but offer no equivalent functionality for multi-row transactions. Cassandra [13], which shares similarities with Dynamo, uses plain wall clock timestamps, without taking clock error into consideration, and thus does not provide consistency either within or between replica groups. Similarly, BigTable [4] and its open source counterpart, HBase [1], assign timestamps locally on each server and do not offer consistency guarantees across replica groups (tablets). Horizontally partitioned SQL systems such as MySQL (typically described as "sharded MySQL") similarly offer within-partition consistency but offer no ability to query a snapshot across multiple partitions.

Consistent systems would be trivial to implement if all participants had access to a perfectly synchronized physical clock. However, it is well-known that it is impossible for servers in a distributed system to have perfectly synchronized clocks. Even with time synchronization mechanisms such as NTP [21], processes running on different machines in a system have inaccurate clocks which suffer some absolute error compared to an unknown theoretical reference clock. Additionally, real-life clocks exhibit skew over time: clocks on different machines may drift farther apart from each other as time progresses. Thus, databases have typically resorted to other mechanisms to assign timestamps to transactions in such a way that a causally consistent snapshot may be computed.

One such timestamping mechanism is to use logical clocks such as Vector Clocks [10, 20] or Lamport Clocks [15]. Systems utilizing such clocks to order events can be globally consistent; however, they are not externally consistent since they require that all client participants propagate clocks. For example, if client *A* communicates with client *B* over a hidden channel without forwarding a timestamp, there is no guarantee that client *B* will be able to see any modifications performed by client *A*.

Other systems such as Percolator [22], HBaseSI [27], and Cloudtps [26] use a centralized timestamp oracle process to assign monotonically increasing timestamps to transactions and determine a serialization order. Thus, they are both globally and externally consistent. However, this approach does not scale well to high transaction throughput, and would add unacceptable latency if a geo-distributed setup, where transactions running in remote datacenters would have to pay a costly round-trip to the timestamp oracle for each transaction.

Some systems implement some form of causal consistency across wide-area networks. COPS [18] or ChainReaction [2] enforce causality constraints across geo-distributed systems, but do not embed physical meaning in the timestamps. Finally, there has been noteworthy work [24] on reasoning about physical time and clock error with the aim of allowing to use physical time in similar ways as logical time, such as global predicate detection, but as far as we know it hasn't been applied to distributed databases.

Spanner's key innovation is that timestamps assigned by the system can be used to achieve external consistency, but also have physical meaning. This is a new aspect and is contrary to Vector Clocks and Lamport Clocks, in which timestamps are purely logical. Moreover Spanner assigns timestamps with known error bounds. Similarly we'll show that **HybridTime also provides physically meaningful timestamps with bounded error**.

3 HybridTime

As we'll show in this paper **HybridTime is always *globally* consistent, and through selective application of commit-wait is *externally* consistent**. As far as we know, ours is the only system that both supports message-based consistency (HybridTime) and time-based consistency (commit-wait) at the same time, allowing the user to choose on a per-transaction basis which one best fits the current needs. In the absence of hidden channels, HybridTime provides lower latencies and better throughput. In the presence of hidden channels, the user is free to revert to commit-wait and thus obtain *external consistency* guarantees.

We now present the details of HybridTime. In Section 3.1 we introduce the underlying assumptions. We then introduce the HybridTime clock and its update algorithm along with proofs of correctness, in Section 3.2.

3.1 HybridTime Assumptions

Independent of how physical time measurements are obtained, HybridTime relies on certain assumptions. In this section we will introduce these assumptions and show that they are plausible given the environment in which most modern distributed systems are deployed. *TrueTime* also makes similar assumptions. HybridTime assumes that machines have a reasonably accurate physical clock, represented by the $PC_i(e)$ function, which outputs the numeric timestamp returned by the physical clock as read by process i for event e, that is able to provide absolute time measurements (usually in milli- or microseconds since 1 January 1970). Virtually all modern servers are equipped with such a physical clock.

We moreover assume that there is an underlying physical synchronization substrate that keeps the physical clocks across different servers synchronized with regard to a reference server, the "reference" time[1], represented by the $PC_{ref}(e)$ function which outputs the numeric timestamp returned by the "reference" process for event e. Additionally, we assume that such a substrate is able to provide an **error bound** along with each time

measurement, denoted by the $E_i(e)$ function, which outputs the numeric value ε error of process i at the time e occurred[2]. Again this is reasonable to assume since virtually all large scale clusters execute time synchronization daemons, such as the previously mentioned NTP, which both synchronizes the server's clocks and provides at any moment a maximum bound on the clock error. It is worth mentioning that *TrueTime* makes similar assumptions. Finally we note that we make no assumptions on the actual accuracy of the clocks, i.e. the physical timestamps returned by server's clocks may have an arbitrarily large but finite error, as long as this error's bound is known[3]. Assumption 1 formalizes the expected relationship between $PC_{ref}(e)$, $PC_i(e)$ and $E_i(e)$.

Assumption 1. ***Physical clock error is bound***

$$\forall i, e : |PC_{ref}(e) - PC_i(e)| \leq E_i(e) \quad (1)$$

That is, we assume that the physical timestamp returned by each server's physical clock is within certain value of the reference time which is given and bound by the $E_i(e)$ function. This assumption is plausible since most NTP deployments do provide a maximum error with regard to a master daemon, the time reference. Spanner's authors mention that the TrueTime API does something similar albeit with much greater precision. It is noteworthy that that $E_i(e)$ represents different error functions, one per process, that the output of this function varies from process to process and over time.

