Beyond TrueTime: Using AugmentedTime for Improving Spanner

Murat Demirbas
University at Buffalo, SUNY
demirbas@cse.buffalo.edu

Sandeep Kulkarni
Michigan State University,
sandeep@cse.msu.edu

Spanner [1] is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. In order to support distributed transactions at global scale, Spanner leverages on a novel TrueTime (TT) API that exposes clock uncertainty. In this paper, we discuss how Spanner's TT-based approach relates to the concepts of causality and consistent cuts in the distributed systems literature. Then, armed with a better understanding of the tradeoffs made by the TT-based approach, we discuss how to eliminate its shortcomings.

We consider two main issues with Spanner's TT-based approach. First, it requires access to special hardware for maintaining tightly-synchronized clocks and minimizing uncertainty in TT. And second, transactions in Spanner are still delayed at commit time to compensate for the remaining TT uncertainty. To eliminate these shortcomings, we propose the use of AugmentedTime (AT), which combines the best of TT-based wallclock ordering with causality-based ordering in asynchronous distributed systems. We show that the size of AT can be kept small and AT can be added to Spanner in a backward-compatible fashion, and as such, AT can be used in lieu of (or in addition to) TT in Spanner for timestamping and querying data efficiently.

1 Brief review of Spanner and TrueTime

Spanner supports general-purpose long-lived transactions and provides a SQL-based query language. Data is stored in semi-relational tables, and is versioned. Each version is automatically timestamped with its commit time by the TT API. Spanner provides externally-consistent (linearizable) reads and writes. Using TT, Spanner assigns globally-meaningful commit timestamps to transactions reflecting the serialization order: *if a transaction $T1$ commits (in absolute time) before another transaction $T2$ starts, then $T1$'s assigned commit timestamp is smaller than $T2$'s.*

TrueTime API. TT.now() returns a TTinterval:[*earliest, latest*] that is guaranteed to contain the absolute time during which TT.now() was invoked. The error bound is denoted as ϵ, which is half of TTinterval's width. Google keeps ϵ less than 6ms by using a set of dedicated time masters (with GPS and atomic clock references) per datacenter and by running a time slave daemon (that polls the masters) per machine.

Spanner implementation. A zone is the unit of administrative deployment, and has 1 zonemaster and 100 to 1000s of spanservers. Zonemaster assigns data to spanservers; spanserver serves data to clients. Each spanserver is responsible for 100 to 1000 tablets. A tablet implements a bag of the mappings: (key:string, timestamp:int64) $\rightarrow$ string.

To support replication, each spanserver implements a Paxos state machine on top of each tablet. At every replica that is a leader, each spanserver implements: a lock table (mapping ranges of keys to lock states) to implement concurrency control, and a transaction manager to support distributed transactions. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since the lock table and Paxos together

provide transactionality. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform 2-phase commit. One of the participant groups is chosen as the coordinator, and its leader is referred to as the coordinator leader.

Read-Write transactions. Spanner supports read-write transactions, read-only transactions, and snapshot reads. Standalone writes are implemented as read-write transactions; non-snapshot standalone reads are implemented as read-only transactions. A snapshot read is a read in the past.

Read-write transactions use 2-phase locking and 2-phase commit. First, the client issues reads to the leader of the appropriate group, which acquires read locks and reads the most recent data. When a client has completed all reads and buffered all writes, it starts 2-phase commit. Read-write transactions can be assigned commit timestamps by the coordinator leader at any time when all locks have been acquired, but before any locks have been released. For a given transaction, Spanner gives it the timestamp s_i =TT.now().latest that the coordinating leader assigns to the Paxos write which represents the transaction commit. To wait out the uncertainty in TT, there is a *Commit Wait*: The coordinator leader ensures that clients cannot see any data committed by T_i until $s_i <$ TT.now().earliest.

