Comparative Study of Distributed Transaction Architectures

Aditi Tripathi
Carnegie Mellon University,
Pittsburgh
aditit@andrew.cmu.edu

ABSTRACT

Geographically distributed transactional workloads are an emerging problem in distributed databases. Consensus algorithms like Raft do offer totally ordered operations in the context of a single replication group but there is, as of yet, no consensus on how to build fast isolated transactions across multiple consensus groups. The paper delineates several approaches used by 5 different open-source distributed databases, the existing theoretical approaches and compares the latency and consistency models employed by these databases.

Keywords

Distributed Databases, Percolator, Google Spanner, CockroachDB, YugabyteDB, Concurrency Control, Replication, Correctness Anomalies.

1. INTRODUCTION

The paper discusses 5 open source distributed databases: Percolator, Spanner, YugaByteDB, CockroachDB and Calvin. We compare these at a high level to find similarities and differences. The paper considers the absolute most difficult scenario a system might want to operate in, which includes cross-shard transactions with strong consistency and all of this is done in a globally distributed fashion, which means there exists redundancy across multiple datacenters and also among clients that are updating and accessing from everywhere from those data centers.

This is usually not the case for many applications. There are many ways a system could do better, like if transactions are bound to stay inside a single shard we can go much faster; if we relax the consistency model to something like causal we don't need to go through some of the network hops; we can relax durability by committing only inside one datacenter and not blocking for other datacenters to catch up. We can also co-locate clients with their data, so maybe if we are running a site like Craigslist, we can safely assume that all the people from San Francisco are primarily interacting with the San Francisco dataset, so if we run a San Francisco shard then all these San Franciscians interacting with that shard can use a local copy instead of having to go to some other state or country. We can also co-locate shard leaders together, for some of these protocols require that we talk between different shards and if those shard leaders are hosted in different data centers, there's additional cost. If we can manage to site those things nearby each other, we can cut down on the number of round trips. So, there's lot of ways improvements can be made on this worst case scenario and that could make all the difference as far as a system's actual workload goes So keeping in mind that specifically the worst case is being considered, we hereby look for places where optimizations could be applied.

The paper is organized as follows. Section 2 and 3 review the basic terminology and implementation generalities of distributed databases. Section 4 discusses the various database systems. Section 5 summarizes and concludes this paper.

2. Consensus and Consistency Models

Lamport's paper [8] on strong lower bounds for asynchronous consensus states that the number of nodes in the cluster has to be greater than twice the number of failing nodes, ignoring Byzantine faults i.e. $N > 2f$ and that implies we need to have a majority of nodes available and working together in order for us to make progress on consensus. It also states that it takes a minimum of 2 message delays if there are conflicting requests; as is generally the case, to reach consensus. This is practically achievable using, strong consistency consensus protocols. Paxos [17] naively does it in 8 hops or as short as 2 hops with optimizations, ZAB [8] and Raft [10] – do it in 2 network hops or 1 round trip.

A consistency model is a set of histories. A history is a collection of operations, including their concurrent structure. We use consistency models to define which histories are "good", or "legal" in a

system. In a more general parlance, we can refer to smaller, more restrictive consistency models as "stronger", and larger, more permissive consistency models as "weaker". Not all consistency models are directly comparable. Often, two models allow different behavior, but neither contains the other.

Linearizability [11] implies sequential consistency because every history which is linearizable is also sequentially consistent. This allows us to relate consistency models in a hierarchy. We want real systems to satisfy "intuitively correct" consistency models, so that we can write predictable programs. In a distributed system–one in which it takes time for an operation to take place–we must relax our consistency model; allowing some ambiguous orders to happen. The question one might ask then is How far can we go? Must we allow all orderings? Or can we still impose some sanity on the world?