We moreover assume that the timestamp returned by the physical clock is monotonically increasing:

Assumption 2. ***Physical clock timestamps are process-wise monotonically increasing***

$$\forall i, e, f : e \rightarrow f \Rightarrow PC_i(e) \leq PC_i(f) \quad (2)$$

That is, when a server's physical clock is queried for the current time, it never outputs a value that is less than a previous result. Again, this is plausible: NTP can be configured to adjust server clocks by slowing them down or speeding them up over reasonably large periods of time to ensure that applications that rely on time reads are not greatly affected by the adjustment. In certain cases, NTP may skip time forward or backward, but such cases are extreme and easily detected by monitoring the NTP daemon status. Servers may choose to decommission themselves or fail-stop upon detection of such an event.

Based on these assumptions we can now introduce the Physical clock API that is required at every node to support HybridTime, using each server's physical clock and the NTP protocol:

3.2 HybridTime Clock and Protocol

In this section we present the HybridTime clock (HTC). A HybridTime clock timestamp is a pair

Algorithm 1 The Physical Clock API

```
i = server_id()

1: function NOW : int physical, int ε
2:     physical = PC_i(now)
3:     ε = E_i(physical)
4:     return p, ε
5: end function
```

(*physical*,*logical*) where the first component is a representation of the physical time at which the event occurred and the second component is a logical sequence number. Algorithm 2 depicts the HTC algorithm.

Obtaining timestamps from the HTC clock works as follows: each time a timestamp needs to be assigned to an event or a message is sent, $HTC.Now()$ is called to obtain the latest Timestamp. The physical component of this "latest" timestamp is either the value obtained from the physical clock (PC) or the value stored in *last_physical* whichever is greater. Similarly the logical value of the "latest" timestamp is either 0 if the current PC value was selected or the next logical value in the sequence if *last_physical* was selected.

Updating the HTC clock works as follows: For any incoming timestamp ("in" in Algorithm 2) if the incoming value's physical component is lower that the one obtained through $HTC.Now()$ then no action is required. On the other hand if the incoming value's physical component is equal to the one obtained through $HTC.Now()$ then we set *next_logical* to be the maximum of the current logical value and the incoming logical value, plus one. Finally, if the physical component of the incoming value is higher than the one obtained through $HTC.Now()$ we set *last_physical* to it and *next_logical* to be the incoming timestamp's logical component plus one.

We later show that if one interprets an HTC Timestamp as a simple lexicographically comparable value, **Algorithm 2 implements a Lamport Clock**, with the additional **advantage that generated timestamps have physical meaning and are accurate representations of physical time within a bound error**.

Figure 1 depicts this update procedure for a sequence of 7 events where some are causally related. As can be seen in the figure, for any chain of events, both the physical and logical components of the assigned timestamps are monotonically increasing.

To order the events timestamped using the HybridTime Clock algorithm we use Definition 1.

Definition 1. $HCT(e) < HCT(f)$ **is defined as the lexicographical ordering of the timestamp two-tuple (physical,logical)**

Algorithm 2 The HybridTime Clock Algorithm

```
1:  types:
2:  type Timestamp of {int physical, int logical}
3:  var:
4:  int last_physical = 0
5:  int next_logical = 0
6:  PhysicalClock pc;        ▷ Provides the API in Algo. 1

7:  function NOW : Timestamp
8:      Timestamp now;
9:      int cur_physical = pc.now().physical;

10:     if cur_physical ≥ last_physical then
11:         now.physical = cur_physical;
12:         now.logical = 0
13:         last_physical = cur_physical;
14:         next_logical = 1;
15:     else
16:         now.physical = last_physical;
17:         now.logical = next_logical;
18:         next_logical++;
19:     end if
20:     return now;
21: end function

22: function UPDATE(Timestamp in) : void
23:     int Timestamp now = Now();
24:     if now.physical > in.physical then
25:         return;
26:     end if
27:     last_physical = in.physical;
28:     next_logical = in.logical + 1;
29: end function
```

The definition of $<$ usually allows to compare the majority of events without resorting to the *logical* value of the timestamp, as illustrated by Fig. 1. The HybridTime clock creates an ordering of events such that:

Theorem 1. ***The HybriTime clock happened-before relation forms a total order of events***

The proof of Theorem 1 is trivial since we are using lexicographical ordering to order the events.

Although HybridTime clock does not make a direct use of time measurement error we now show that such errors in measurement are bounded such that the algorithm does not accumulate error over time. This is important because, if no bound exists, then the physical components of timestamps will become meaningless and the ordering becomes nothing more than the one defined with Logical clocks.

We start by defining the error in any HybridTime timestamp assignment. Let $HTC_i(f)$ represent a HybridTime clock timestamp assignment which takes the form of a two-tuple $(physical, logical)$ value. Let t_f^{real} be the "real" time at which f happened and let $error(f)$ be the error on the assignment of the physical timestamp component by the HybridTime clock, i.e. $error(f) = t_f^{real} - HTC_i(f).physical$.