Read-only transactions. A read-only transaction executes in two phases: assign a timestamp s_{read}, and then execute the transaction's reads as snapshot reads at s_{read} (without locking, so that incoming writes are not blocked). The simple assignment of s_{read} = TT.now().latest to a read-only transaction preserves external consistency. The snapshot reads can execute at any replicas that are sufficiently up-to-date. Every replica tracks a value called t_{safe}, which is the maximum timestamp at which a replica is up-to-date, and can satisfy a read at a timestamp t if $t \leq t_{safe}$. Each replica maintains $t_{safe} = \min(t^{Paxos}, t^{TM})$, where t^{Paxos} is the timestamp of the highest-applied Paxos write known and t^{TM} is the request timestamp of the earliest prepared but not committed transaction. t^{TM} is ∞ at a replica if there are no prepared transactions.

2 How TrueTime translates to distributed systems concepts

Spanner's TT-based approach and the causality tracking approach of asynchronous distributed systems sit in two extreme opposite ends of the spectrum. The literature on asynchronous distributed systems ignores wallclock information completely (i.e., it assumes an infinite uncertainty interval), and orders events by just tracking logical causality relations between them based on applying these two rules transitively: 1) if events e and f are in the same site and e occurred before f, then e happened-before f, and 2) if e is a sending of a message m and f is the receipt of m, then e happened-before f. Events e and f are concurrent, if both e happened-before f and f happened-before e are false. This causality information is maintained, typically using vector clocks (VCs), at each site with respect to all the other sites. As the number of sites (spanservers) in Spanner can be on the order of tens of thousands, the causality information maintained as such is highly prohibitive to store in the multiversion database, and very hard to query as well.

In the other end of the spectrum, Spanner's TT-based approach discards the tracking of causality information completely. Instead it goes for an engineering solution of using highly-precise dedicated clock sources to reduce the size of the uncertainty intervals to be negligible and order events using wallclock time—provided that the uncertainty intervals of the events are non-overlapping. This wallclock ordering in TT is in one sense stronger than the causal happened-before relation in traditional distributed systems since it does not require any communication to take place between

the two events to be ordered; sufficient progression of the wallclock between the two events is enough for ordering them. However, when the uncertainty intervals are overlapping TT cannot order events, and that is why in order to ensure external consistency it has to explicitly wait-out these uncertainty intervals. Moreover this approach also requires access to GPS and atomic clocks to maintain very tightly synchronized time at each spanserver.

TrueTime, what is it good for? What exactly TT buys for Spanner is not discussed in the Spanner paper. Although most readers may have the impression that TT enables lock-free reads in Spanner, lock-free reads (i.e., read-only transactions) could in fact be implemented without TT: Since read-only transactions are also serialized by coordinating leaders and Paxos groups along with read-write transactions, the client will obtain as an answer the latest (and consistent) version numbers for the data items it requested, and the read can then get these specific versions of the data from the spanservers.

We propose that *TT benefits snapshot reads (reads in the past) the most!* By just giving a time in the past, the snapshot read can get a consistent-cut reading of all the variables requested at that given time. This is not an easy feat to accomplish in a distributed system without using TT and tightly-synchronized time. This would then require capturing and recording causality between different versions of variables using VC, so that a consistent cut can be identified for all the variables requested in the snapshot read. However using VC is infeasible in Spanner as we discussed above. TT provides a convenient and succinct way of encoding and accessing consistent-cuts of the Spanner multiversion database in the past. (We discuss this further in the next section.)

In sum, TT's conciseness and universality are its most useful features. Next we show how AT can achieve the same features without the disadvantages of TT.

3 AugmentedTime: Extending TrueTime with causal information

AT fills the void in the spectrum between TT and causality-tracking in asynchronous distributed systems [2]. When ϵ is infinity, AT behaves more like vector clocks (VCs) [3] used for causality tracking in asynchronous distributed systems. When ϵ is small, AT behaves more like TT, but also combines the benefits of TT with causality tracking in uncertainty intervals.