If we assume that there is a single global state that each process talks to; if we assume that operations on that state take place atomically, without stepping on each other's toes; then we can rule out a great many previous states indeed. We know that each operation appears to take effect atomically at some point between its invocation and completion. We call this consistency model linearizability; because although operations are concurrent, and take time, there is some place–or the appearance of a place–where every operation happens in a nice linear order. The "single global state" doesn't have to be a single node; nor do operations actually have to be atomic. The state could be split across many machines or take place in multiple steps–so long as the external history, from the point of view of the processes, appears equivalent to an atomic, single point of state. Often, a linearizable system is made up of smaller coordinating processes, each of which is itself linearizable; and those processes are made up of carefully coordinated smaller processes, and so on, down to linearizable operations provided by the hardware. We can use the atomic constraint of linearizability to mutate state safely. Linearizability guarantees us the safe interleaving of changes. Moreover, linearizability's time bounds guarantee that those changes will be visible to other participants after the operation completes thus prohibiting stale reads. Each read will see some current state between invocation and completion; but not a state prior to the read. It also prohibits non-monotonic reads–in which one reads a new value, then an old one. Because of these strong constraints, linearizable systems are easier to reason about–which is why they're chosen as the basis for many concurrent programming.

3. Terminology

3.1 Transactional Model

Transactional model is a system model where operations (usually termed "transactions") can involve several primitive sub-operations performed in order. It is also a multi-object property where operations can act on multiple objects in the system.

3.2 Snapshot Isolation

In a snapshot isolated (SI) system, each transaction appears to operate on an independent, consistent snapshot of the database. Its changes are visible only to that transaction until commit time, when all changes become visible atomically.

Snapshot isolation is a transactional model. Snapshot isolation cannot be totally available; in the presence of network partitions, some or all nodes may be unable to make progress. For total availability, at the cost of allowing long-fork anomalies, parallel snapshot isolation [5], or (weaker, but more commonly supported) read committed can be considered.

3.3 Strict Serializability

When Herlihy and Wing introduced linearizability [4], they defined strict serializability in terms of a serializable system which is compatible with real-time order. Informally, strict serializability (a.k.a. PL-SS, Strict 1SR, Strong 1SR) means that operations appear to have occurred in some order, consistent with the real-time ordering of those operations; e.g., if operation A completes before operation B begins, then A should appear to precede B in the serialization order. Strict serializability is a transactional model and guarantees that operations take place atomically. It is also a multi-object property: operations can act on multiple objects in the system. Indeed, strict serializability applies not only to the particular objects involved in a transaction, but to the system as a whole. Strict serializability cannot be totally available; in the event of a network partition, some or all nodes will be unable to make progress.

4. Systems under consideration

4.1 Percolator

The first system discussed is Percolator [16] which comes out of the Google paper of 2010 by Peng and Dabek[1]. It's basically an alternative to MapReduce for building things like Google's search index. MapReduce is a great system for doing parallel processing, but it does all of its processing in two big step which means having to burn a lot of time to do all those updates in one big chunk and not being able to do incremental changes. So, percolator was developed to do incremental operations on this offline datastore. It offers ACID transactions which implies snapshot isolation. It has fast read-only transactions with point in time read. It uses two phase commit (2PC) for updates and has relaxed latency requirements, the paper mentions this explicitly saying that they make some optimizations which pushes up client latencies because it's not super important that clients return information right away to the end users.

Google's BigTable is a sparse key-value store and offers atomic read-modify-writes. Its open source equivalent would be Cassandra level transactions or MongoDB using linearizable majority read and write consensus. BigTable however has no multi-key transactions i.e. we can't do updates across 2 different keys in a way that's atomic. Percolator then is a client library that provides such transactions on top of BigTable. The way it does this is by having a service called the time stamp oracle (TSO). A leader can request the TSO to give it a timestamp and it will provide a monotonically increasing number which can be used to figure out the order in which the snapshot operations committed. So, we get one timestamp for the start of our snapshot and one timestamp for our commit and we can then use those to figure out what the right order is, for testing for conflicts when we go for a commit.

The protocol works something like this. We start off a transaction by obtaining a timestamp from the TSO timestamp coordinator. If we are distributed across multiple datacenters, (here we consider 3 of them) then we may have to do a long request to a different datacenter to talk to the TSO leader because there's only one of these things globally. We get our start timestamp and then when we perform reads, we have to read from leaders in order to make sure that we observe current logs and not stale states which requires another round trip between datacenters to agree.