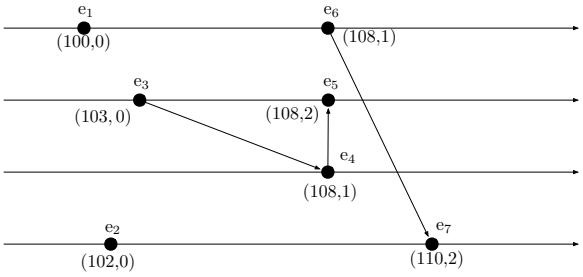

Figure 1: Example execution of the HybridTime clock algorithm

For any event f, if f takes its physical timestamp component from the physical clock due to Assumption 1 we know that any physical clock read at process i, ($PC_i()$) has bounded error given by function $E_i()$. This implies that, in Algorithm 2 if the event takes the value of the current clock read (line 11), then its error is equal to the error obtained from the physical clock. This translates to:

$$\begin{aligned} &\forall f, i : HTC_i(f).physical = PC_i(f) \\ &\rightarrow error(f) \in [-E_i(f), E_i(f)] \end{aligned} \tag{3}$$

We will now show that even if an event adopts the physical timestamp component of the event that preceded it, its error is still bound and is given by:

Theorem 2. ***For any event in a causal chain f, the physical component of a HTC timestamp approximates the "real" time the event occurred, with a error defined and bounded by***

$$\begin{aligned} &\forall e, f, (e \rightarrow f) \\ &\wedge (HTC_i(e).physical = PC_i(e)) \\ &\wedge (HTC_i(f).physical = HTC_j(e).physical) \Rightarrow \\ &error(f) \in]-E_j(e), E_i(f)[\end{aligned} \tag{4}$$

We provide a full proof of Theorem 2 in Appendix 6. Theorem 2 translates to the following. If the HybridTime Clock timestamp assigned to f originates from an event e that precedes it in a causal chain, and if e's timestamp was obtained from the physical clock, then the physical time error of the assignment of e's physical timestamp component to f is in the interval $(-E_j(e), E_i(f))$.

This means that the only difference between assigning a physical timestamp component to an event from the local clock or accepting one from another node is that

that the left side of the interval is bound by the error as measured when e occurred in node j instead of being bound by the local error.

4 Implementation, Relation to Spanner

We have implemented HybridTime and commit-wait external consistency on top of a research prototype distributed and partially ACID compliant database and performed a series of experiments in practical scenarios similar to the ones Spanner was designed for. In this section we introduce the research prototype database consistency modes, architecture and transaction semantics.

Our implementation supports multiple external consistency modes simultaneously. Clients are free to choose which one best fits the use case and choose a consistency mode for each write request. The following consistency modes are supported.

- **No Consistency** - In this mode there are no external consistency guarantees, transactions are assigned timestamps from each server's physical clock and no guarantee is made that reads are consistent or repeatable.
- **HybridTime Consistency** - In this mode our implementation guarantees the global consistency as Spanner, absent hidden channels, but using HybridTime instead of commit-wait. Clients choosing this consistency mode on writes must make sure that the timestamp that is received from the server is propagated to other servers and/or clients. Within the same client process, timestamps are automatically propagated on behalf of the user. If there are hidden channels, i.e. if somehow a write or a read causes another write or read without the timestamp being propagated, there is no guarantee of external consistency.
- **Commit-wait Consistency** - In this mode our implementation guarantees the same external consistency semantics as Spanner by also using commit-wait in the way described in the original paper. However instead of using TrueTime, which is a proprietary and private API, we implemented commit-wait on top of the widely used Network Time Protocol (NTP). Hence, in this consistency mode we support hidden channels.

This flexibility means that, when the application developer knows that there are no hidden channels, or is willing to propagate timestamps, they pay no commit-wait cost. However, in the case that the application developer does make use of an external channel (eg an enterprise service bus or external web service) they may selectively apply commit-wait in these situations. Based on the authors' experience working with large enterprises with big data applications, we believe that many applications are fully-closed systems with no hidden channels, and thus the vast majority of transactions can use HybridTime Consistency.

4.1 Architecture

We omit the full implementation details of the research prototype database and instead introduce the characteristics that are relevant for the implementation of the different consistency modes.

Our research prototype database partitions the key space and distributes these partitions across multiple machines. These partitions are analogous to *tablets* in BigTable or *regions* in HBase. Each partition is served by a set of replicas executing a consensus algorithm analogous to Paxos. The leader is the only replica that can accept writes. Each partition supports local transactions, i.e. updates to a single replica set are ACID compliant, but updates across replica sets are not atomic(A) or isolated(I) in the current implementation (although they are durable(D) and consistent(C))[4]. Hence clients wanting to read/write to multiple partitions issue multiple, independent, requests to each one.

We use MVCC to store multiple historical versions of each data item, each tagged with the timestamp at which the data was written, similar to the design used by BigTable or Vertica [14]/C-Store [25]. Read operations are assigned a timestamp and always read the most recently written version prior to that timestamp, ignoring any future or past data.

4.2 Types of transactions

Our prototype supports three types of operations. We use the same terminology that is used in the original Spanner paper:

1. **Read-Write(RW) Transactions** - These encompass all transactions that mutate existing data. A RW transaction may be preceded by a a read or sequence of reads and needs to acquire locks for each touched row before proceeding. In Spanner RW transactions come in two types. Those that touch multiple replica-groups (RG) and those than touch a single RG. In the context of this paper we have only implemented and evaluated the latter, but explain how HybridTime could be used for the former.
2. **Snapshot Transactions** - Snapshot Transactions are lock-free read-only transactions in the present. That is, a client requests the latest state of a certain set of data. However, because these rows may span different RGs and even data centers, nodes need to agree

what is the "present" and exactly which operations have been committed before the "present".

