Totally-Ordered Prefix Parallel Snapshot Isolation

Nuno Faria
nuno.f.faria@inesctec.pt
INESCTEC and U. Minho
Portugal

José Pereira
jop@di.uminho.pt
INESCTEC and U. Minho
Portugal

Abstract

Distributed data management systems have increasingly been using variants of Snapshot Isolation (SI) as their transactional isolation criteria as it combines strong ACID guarantees with non-blocking reads and scalability. However, most existing proposals are limited by the performance of update propagation and stability detection, in particular, when execution and storage are disaggregated.

In this paper, we propose TOPSI, an approach providing a restricted form of Parallel Snapshot Isolation (PSI) that allows partially ordering recent transactions to avoid waiting for remote updates or using a stale snapshot. Moreover, it has the interesting property of making a prefix of history in all sites converge to a common total order. This allows versions to be represented by a single scalar timestamp for certification and storage in a shared store. We demonstrate the impact on throughput and abort rate with a proof-of-concept implementation and the industry-standard TPC-C benchmark.

CCS Concepts. • **Information systems** → **Parallel and distributed DBMSs**; **Database transaction processing**; • **Computing methodologies** → *Distributed algorithms.*

Keywords. transactional consistency, Parallel Snapshot Isolation

ACM Reference Format:
Nuno Faria and José Pereira. 2021. Totally-Ordered Prefix Parallel Snapshot Isolation. In *8th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3447865.3457966

1 Introduction

Transactional isolation and strong consistency are sought in database systems by application developers, as they allow focusing mainly on the application logic, leading to simpler and swifter developments [7]. On the other hand, these guarantees can often negatively impact the performance of data access operations, as concurrency control protocols incur overhead and limit scalability. In fact, many methods used to ensure isolation and consistency guarantees [4] rely on the locking of resources or on aborting operations, affecting both response time and useful throughput.

This problem is even more pronounced on distributed database systems – needed to handle higher loads – where the exchange of multiple messages, remote locking, tight synchronization, among others, are commonly needed for strong guarantees [3]. Besides, database systems deployed in geo-distributed environments must deal with the inherent network latencies, where round-trip times can reach hundreds of milliseconds [20]. Some systems avoid these problems by providing weaker guarantees, such as eventual [18] or causal [11, 21] consistency, which might cause increased application code complexity. *Snapshot Isolation* (SI) [2] has been recognized as a good tradeoff for distributed systems, combining ease of use with potential scalability.

In this paper, we focus precisely on the impact of update propagation and stability in systems providing SI [8]. This is particularly important for geographically distributed systems, where network latency between different sites across multiple data centers is higher. The result is that transactions are delayed or allowed to start with a stale snapshot, leading to false conflicts and aborts. On the other hand, relaxed forms of SI such as *Parallel Snapshot Isolation* (PSI) [15] address this by avoiding that transactions are totally ordered and allowing them to run on concurrent snapshots [15], much like *Transactional Causal Consistency* (TCC) [21].

However, common implementations of relaxed forms of SI are built on the assumption of data partitioning, either by sharding or with the concept of proprietary data, as is the case with Walter and FPSI [9, 15]. Because of this, different sites end up with different views of the transaction history that cannot easily be mapped to a common shared storage. This means that they cannot be applied to increasingly popular systems that disaggregate execution from storage [5, 13, 17, 19]. In addition, providing snapshots to systems with disjoint transaction history points to a more complex timestamp management. For example, Walter's PSI uses vector clocks with the same length as the number of sites in the system, meaning that it is harder to use when sites are added

PaPoC'21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8338-7/21/04…$15.00
https://doi.org/10.1145/3447865.3457966

and removed, and both storage and snapshot computation become more expensive as the number of sites increases.

The main contribution of this work is thus the proposal of TOPSI, for *Totally-Ordered Prefix Parallel Snapshot Isolation*, ensuring PSI while at the same time having history at different sites converge to the same totally ordered sequence of transactions. This makes it possible to store data resulting from a stable prefix of operations in a typical versioned storage system, that assigns a simple scalar timestamp to each item. It does so by relying on just two timestamps, global and local, with the additional advantage that snapshot computation and storage are as simple as in a centralized SI system. We demonstrate the impact of the proposed approach with a proof-of-concept implementation and the industry-standard transactional TPC-C benchmark.