For the writes, we buffer up all of our writes and then perform them in a single pre-write operation. The pre-write goes to go to all of the leaders of different shards, which then updates those different keys, requiring a consensus step. Finally, we get back an acknowledgement from the leaders, and the clients saying that the operation is ok [figure 1].

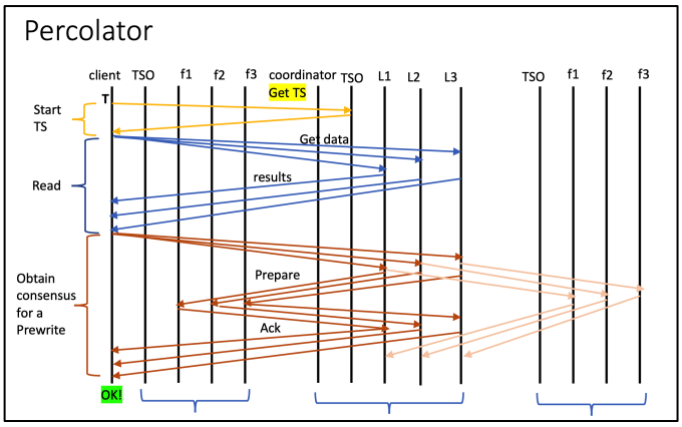

Figure 1: Lamport diagram
(lighter shades indicate redundancy)

Phase 2 is when we need to commit so we talk to the TSO again to get a commit timestamp and then we go ahead and commit the primary record for the transaction. Every shard keeps a point of the primary record which serves as the single point of truth of whether the transaction is committed or not. So, we commit, we do a round of consensus to repose the knowledge, once that's complete the client can return and can tell the client's clients that ok transaction is complete [figure 2].

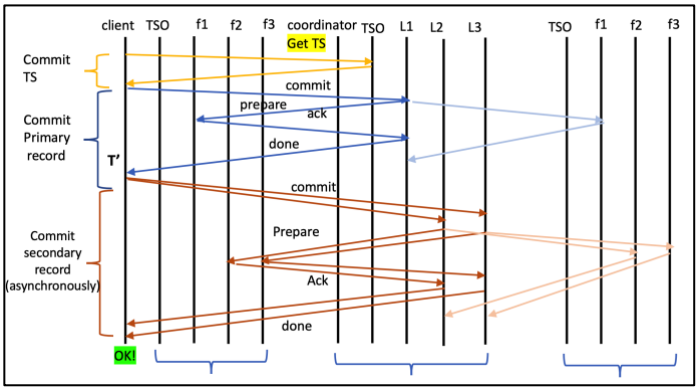

Figure 2

Asynchronously you go and commit the secondary records by promoting them to full-on writes and cleaning up any locks they may have acquired, so this requires a lot of round trips.

If we had all of our leaders co-located and we knew this fact up-front, then maybe you could proxy some operations over to a coordinator in a remote datacenter and cut down on the number of reads that have to go back and forth, so you send over a read request and the coordinator gets your read timestamp and its lease locally and then returns both to you in one pass. The same could be applied to the pre-writes. Not sure if this optimization is done by the production percolator usage. So, percolator naively requires 14 cross-datacenter hops (as counted between **T** and **T'** in figure 1 and 2) and 0 local roundtrips.

4.2 Spanner

Spanner [15] is a globally replicated SQL style online database. Unlike percolator, it is intended to be used by actual people in the real world who expect short response time. Spanner is similar to percolator in the sense it is a versioned database, so it has this notion of a point in time, consistent snapshots, fast reads-writes and also it offers what they call external consistency which is close to strict serializable. It uses TrueTime API instead of TSO which is a library that runs inside of every process and it basically talks to magical clocks which means it's sensitive to clock latency, network skew or any hiccups inside of the process. Google uses a combination of atomic clocks and GPS clocks from multiple vendors. It votes out bad nodes to get rid of any jitter in the data using a complex clock synchronization algorithm.