3. **Time-travel Read** - Time-travel reads are lock free read-only transactions in the past. That is the client request the state of a certain set of rows at a certain past point in physical time. Because, again, such rows may span different replica groups or data centers, all participating nodes need to agree on exactly which operations have been committed before the selected "past" instant.

It is noteworthy that, from a client perspective, the conceptual differences between *Snapshot Transactions* (2) and *Time-travel Reads* (3) are only that the former needs to be assigned a timestamp corresponding to the *present*[5] and the latter has its timestamp chosen as some *past* point in time. Because of this in our implementation we make only the distinction between **Write Transactions**, which mutate data and need to be assigned a timestamp which all intervening nodes agree upon, and **Read Transactions** which operate on some consistent point-in-time snapshot. Read-Write transactions, while interesting from a database perspective, make no difference on the implementation with regard to HybridTime since the only relevant difference is that they undergo a read phase where read locks are acquired prior to the write phase.

Delving more into the concept of "external consistency" in a database such as Spanner, we find that it is very close to two fundamental concepts: *i)* Operations must act upon and yield a **consistent global state** [12](CGS); *ii)* Operations must be **atomic** in the sense that, when acting upon multiple replica groups under a single transaction, a client which can observe changes to a replica group can observe changes to all involved replica groups. In practical terms *i)* means that if one client executes transaction A, which touches a set of machines, and then executes another independent transaction B, which touches another disjoint set of machines, other clients may only observe the transaction sequences $[]$, $[A]$, $[A,B]$. Since A and B are causally related the sequence $[B]$ is not a CGS and should not be observed. One the other hand *ii)* means that if a client executes a transaction that performs operation C on one replica group and operation D on another replica group, another client can only observe either $[]$ or $[C,D]$.

For the sake of clarity we omit the fact that in virtually all message exchanges, clients send to and receive from servers their last known HT timestamp. Similarly, this happens between servers, with the difference that servers can read and send the current value from the HT clock, while clients cannot and can only propagate HT timestamps received from other servers.[6]

4.3 Write Transactions

The execution of Write Transactions with HybridTime shares similarities with the execution of RW Transactions in Spanner. For a single-RG Write Transaction, the RG leader first acquires all relevant locks and then assigns a timestamp to the transaction, ts^{ht}. When replicas replicate the transaction they update their HT clocks according to the update rules introduced in Section 3.2. As soon as the leader receives a majority of replication acknowledgements, it applies the changes, deems the transaction committed, and notifies the client. Since for each transaction the HT clocks of participating replicas are updated, they agree that $ts_t^{ht}x$ is in the past, which is relevant for the execution of Read Transactions. Since all single-RG transactions have to go through the leader, there is a guarantee that if a transaction B is assigned a timestamp after the the timestamp of transaction A, all replicas will agree that B follows A.

Transactions that touch multiple RGs require a coordinator RG which executes a two-phase commit protocol, on top of the consensus protocol that executes within each RG. Upon receiving a request from the client, the leader from the coordinator RG issues *Prepare* commands to all participant RGs. The leader of each non-coordinator participant RG acquires the relevant locks and replies with its current HT timestamp. The coordinator RG leader then chooses the highest of the timestamps received and assigns that timestamp to the transaction, piggybacking this information as it sends an $Apply$ command to participant RG leaders. After receiving acknowledgments that the transaction has been applied by all participants the coordinator RG leader replies to the client. Note that in Spanner the client drives the two-phase commit protocol. This is similarly an alternative here but one which we left out for simplicity.

As clients piggyback their last known timestamp obtained from any server on each following request, it is easy to see that, according to the HT clock update rules, sequential transactions initiated from the same client to different replica groups have sequential timestamps, guaranteeing the desired "global consistency" property. If transactions are causally related but their causal relationship is established through hidden channels (i.e. channels in which the last known HT timestamp is not propagated), e.g. client 1 executes transaction A which then causes, through hidden channels, client 2 to execute transaction B, HT clocks alone cannot guarantee external consistency. In this case, the user may explicitly perform transaction A with commit-wait enabled, thus ensuring the external consistency property at the cost of commit latency. In the case that client 1 performs many write transactions before communicating on the channel, they may enable commit-wait only on the last of these trans-

action in order to achieve the same consistency with little impact on overall performance.

4.4 Read Transactions

All read transactions are executed on a consistent snapshot of global state without acquiring locks. Single node reads achieve this by implementing Multi Version Concurrency Control [3](MVCC) a well known database technique. Clients can choose to specify a timestamp for the transaction or leave it unspecified, in which case the transaction will execute on the most recent state possible.

For read transactions that access a single RG, if the client does not specify a timestamp, the transaction will be performed by the leader, as it is the only node that is guaranteed to have the most recent state. Upon receiving the request from the client, the leader assigns the transaction an HT timestamp, corresponding to the latest timestamp obtained from the HT clock, obtains an MVCC snapshot and executes the transaction. Still for the single RG read transaction case, the client might specify some timestamp in the past in which to execute a transaction. Replicas within a RG are required to keep up with the leader within a fixed, pre-configured time interval, otherwise are evicted. This means that if a client wishes to perform a point-in-time read that it knows is further in the past than the replica delay period, it can route the request to any replica instead of having to mandatorily route it to the leader, as it is guaranteed that if the replica is still alive it contains the transactions up to and including the requested point in time. If the client specifies a point-in-time that is more recent than the aforementioned period, the transaction reverts to being routed to the leader.