The rest of this work is organized as follows: Section 2 presents background information related to consistency and isolation; Section 3 describes the abstract system model assumed for a distributed data storage system; Section 4 describes the TOPSI algorithm; Section 5 displays some preliminary results that compare the solution with other alternatives; finally, Section 6 discusses the proposal and future work.

2 Background

The strictest isolation levels (e.g. *Strict Serializability*) ensure that a transaction is not affected by the concurrent execution of others, simulating serial execution. This supports a wider variety of application semantics and makes it easy to use. Although they provide the strongest guarantees, they can also have a major impact on performance and scalability. A popular isolation level that provides a comparatively better tradeoff is *Snapshot Isolation* (SI) [2], where a transaction reads from a snapshot of all previously committed transactions, allowing non-blocking reads. Unlike serializable isolation levels, SI cannot prevent *write-skew* conflicts.

Given that SI requires a snapshot to contain the updates of all previously committed transactions, it becomes less doable in distributed databases, as it forces transactions to wait before starting or finishing due to synchronous replication. To avoid this, *Generalized Snapshot Isolation* (GSI) [8] extends SI to allow transactions to start from an older snapshot. In a distributed database environment, there is *Prefix-Constant Snapshot Isolation* (PCSI), an instance of GSI, that states that the acquired snapshot, although possibly still not the latest globally, contains the updates of the transactions already applied in the local site. Although both GSI and PCSI have been used to enable scalable distributed database management systems, they still require total order guarantees when it comes to applying updates, which can increase snapshot age and in turn lead to increased abort rates (see Section 4).

On the other end of the spectrum, we have weaker consistency levels such as *Eventual Consistency* (EC) [18], where different sites can execute transactions without any commit time certification, improving performance and availability. Causal consistency criteria such as *Causal+* [11] or *Transactional Causal Consistency* [21] also solve conflicts after committing, while also guaranteeing that transactions are applied based on their causal order. Remaining data conflicts are later handled with rules ranging from basic *last-writer-wins* [16] to *Commutative Replicated Data Types* (CRDTs) [14], which greatly ease the implementation of complex semantics.

One way to achieve a middle ground between strong and weak consistency is to make sure that only those operations that actually require strong consistency end up paying the price, which is the case with the RedBlue consistency [10]. Briefly, operations that commute are tagged as *blue*, meaning they can be asynchronously replicated to other sites. Operations that do not commute are tagged as *red*, meaning they require synchronization. In systems where most operations are tagged *red*, transaction application becomes mostly sequential, simulating serializability. However, this makes development harder as every operation must be tagged.

Another way to achieve a better tradeoff is to judiciously relax *Snapshot Isolation*, while still keeping the core guarantees and avoiding the need to change applications: Reading from a snapshot and a transaction certification procedure that avoids committing conflicting updates. Namely, *Parallel Snapshot Isolation* (PSI) [15] extends SI to allow different sites to apply different transactions with different orders, as long as they respect causal dependencies. This results in snapshots from different sites to evolve independently, decreasing data staleness. Although it is not as strict as SI/GSI, which could be a downside for some applications, it reduces the need for distributed coordination. There is also the *Fresher Parallel Snapshot Isolation* (FPSI) [9], an implementation of PSI that aims to reduce the staleness of the snapshot read in comparison to Walter. As alternatives to PSI there is the *Causally-coordinated Snapshot Isolation* (CSI) [12] and *Non-Monotonic Snapshot Isolation* (NMSI) [1]. In CSI, causality is ensured based on a transaction's read and write sets, while in PSI it is ensured based on its commit time. NMSI allows a transaction to read versions of data committed after it started, similar to FPSI. *Collaborative Global Snapshot Isolation* (CGSI) [6] ensures that if a transaction T_2 reads the effects of T_1 in some partition, it must see its effects in all the other partitions, in addition to operating under total order guarantees. Unlike SI, however, it enforces this restriction only on transactions that depend on each other, achieving this with read and write sets certification.

3 Model and Assumptions

Figure 1 depicts the system model assumed with a set of physical sites that interact through logically shared *transactional oracle* (TxO) and *key-value store* (KVS). Sites are

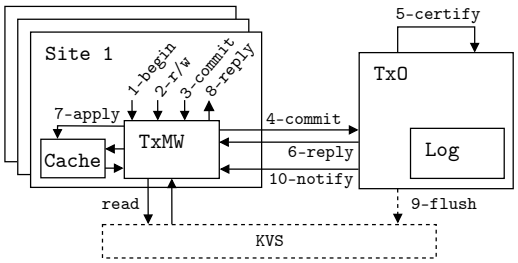


Figure 1. Sites with embedded local *middleware layer* (TxMW) and *cache*, and shared *transactional oracle* (TxO) and *key-value store* (KVS).