Spanner basically runs 2PC on top of Paxos, uses timestamps to acquire locks, these locks are taken out on Paxos leaders alone. There is a coordinator Paxos group whose job is to synchronize the committing of the entire transaction. You block to ensure nobody else can allocate a timestamp smaller than the one you are going to allocate for your commit time, that in effect makes sure that your locks are still in effect for anyone else to do concurrent reads and then you can clean up your locks asynchronously. So, the phase where we had to acquire a timestamp in Percolator and do a round trip, that's now 0 operation, since we just know what time it is everywhere all the time to within 7ms or so, based on TrueTime. So you get your start timestamp, you perform your reads, like percolator you have to the leader which could be in a remote datacenter, then you perform your writes,

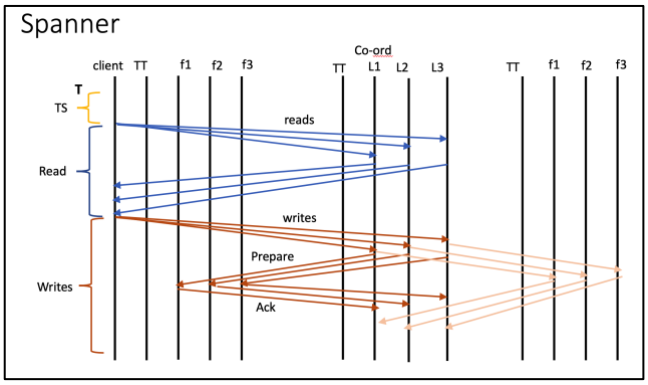

Figure 3

which get batched up, you write to all different shard leaders, they do a phase replication through Paxos. [figure 3]. Once they have completed that, instead of coming back to the client directly, the leaders actually notify the coordinator [figure 4]. We tell the coordinator about the timestamps that we chose when we were proposing our writes, the leader then selects a timestamp that is higher than its local time and also every prepare timestamp. Once that commit timestamp is selected, it does another round of Paxos to make sure that the commit record is durable. You wait for that timestamp to be guaranteed to be in the logical pass for everybody based on clock jitter and then you can return the commit timestamp to the client which completes the transaction and asynchronously apply the transaction and release any locks. So, having access to these clocks really cuts down the number of round trips, and also makes the system dramatically more efficient. Spanner thus takes 8 cross-datacenter hops for a read/write transaction and one local hop.

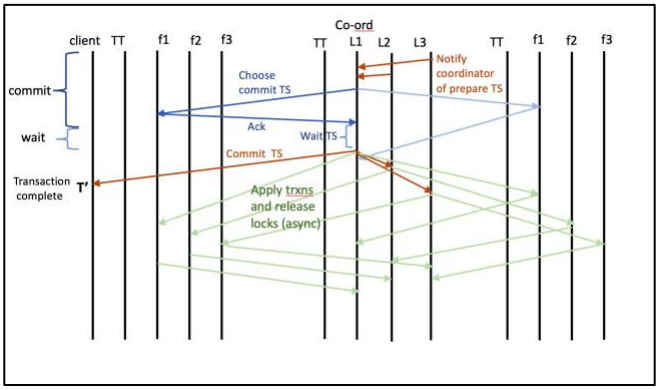

Figure 4

For local reads, since you can pick a timestamp locally and it's guaranteed to be current, you can do a

local read without needing to go across datacenters and get what is a linearizable read in 0 round trips to other datacenters. That's pretty incredible because every other system on the planet currently requires a round-trip to get linearizable reads. So that's a neat advantage of having good clocks.

4.3 YugaByte DB

YugaByteDB [13] is a commercial open source implementation, inspired by Spanner. It uses, based on a more recent paper [7], something called hybrid logical clocks (HLC), which are basically monotonic wall clocks combined with Lamport clocks [6], so you increment them when you replicate clocks between different nodes and also couple this to the RAFT state for each shard, so there is a lot of complex logic going on to make sure that these things actually look monotonic. In essence HLC can identify and return consistent global snapshots without needing to wait out clock synchronization uncertainties and without needing prior coordination, in a posteriori fashion.