Read transactions that span multiple RGs execute much like the ones that span a single RG with some subtle differences. If the client provides no timestamp or if the specified timestamp is sufficiently close to the *present* that it falls within the maximum physical time measurement error (which is normally set to 1 second) of at least one replica, then an initial negotiation round must be performed where the client obtains the current timestamps from all participant leaders and chooses the oldest one as the timestamp for the snapshot. The Read Transaction then proceeds as in the single RG case. In this case Spanner's authors chose to have the client choose the timestamp in all cases, which avoids the initial negotiation round for recent transactions but may force the Read Transaction to wait until the chosen timestamp is safe (i.e. definitely in the past). We chose not to adopt this approach, as clients may often run on unsynchronized machines. Due to lack of time for the purposes of this paper, we did not implement the negotiation phase where the client obtains the most recent timestamps from the leader, and focused our evaluation on single RG Read Transactions. We note that the original Spanner paper doesn't measure such transactions either, and we would have no basis for comparison in any case.

5 Experimental Evaluation

In this section we present our experimental results and setup and demonstrate how HybridTime can be used along side commit-wait and how it compares in terms of latency.

5.1 Workload

We used the well know YCSB [6] as a workload for all experiments. We used up to three simultaneous clients, each running 8 threads with a mix of 60% insert, 20% update, and 20% single-row read. The three clients correspond to the three different consistency modes: each client performs all writes with a single but differing mode.

5.2 Experimental Setup

We executed all experiments in Google Compute Engine, an Infrastructure-as-a-Service (IaaS) cloud provider. All experiments were performed in the *us-central1-a* and *europe-west1-a* "zones". In each "zone" we assembled a cluster of 10 machines of type *n1-standard-8*. This type of machine has 8 "Virtual CPUs" each of which corresponds to: *"a single hyperthread on a 2.6GHz Intel Sandy Bridge Xeon or Intel Ivy Bridge Xeon (or newer) processor"* so, more precisely, each machine has 4 cores with hyperthreading. Each machine also has 30 GB of memory and to each one we attached a *"persistent disk"* of 350 GB capacity, formatted in the *ext4* format, in which we store the persisted data and the write-ahead-logs for the database. Our prototype database runs on Linux, hence each machine was booted to a `debian-7-wheezy` image. The operating system is installed to a different partition from the one that stores that database data.

Server machines are configured to run NTP version 4.2.6 with the following configurations changed from the default:

- The machine is configured with a single timeserver provided by the Google Compute Engine infrastructure. We assume that this timeserver is typically in the same datacenter as the machines being synchronized, providing for low RTT and good upstream clock quality. We believe that most datacenters already have NTP sources available within a short RTT, and as databases using techniques such as HybridTime become more commonplace, cloud

providers will start to treat high-quality NTP as standard infrastructure.

- The maximum polling interval is set to 8 seconds, to ensure that the clock error bound is adjusted frequently.

- The Allan intercept configuration is modified to 8 seconds.

These non-standard configurations prioritize tight clock error bound estimates over actual clock accuracy. That is to say, effects of jitter on the actual clock quality is worsened, but the maximum error bounds provided to the kernel remain much smaller. For systems such as HybridTime, a tighter bound on error has more benefit for the application than smaller average clock offset. The following table shows the maximum error numbers we extracted from our clusters for one and two datacenters.

NTP Max. Error	Min.	Max.	Avg.	Stddev
Single DC	11.5	16.7	14.73	$\pm$ 2.468
Two DCs	12.3	20.1	16.36	$\pm$ 2.581

Unless explicitly stated all experimental results were obtained **with full durability enabled**, i.e. no mutations become visible without the system having made a call to `fsync()/fdatasync()` for the the client's data[7], so that the chances of losing data on a machine crash are minimized. This on par with or beyond the settings of some widely used large scale databases, e.g. Cassandra uses a background thread that calls `fsync()` periodically thus leaving a bigger window for data loss on a cluster-global power outage, and HBase, at the time this paper was written didn't yet have "proper fsync support"[8].

Because we performed the experiments on an IaaS cloud provider, we were unable to obtain the hardware/software layout that underlies the storage system, making it impossible to provide a full description of how *"persistent disks"* in Google Compute Engine are actually implemented. However, as durability is usually a major concern for any database and an `fsync()` system call typically takes a non-trivial amount of time to complete on commodity hardware, for reference, we took some measurements of the latency characteristics of the storage medium in which we persist the data and write-ahead-log. We used the *"flexible I/O tester"* benchmark tool with a 4 thread workload with 2 writers and 2 readers, each writer making an `fsync()/fdatasync()` call on each write operation, and writing 1KB blocks, roughly the same size as a row in the database benchmark workload. For this paper the most relevant results are the write latencies as these may represent a significant portion of total request time and these were (in msec): `min=1 , max=39 , avg= 1.93, stdev= 2.08`. This means that, **in average, each client request will spend at least $\pm$ 2 milliseconds waiting on I/O** which is an important factor in the experiments we describe in the following section.