In the worst case an AT at spanserver j, denoted as $at.j$, is a vector that contains an entry for each spanserver in the system: $at.j[j]$ denotes the wallclock at j and $at.j[k]$ denotes the knowledge j has about the wallclock of spanserver k. Since the wallclocks are strictly increasing, we have $\forall j, k, at.j[k] \leq at.k[k]$. Any message sent by j includes its AT. And, any message received by j updates its own AT to reflect the maximum clock value that j is aware of as in the updating procedure in VC. However, in contrast to VC, $at.j[j]$ is updated by the wallclock maintained at j.

AT reduces the overhead of causality tracking in VC by utilizing the fact that the clocks are *reasonably* synchronized. Although the worst case size for AT is very large, we observe that if j does not hear (directly or transitively) from k within ϵ time then $at.j[k]$ need not be explicitly maintained. In that case, we still infer implicitly that $at.j[k]$ equals $at.j[j]-\epsilon$, because $at.j[k]$ can never be less than $at.j[j]-\epsilon$ thanks to the clock synchronization assumption. [1] Therefore, in practice the size of $at.j$ would only depend on the number of spanservers that communicated with j within the last ϵ time. For spanservers that are far apart (e.g., across continents) from

[1] This approach also helps support dynamically created spanservers.

j, the minimum communication latency will typically exceed ϵ, and hence, $at.j$ would never need to include information for them, but would include information only for *nearby* spanservers that communicated with j in the last ϵ time and provided a fresh timestamp that is higher than $at.j[j]-\epsilon$. Thus $at.j$ will reduce to only one entry, $at.j[j]$, for the periods that do not involve communication with nearby spanservers.

Read/Write transactions with AT. Using AT, a read-write transaction would be executed the same way except that the timestamp assigned to the transaction at the coordinating leader is the leader's AT timestamp instead of a single int64 TTstamp. The size of this AT timestamp will be usually very small, because in many scenarios the extra information that the leader has about other spanservers is likely to be the same as that described by the clock synchronization limit ϵ, and hence, this information is not stored explicitly. AT can readily be integrated to Spanner's tablet mappings of (key:string, timestamp:int64) $\rightarrow$ string. The coordinating leader's own single clock entry (or TT) is entered in the single int64 TTstamp area as before, and the AT itself is included in the value (string) part. This placement makes AT backwards compatible with the TT-based implementation, and also accommodates how AT-based snapshot reads are executed.

Using AT, a transaction would commit without delay: even though the leader clock is not sufficiently advanced (beyond the uncertainty interval), it can leverage the knowledge about clocks of other relevant spanservers for ordering events. Using this causality information captured in AT we have the guarantee that if transaction $T2$ utilizes variables updated by transaction $T1$ then the commit timestamp of $T2$ will still be strictly higher than the commit timestamp of $T1$.

Snapshot reads with AT. Using AT, a snapshot read would also be executed the same way as in TT-based Spanner. For a snapshot read of data items x and y at absolute time t, the client executes the reads at spanservers, say j and k, that are hosting x and y, and that are sufficiently up to date (i.e., updated to at least $t-\epsilon$). Let t_x (respectively t_y) denote the timestamp of the latest update to x (resp. y) before $t-\epsilon$ at j (resp. k). Reading the values of x at t_x and y at t_y give a consistent cut/snapshot, because at time t the values of x and y are still the same as those at t_x and t_y by definition of t_x and t_y.

If x does not have another update within the uncertainty interval of t_x (i.e., within ϵ of $at.j[j]$), then returning t_x works the same in AT as in TT, because wallclock comparison is enough to identify and return t_x in this case. This case should actually cover a large percentage of all snapshot reads in practical deployments, and for this case AT does not introduce any overhead over TT.