Like spanner, Yugabyte uses leader leases to ensure that the leaders are current and not overlapping. Like spanner there is some blocking, they call it safe time, to ensure that things are guaranteed to be visible. Other than the fact that there is a need to compute a timestamp that's not going to intersect with somebody else's operation that are ongoing, it looks pretty similar to Spanner – there's 2PC, a single transaction record and asynchronous cleanup. But unlike Spanner, they make the transaction record an explicit extra phase.

So pragmatically, the system has a SI-CQL (Cassandra Query language) style interface. You also get linearizable single key operation, but because of the structure of CQL there is no generalized

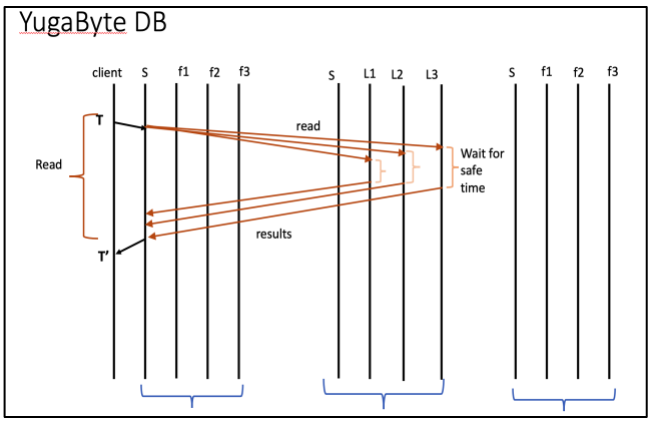

Figure 5

transactions in Yugabyte DB. We can do exactly 2 types of transactions - one of them is a single read operation that could be accomplished on multiple keys and the other one is a batch write where we can write to many records, but it's not possible to do a transaction that reads and writes data or does reads across multiple tables, as those transaction types are just not expressible in CQL. The read-only transaction is fairly straightforward, you go to the leaders, which could be in a different datacenter, you wait for the safe time to elapse and get back results and the transaction is done [figure 5].

For the write [figure 6], however, you send the transaction/operation to a leader of the transaction status record which could be in every data center, so

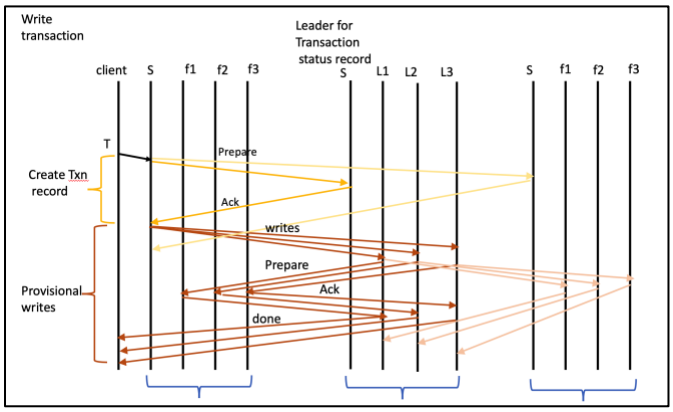

Figure 6

just pick one which happens to have a local leader and that's the one that's going to track the state of the transaction (like the coordinator from Spanner). It does a propose and acknowledge for consensus to confirm that the transaction record exists. At that point the transaction status coordinator goes and performs all of writes to different shards. So, it does a provisional write on each shard, (shards are replicated via raft) and once we get the responses, we know your provisional writes are available on every node. We can then go ahead and propose that transaction to be committed, so we talk to the local transaction status leader again, do a round-trip to make sure that that transaction status record is committed and then we are done [figure 7]. The transaction gets committed and we can return to the client and inform that it's complete. Like Spanner there is an asynchronous clean up phase, so the coordinator then goes to all of the shard leaders and informs them to complete those provisional writes, promote them to real writes and then clean up the transaction status record after all that's done, but those things don't block the client.

Thus, Yugabyte requires something like 10 cross datacenter hops and 4 local hops in order to execute a read and a write transaction.