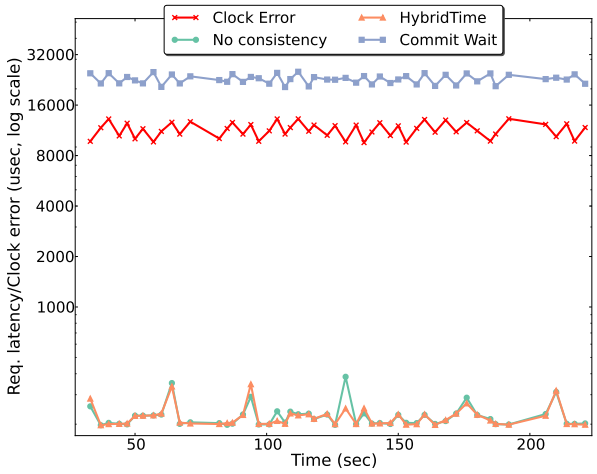

Figure 2: Write latency values with without replication and `fsync()`, log scale

5.3 Experiment 1: Single datacenter, no consensus

The goal of Experiment 1 is to reveal the impact each of the consistency modes has on request latency without other database concerns such as durability or fault-tolerance coming into play, since addressing these two requirements has a sizeable impact on database performance. **The question we aim to answer with this experiment is, other database concerns aside, what is, as close as possible, the standalone impact of the different consistency modes: no consistency vs. HybridTime consistency vs. Commit-Wait consistency in total request latency**. Hence, for this experiment only, we disabled `fsync()` calls, relying on the OS buffer to best choose the time to actually persist the data. Moreover, for this experiment only, we disabled consensus replication thus writing data from client requests to a single node. We performed Experiment 1 on a 10 node cluster on a single datacenter. Figure 2 displays the main results we obtained.

The chart in Fig. 2 plots latency over an arbitrary period of time, as measured on the server, for each of the consistency modes, overlapped with clock error. Latency times are a moving average, each data point representing the average latency, in microseconds, since the previous datapoint. The chart in Fig. 3 plots the latency histograms, as measured on the YCSB client, for percentiles *50%,75%,90%,99% and 99.9%*.

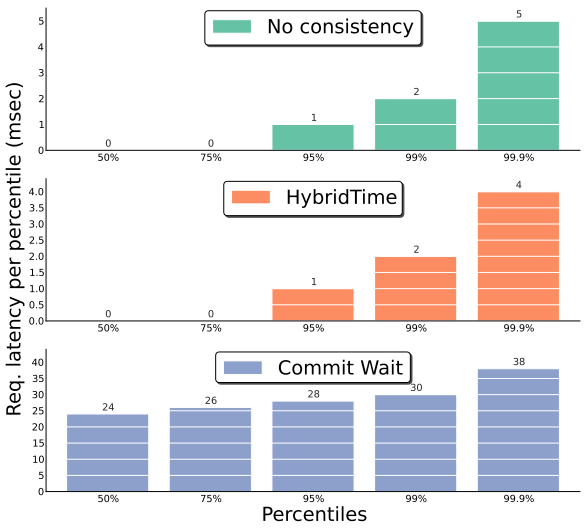

Figure 3: Write latency percentiles with without replication and `fsync()`

An important takeaway is that HybridTime consistency has virtually no impact on latency. Although expected, this **shows that, as claimed, it is possible to obtain the same consistency benefits of commit-wait through HybridTime** by simply requiring that a single timestamp be propagated by clients. Another takeaway from both charts is that, durability and replication aside, the clock error has a very big impact on total latency for commit-wait consistency. Commit-wait consistency requests are about 2 orders of magnitude slower than non-commit-wait ones (without `fsync()` and replication) with the maximum clock errors we were able to extract from a traditional, albeit slightly tweaked, NTP setup. As expected, the latency of commit-wait requests closely track the clock error, increasing and decreasing as it does. This serves as a baseline for the following experiments.

Another very important takeaway is that while the maximum clock error values that we were able to obtain from NTP were higher than the ones that TrueTime achieves and Spanner leverages (11-15 msec vs. 5-7 msec), they are not much higher, usually 2-2.5x. While twice the time is significant, we note that we used public servers to synchronize our time master, and the time master itself had a 6-9 msecs maximum error. We argue that **this signifies that a setup in which commit-wait is used with the typical NTP protocol, while maintaining latency times close to the ones in Spanner, is not only plausible but possible**, e.g. for close enough datacenters with a common, good time source, or whenever there is access to a/some close and trustworthy stratum 1^9 source(s). We intend to explore this further in future work.

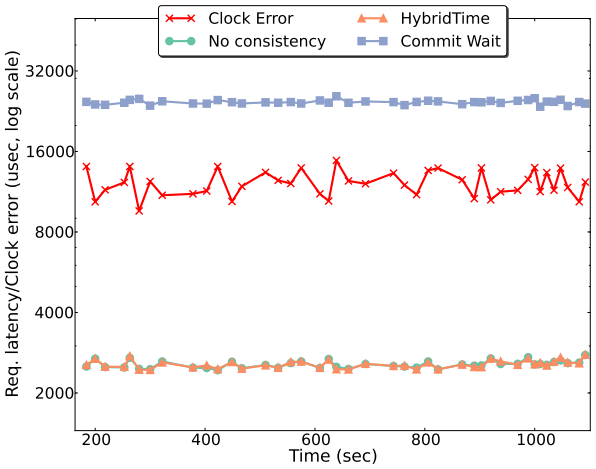

Figure 4: Write latency values with replication and `fsync()`, log scale

5.4 Experiment 2: Single datacenter with consensus

For Experiment 2 we evaluated request latency with durability and consensus replication enabled. **We aim to answer two questions with this experiment: i) What is the percentage of time spent waiting for commit-wait consistency with replication and durability enabled; i) Is it such that it makes HybridTime consistency a preferred option in some cases**.

The chart in Fig. 4 plots, again, latency over an arbitrary period of time, as measured on the server, for each of the consistency modes overlapped with clock error. Again, latency times are a moving average, each data point representing the average latency, in microseconds, since the previous datapoint.