If x has another update with timestamp t'_x within the uncertainty interval of t_x, then we use AT comparison to order t_x and t'_x and identify the latest version to return from j. This ordering of t_x and t'_x with overlapping uncertainty intervals was not possible in TT, because TT timestamped using only one clock value: that of the coordinating leader's. That was why TT had to wait-out uncertainty intervals with commit-wait so that it does not need to compare t_x and t'_x with overlapping uncertainty intervals. Using AT, on the other hand, t_x and t'_x are timestamped with their corresponding coordinating leader's AT values at commit. These AT timestamps include entries for spanservers involved in the transactions and include the spanserver hosting x, since x was updated by both transactions. So, using AT comparison (as in VC comparison), we can simply order t_x and t'_x and return the latest one as the value to be included in the snapshot read. Thus even in the handling of this case, the overhead of AT is very small. This same argument applies for y as well, if y has another update within ϵ of t_y.

4 Concluding remarks

A major advantage of the AT-based implementation is that it is wait-free and allows higher throughput. Since TT requires waiting-out uncertainty intervals for the transaction commit, ϵ determines the throughput of read-write transactions on a tablet (actually at a sub-tablet) level. While TT's commit-wait can be overlapped with Paxos communication, the Spanner paper states in the future work section that: "Given that we expect many applications to replicate their data across datacenters that are relatively close to each other, TrueTime ϵ may noticeably affect performance." The AT-based implementation, on the other hand, does not require waiting ϵ out, instead it records finer-grain causality relations within this uncertainty interval. Therefore, the AT-based implementation will not restrict performance and will allow higher throughput.

In return, AT provides a slightly relaxed version of the external-consistency guarantee in TT-based implementation of Spanner. In AT, when a transaction $T1$ commits (in absolute time) before another transaction $T2$ starts, it is still possible to have a overlap between the uncertainty intervals of $T1$ and $T2$. (This is avoided in TT because TT waits-out these uncertainty intervals.) In case $T1$ and $T2$ are causally-related (e.g., $T2$ uses data updated by $T1$), then AT will still give the same guarantee as TT because $T2$'s assigned AT commit timestamp will be bigger than $T1$'s. Otherwise (if $T1$ and $T2$ are not causally-related), then AT will give a slightly relaxed guarantee, and will only ensure that $T2$'s assigned AT commit timestamp will not be smaller than $T1$'s.

A criticism against using AT could be that AT may not be able to capture hidden backchannel dependencies (e.g., an observer uses $T1$'s commit to write to a message queue outside Spanner, another system reads that and starts a transaction $T2$). Even in the presence of hidden backchannel dependencies, the external-consistency guarantee AT provides is still sufficient for consistent snapshots because it is able to capture the latest value of each data item to be included in the consistent snapshot. It is also possible to eliminate the backchannel dependencies problem by introducing client-notification-waits when using AT. Recall that TT uses commit-waits to prevent the backchannel dependencies problem to the extent of reducing the throughput on writes. Instead in AT we can add a client-notification-wait after a transaction ends, so that the hidden dependencies via client backchannels are prevented while the throughput on writes remains unrestricted for Spanner transactions. AT can still avoid commit-waits and another transaction $T3$ can start before the client-notification-time of $T1$ expires (as such $T3$ is guaranteed to be free of hidden-dependencies to $T1$). If $T3$ then uses data written by $T1$ then AT will detect that due to its causality tracking.

As another advantage, AT obviates the need for dedicated GPS or atomic clock references, and can work with NTP servers that provide an ϵ of several tens of milliseconds. Since the size of AT is dynamically bounded by the number of nearby spanservers that this spanserver communicated with within the last ϵ time, the size of AT using NTP will still be reasonably small and manageable.

Acknowledgment. We thank the Google Spanner authors for useful feedback and discussion.

References

[1] J. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[2] S. Kulkarni and Ravikant. Stabilizing causal deterministic merge. *J. High Speed Networks*, 2005.

[3] F. Mattern. Virtual time & global states of distributed systems. *Parallel and Distrib. Algorithms*, 1989.