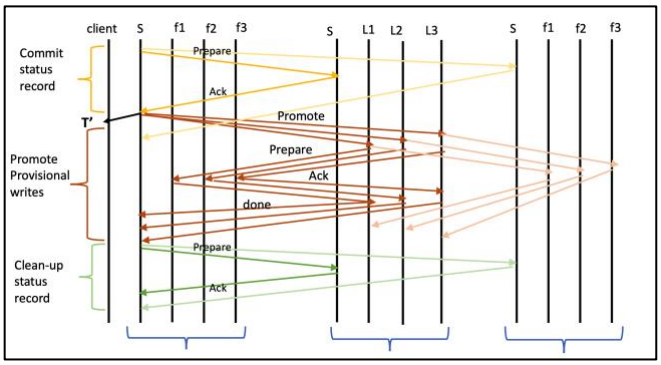

Figure 7

4.4 CockroachDB

CockroachDB [14] is another Spanner inspired database that also uses hybrid logical clocks and offers serializability. It uses Raft for each shard much like YugaByte DB and also leader leases i.e. have a timeout to make sure that no leaders are running concurrently. There is no TrueTime like Spanner. Instead it uses hybrid logical clocks which are basically subject to whatever clocks you have got running locally plus however good your network is in getting clock data back and forth. So that means that the clock skew tolerance has to be significantly higher. So, where Google synchronizes to within 7ms in the 2012 paper and possibly lower now, CockroachDB shows a 250 ms clock skew tolerance to start. Another difference is that Spanner blocks every single write transaction for 7 ms or however long the. current TrueTime boundary is; whereas CockroachDB only blocks read that are contended. So, there is a possibility we might see much higher latencies on reads up to 500ms, but it should only happen if we are reading contended records. One curious thing about CockroachDB is that it uses a one phase commit (1 PC) rather than a 2PC.

For reads, where we talk to a leader in Spanner and Yugabyte DB; in CockroachDB we talk to what's called a "Gateway" which will proxy our operations over to the leaders and in case all of the leaders are in the same datacenter as the gateway, that makes one round trip across datacenter traffic for reads and then for a write which you broadcast to the gateway, the gateway send it to all the shard leaders, the shard leaders do a round of Raft consensus, come back with a response and then we are done. Thus, it takes 6 datacenter hops and 4 local hops to do a distributed read/write transaction.

4.5 Calvin

Calvin [12] comes out of a paper from Yale 2012 by Ren, Shao and Abadi. It is built to do globally distributed serializable transactions. It has couple of really neat insights which are kind of subtle but also make a lot of sense. One of them is that databases spend a lot of time negotiating locks and doing automatic concurrency control in order to make the system look like it's executing in one single order. But if getting the appearance of an order is all we need then maybe we can separate the problem of locking from the problem of ordering; like if we knew what the order of the transaction was upfront, we wouldn't need to communicate any other information other than that. We could just apply the transactions in the same order everywhere, which means our transactions have to be pure and we have to know everything they do upfront. In all the systems we have talked about so far, there are separate phases for doing reads and writes, and if our reads are dependent on previous reads, we may have multiple phases there. In a system like Calvin, our client sends the transaction to the database once and that includes all the reads and writes it's going to execute and then it gets a response which cuts down the round trips we have to do between the client and the database [figure 8].

In Calvin, while we do need Paxos to get order in a fault tolerant way to construct an order of

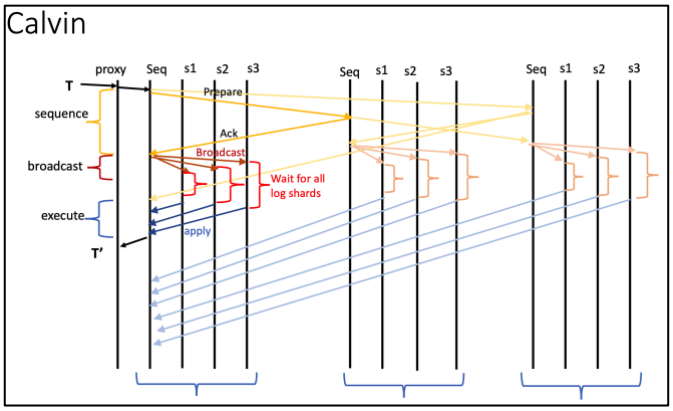

Figure 8

logs, but once we have it there is no need to do Paxos again to get order between log shards, we can just stack the logs on top of each other, and get a total order that's the same everywhere- a result that comes from Lamport's paper [6]. Just make sure that every single node is available to exchange that data. Thus, it's a much less firm constraint to say that we have to have every shard available since we can still tolerate loss of nodes inside any number of shards as