At is can be seen in Fig. 4, even with durability and replication enabled, HybridTime consistency requests take about one order of magnitude less to complete than commit-wait request. In fact **HybridTime requests take about 5 times less to complete than single replica group writes as reported on the Spanner paper, which makes HybridTime a possible choice in latency-sensitive scenarios even if TrueTime-like accurate clocks are possible.**

Fig. 4 shows that even with a full database stack, a lot of the time for commit-wait consistency requests is still spent waiting out the clock error. That is, even overlapping replication which transmits data to other nodes (who must call `fsync()` before replying) and the local write-ahead-log append, 99.9% of the requests spend 86% percent of the total request time waiting out clock error. We do not make a direct waited-time comparison to the numbers published on the Spanner paper because they're reported single-replica, no commit-wait, request

times are higher than the times we've obtained from our database prototype even with replication included. We speculate this is possibly due to other features implemented in Spanner that are unpublished or that our research prototype does not yet support, like multi-replica-group write transactions.

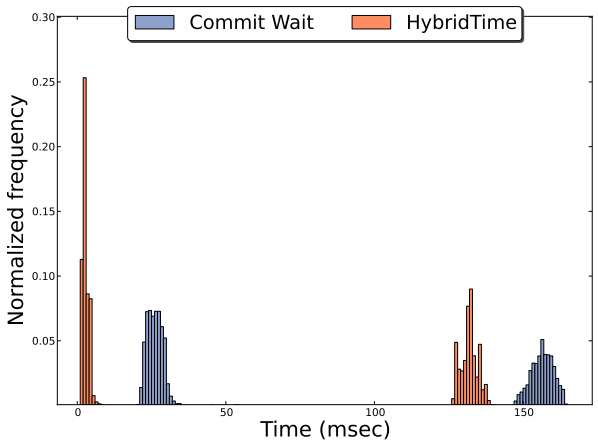

Figure 5: Write latency histograms, 2 datacenters.

5.5 Experiment 3: Two datacenters

For Experiment 3 we deployed our research prototype database to two different datacenters and had clients in each datacenter write to all partitions. The goal of this experiment is both to demonstrate that, like Spanner's commit-wait, HybridTime also works on a geo-distributed cluster and that throughput from the YCSB client is much higher for HybridTime consistency, vs commit-wait consistency.

Fig. 5 displays the normalized latency histograms for Experiment 3. One important takeaway from this experiment is that not only does HybridTime show lower latency values but these are also more stable, for instance 25% of all requests fell under the same bin in YSCB. This makes sense since commit wait latencies vary with the clock error, while HybridTime latencies only depend on the server implementation, which hints that HybridTime might be a better alternative when latencies need to follow strict SLAs.

Throughout the experimental evaluation we did not focus on throughput as that is highly dependent on the number of servers and clients, however it is worth mentioning that a single YSCB client using HybridTime consistency achieves roughly 20 times the throughput of a client using commit-wait consistency.

6 Conclusion

In this paper we've introduced a new global consistency algorithm, HybridTime, which experiments show keeps 1 order of magnitude lower latency than commit-wait implemented on top of NTP. HybridTime both provides *happened before* ordering of transactions across a globally distributed database and error bound, physically meaningful timestamps, which can be used to answer point-in-time queries.

We've also demonstrated how commit-wait can be implemented on top of the widely used NTP protocol with latencies that are worse, but within the same order of magnitude, as Spanner's. Moreover we've show that both HybridTime and commit-wait can live in the same system so that users are free to choose one or the other on a per use-case basis. If hidden channels may exist, then commit-wait is the better alternative and the only one that ensures global consistency at the cost of being 1 order of magnitude slower to answer client request. If on the other hand the system is closed and there are no hidden channels then HybridTime still provides cluster-wide consistency, even for geo-distributed clusters, but keeps much lower latencies.

A - Proof of bound physical time error on HybridTime timestamp assignment

Proof. Let e be the event occurring at node j prior to event f occurring at node i, whose timestamp was transported along with a message and used to call $HTC.Update()$. Let it also be the case that the physical timestamp component of e is higher than $PC_i(f)$ such that the test in line 24 in Algorithm 2 passes, so that the value of $last\_physical$ is updated to the value of e's physical timestamp component and later assigned as fs physical timestamp component.

If e's timestamp was obtained from the physical clock ($PC_j(e)$) then, due to Assumption 1, the "real" time at which e happened must fall under the interval:

$$t_e^{real} \in [PC_j(e) - E_j(e), PC_j(e) + E_j(e)] \tag{5}$$

We also know that the "real" time at which f must have happened is in the interval:

$$t_f^{real} \in [PC_i(f) - E_i(f), PC_i(f) + E_i(f)] \tag{6}$$

We know that, since $e \rightarrow f$, the "real" time at which e and f occurred can be related by:

$$t_f^{real} > t_e^{real} \tag{7}$$

Since $e \rightarrow f$, from 5 and 7 we can assert that t_f^{real} must have occurred after e's earliest possible value that is:

$$PC_j(e) - E_j(e) < t_f^{real} \tag{8}$$

On the right side of the interval we can assert that t_f^{real} must have occurred before $PC_i(f) + E_i(f)$, that is:

$$t_f^{real} \leq PC_i(f) + E_i(f) \tag{9}$$

Joining 8 and 9 together we get:

$$PC_j(e) - E_j(e) < t_f^{real} \leq PC_i(f) + E_i(f) \tag{10}$$

If we subtract $PC_j(e)$, the value assigned to the physical component of f's timestamp, from the inequality in 10 we get:

$$\begin{aligned} -E_j(e) < t_f^{real} - PC_j(e) \leq \\ PC_i(f) - PC_j(e) + E_i(f) \end{aligned} \tag{11}$$