individual “nodes” are fault tolerant because they are actually like Raft replicated groups. The log is then divided into epochs where epochs could be sliced arbitrarily small or large depending on how much overhead we want to tolerate for the coordination and how short we want our latencies to be. The protocol then is remarkably short- we submit our transaction to some proxy node, the proxy will then send the request onto a sequencer which is going to be a leader of the Raft group which stores some shard of the log and as long as you have got enough shards, chances are that we can find one as a leader in the local datacenter so we don’t have to talk remotely to get to the leader. The leader will then do a round of Raft – proposing and getting the acknowledgements back. As a slight optimization the responses are not sent just to the leader; the follower or the other sequencer nodes, will broadcast it to every other sequencer node that they know about, which can improve performance if one of them is down. So, once we have got majority acknowledgement it gets known that operation is durable in the log and is going to be executed. We then wait for the epoch to complete and the epoch is then broadcasted to schedulers, let’s say s1, s2, s3, which store some copy of each shard - s1 is shard 1, s2 is shard 2 etc. [figure 8].

Every datacenter has a full copy of the database divided into shards but with the order of transactions known on every datacenter so all we have to do is evaluate it in the same order as computed by appending the logs together for each epoch. Once the coordinator has got all the information it needs, it can send the results of the transaction back to the client and we are done. That means Calvin can run transactions that are serializable in 2 cross datacenter hops and 5 local hops. All of this is true, assuming no failures. Calvin has no clock dependencies at-least as far as safety goes. None of this requires knowing what time it is in order to preserve transactional integrity, but we do need to block for an epoch to complete so there is some minimum latency floor and some clock sensitivity there. There is also an important limitation that we can’t do interactive transactions i.e. there are some queries that cannot be expressed in Calvin because the transactional language might not support them.

5. Summary and Conclusions

In regard to distributed databases, the paper first explains the concepts of transactions, snapshot isolation, and serialization. It then compares existing systems, noting features such as the number of cross-datacenter hops (CDH) and local round trips (LRT).

Percolator is the system Google uses to incrementally update a large data set, and it is used to build Google’s search index. Key to the approach is a timestamp oracle that provides a Total Store Ordering (TSO) to operations. Percolator performs transactions in two phases. First, it performs reads conforming to the TSO and obtains consensus for a pre-write. Then, it obtains a new commit timestamp and another consensus to commit the write. [14 CDH, 0 LRT].

Spanner uses Paxos for consensus, and instead of a TSO oracle, it uses the TrueTime API. TrueTime explicitly accounts for uncertainty and allows each individual process to know the current “time”. Having access to this clock reduces the number of datacenter hops to perform a transaction compared to Percolator. It uses leader leases to ensure leaders are fresh and do not interfere with each other. [8 CDH, 0 LRT].

YugaByteDB, inspired by Spanner, uses hybrid logical clocks (HLC), which are incremented on replication and coupled to each shard via Raft. This approach requires prior no clock synchronization to return globally consistent snapshots, and it also uses the leader lease approach. [10 CDH, 4 LRT].

CockroachDB, is another database inspired by Spanner, and like YugaByteDB, it uses HLCs. Instead of blocking on write transactions, like Spanner does, CockroachDB only blocks contended reads. A “Gateway” centrally proxies operations to all the leaders, reducing the number of datacenter hops needed. [6 CDH, 4 LRT].

Calvin separates locking from ordering. It does one round of Paxos to construct an ordering, but there is no need to do it again later between shards, as logs can just be stacked on each other. At the end, it uses Raft to achieve a sequential consensus. [2 CDH, 5 LRT].

This latency comparison with respect to network hops for both cross datacenter and local are summarized in Table 1.

Thus, in terms of cross datacenter roundtrips, if those are actually the dominant factor in the latency and not the time spent in blocking or the epoch time, Calvin inspired databases might be the fastest option to choose, assuming no byzantine failures.

System	Cross Datacenter	Local
Percolator	8 (optimized) 14 (naive)	0
Spanner	8	1
Yugabyte DB	10	4
Cockroach DB	6	4
Calvin	2	5

Table 1. Table summary for Latency observations

As far as Linearizable read costs are concerned and summarized in Table 2, Cockroach DB does not support linearizable transactional reads while Spanner with its TrueTime is able to execute this in 0 roundtrips between datacenters.

System	Read cost
Percolator	2
Spanner	0
Yugabyte DB	2
Cockroach DB	NA
Calvin	2

Table 2. Table summary for Linearizable read cost (cross datacenter)

System	Min	Max
Percolator	SI	SI
Spanner	SI (r)	Strict 1SR (External consistency)
Yugabyte DB	SI	SI
Cockroach DB	SI (r)	Serializable+ (with some real time guarantees)
Calvin	SI	Strict 1SR

Table 3. Table summary for Isolation Levels

All databases discussed and summarized in Table 3, support snapshot read and write isolation level at minimum. For strong consistency requirements, going with either Spanner or Calvin inspired databases should be a good choice.

Thus, the paper tries to differentiate databases, based primarily on the network hops and the consistency/isolation levels they provide. This qualitative analysis is also consistent with the CAP theorem, as we see the systems making trade-offs mainly between consistency and availability. The final choice for a database mainly depends on the workload a service is expected to see, and whether the end-user experience, for the most part, can be based on relaxed latency or relaxed consistency.

6. Future Work

Future work includes analyzing more newer databases, having the hardware set-up to test clock errors and perform actual database transactions to discover read/write and causal anomalies.

7. REFERENCES

[1] Peng and Dabek, 2010 Large-scale Incremental Processing Using Distributed Transactions and Notifications (OSDI '10) DOI = https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Peng.pdf

[2] Peter Bailis et al, 2013 Highly Available Transactions: Virtues and Limitations. Proceedings of the VLDB Endowment Vol. 7, No. 3 DOI = https://amplab.cs.berkeley.edu/wp-content/uploads/2013/10/hat-vldb2014.pdf

[3] Berenson et al, 1995 A Critique of ANSI SQL Isolation Levels. DOI= https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf

[4] M.P. Herlihy et al, 1990. Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems. DOI = HerlihyW90/p463-herlihy.pdf

[5] Y. Sovran et al. Transactional storage for geo-replicated systems. DOI = http://www.news.cs.nyu.edu/~jinyang/pub/walter-sosp11.pdf

[6] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. DOI = https://amturing.acm.org/p558-lamport.pdf

[7] Sandeep Kulkarni et al. Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases. DOI = https://cse.buffalo.edu/tech-reports/2014-04.pdf

[8] L. Lamport. Lower Bounds for Asynchronous Consensus. DOI = https://lamport.azurewebsites.net/pubs/lower-bound.pdf

[9] B.C. Reed et al . Zab: High-performance broadcast for primary-backup systems. DOI = https://marcoserafini.github.io/papers/zab.pdf

[10] D. Ongaro et al. In Search of an Understandable Consensus Algorithm (Extended Version). DOI = https://raft.github.io/raft.pdf

[11] https://jepsen.io/consistency/models/linearizable

[12] Kun Ren et al. Calvin: Fast Distributed Transactions for Partitioned Database Systems DOI = http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf

[13] YugabyteDB DOI = https://docs.yugabyte.com/latest/architecture/

[14] CockroachDB DOI = https://www.cockroachlabs.com/docs/stable/architecture/overview.html

[15] Spanner DOI = https://research.google/pubs/pub39966/

[16] Percolator DOI = https://research.google/pubs/pub36726/

[17] Paxos DOI = https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

[18] http://rystsov.info/2018/10/01/tso.html