Since We've selected $PC_i(e)$ as f's physical timestamp, i.e. $(HTC_i(f).physical = HTC_j(e).physical) \wedge (HTC(e).physical = PC_i(e))$ we know, from Algorithm 2 that:

$$PC_i(f) < PC_j(e) \tag{12}$$

From (12) we know that $PC_i(f) - PC_j(e) < 0$, that $HTC_i(f).physical = PC_j(e)$ and we previously defined $t_f^{real} - HTC_i(f).physical$ as $error(f)$ so 11 becomes:

$$-E_j(e) < error(f) < E_i(f) \tag{13}$$

That is the the absolute error on the assignment of $PC_i(e)$ as the physical component of f's timestamp is bound by:

$$error(f) \in (-E_j(e), E_i(f)) \tag{14}$$

□

Notes

[1] In *TrueTime* this server is referred to as the "time-master"

[2] $E_i(e)$, $PC_{ref}(e)$, $PC_i(e)$ are assumed to use the same time resolution, usually milli- or microseconds.

[3] As we'll show later in practice we limit the maximum error of a server by evicting it from the cluster if the error surpasses a maximum, but this is not relevant for this analysis

[4] This is implemented through two phase commit in Spanner

[5] Which Spanner's authors solve simply by having the client choose the "current" timestamp and having servers wait until that timestamp is guaranteed to be in the past.

[6] To prevent malicious clients from manipulating the clock, a production implementation should authenticate the clock values using a scheme such as HMAC. Our research prototype assumes that clients are not malicious.

[7] Although the actual durability semantics of an `fsync()/fdatasync()` call is usually dependent on software and hardware the usually accepted contract is that after such a call all previously written data is durable and will survive a power outage

[8] *Proper fsync support* for HBase is currently in-progress and tracked by https://issues.apache.org/jira/browse/HBASE-5954

[9] A stratum 1 source in NTP terms is a GPS or atomic clock

References

[1] HBase. `https://hbase.apache.org/`.

[2] ALMEIDA, S., LEITÃO, J., AND RODRIGUES, L. ChainReaction: a causal+ consistent datastore based on chain replication. *the 8th ACM European Conference* (Apr. 2013), 85–98.

[3] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.

[4] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W., AND WALLACH, D. Bigtable: A distributed storage system for structured data. *Proceedings of the 7th USENIX Symposium on Operating Systems Research* (Jan. 2006).

[5] COOPER, B., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (Aug. 2008).

[6] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. *the 1st ACM symposium* (June 2010), 143–154.

[7] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally Distributed Database. *Transactions on Computer Systems (TOCS 31*, 3 (Aug. 2013).

[8] DAVIS, J. IBM/DB2 Universal Database: Building Extensible, Scalable Business Solutions., 1999.

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (Oct. 2007).

[10] FIDGE, C. J. Timestamps in message-passing systems that preserve the partial ordering. In *Proceeding of the th Australian Computer Science Communication* (1988).

[11] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (July 1990), 463–492.

[12] KEITH MARZULLO, K. M. Consistent global states of distributed systems: Fundamental concepts and mechanisms. *Distributed Systems* (1993).

[13] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (Jan. 4), 35–40.

[14] LAMB, A., FULLER, M., VARADARAJAN, R., TRAN, N., VANDIVER, B., DOSHI, L., AND BEAR, C. The vertica analytic database: C-store 7 years later. In *Proceedings of the VLDB Endowment* (Aug. 2012), VLDB Endowment.

[15] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978).

[16] LAMPORT, L. Paxos made simple. *ACM SIGACT News* (2001).

[17] LEE, J. W., LOAIZA, J., STEWART, M. J., HU, W.-M., AND WILLIAM H BRIDGE, J. Flashback database.

[18] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (Oct. 2011), 401–416.

[19] LOMET, D., BARGA, R., MOKBEL, M. F., SHEGALOV, G., WANG, R., AND ZHU, Y. Immortal DB: transaction time support for SQL server. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (June 2005), ACM, pp. 939–941.

[20] MATTERN, F. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms* (1989).

[21] MILLS, D. L. Network Time Protocol (NTP). *Network* (1985).

[22] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *OSDI'10: Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Oct. 2010), USENIX Association.

[23] RAO, J., SHEKITA, E. J., AND TATA, S. Using Paxos to build a scalable, consistent, and highly available datastore. In *Proceedings of the VLDB Endowment* (Jan. 2011), VLDB Endowment.

[24] STOLLER, S. D. Detecting global predicates in distributed systems with clocks. *Distributed Computing 13*, 2 (2000), 85–98.

[25] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. *C-store: a column-oriented DBMS*. VLDB Endowment, Aug. 2005.

[26] WEI, Z., PIERRE, G., AND CHI, C.-H. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *Services Computing, IEEE Transactions on 5*, 4 (2012), 525–539.

[27] ZHANG, C., AND DE STERCK, H. HBaseSI: Multi-row Distributed Transactions with Global Strong Snapshot Isolation on Clouds. *Scalable Computing: Practice and Experience 12*, 2 (Jan. 2011).