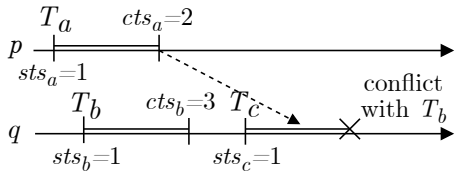
physically distributed and embed a *transactional middleware layer* (TxMW) that mediates all transactional and storage operations, providing isolated transactions. To this end, the middleware makes use of a local *cache*, that can hold items privately to a single transaction or be shared by all transactions in the site.

For simplicity, we assume that the TxO is logically centralized and fault-tolerant, that it totally orders and certifies transactions, and finally ensures that updates are applied to the KVS. Therefore, each site gets a synchronous reply – commit or abort – for each local transaction submitted for certification. It also gets an asynchronous notification of each transaction for which updates are locally visible in the KVS, for both local and remote transactions. For simplicity, we assume that these notifications are issued in commit order.

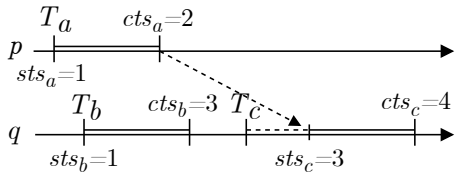
The KVS is assumed to be logically shared and multi-version capable. This means that: All sites read from the same store; the store keeps multiple versions of each item labeled with a scalar timestamp; write operations specify a key, a value, and timestamp; read operations specify a key and a timestamp, that can be $+\infty$, and return the latest value up to that timestamp, if any.

These assumptions map to a range of possible implementations. On the one hand, the TxO can be a centralized server with transactional logging for durability. On the other hand, it can be a replicated component within each site that is made fault-tolerant and persistent with a consensus protocol. Likewise, on one end of the spectrum, the KVS maps to a shared key-value store directly read by all sites. On the other end, it also maps to a private replica of a key-value store for each site, where stability notification corresponds to waiting for updates from remote sites to be received and locally applied by a local representative of a distributed TxO.

A transaction proceeds as illustrated in Figure 1: ① A site starts a transaction by issuing a *begin* request to the local TxMW, which will in turn assign it a consistent snapshot of the data and a unique identifier. ② Next, the site issues reads and writes to the middleware, which resolves reads and caches the writes, ③ followed by a *commit* request when



(a) With GSI: T_c conflicts with T_b, that is visible only after T_a.



(b) With PCSI: T_c is delayed by T_a.

Figure 2. Alternatives when advancing the initial snapshot, with: T_a: $write(x)$; T_b: $write(y)$; T_c: $write(y), write(z)$. The dashed arrow represents update propagation.

done. ④ The *commit* request is promptly forwarded to the TxO, together with the transaction's write-set. ⑤ The TxO then orders and certifies the transaction against previously committed operations, ensuring durability in case of success, ⑥ after which it replies to the originating TxMW. ⑦ Following the certification response, the write-set can be made locally visible from the cache with the appropriate snapshot for future read operations and ⑧ completion signaled to the application. ⑨ Finally, the TxO asynchronously *flushes* the data to the KVS, and ⑩ *notifies* each site.

4 The TOPSI Algorithm

Figure 2 illustrates the motivation for the TOPSI approach. In it, two sites are represented: p, that executes transaction T_a; and q, that executes T_b and T_c. sts and cts denote transaction start and commit timestamps, respectively, for each transaction. In detail, a system aiming at *Snapshot Isolation* (SI) in a distributed system, in which there is a delay for stabilizing updates until they become available in sites other than their origin, resorts to the weaker *Generalized Snapshot Isolation* (GSI)[8] and allows assigning an older snapshot to starting transactions. As Figure 2a shows, this is problematic when the same item is accessed by successive transactions in the same site. In this case, the update to y by T_b does not become visible at q until updates from p for T_a have become stable, because T_a precedes T_b in the total order. Therefore, if T_c updates y again, it will be seen as conflicting with T_b and leads to an abort, as the commit timestamp of the latest update to y, $cts_b = 3$, is greater than the start timestamp of T_c, $sts_c = 1$.

This approach is particularly bad as q in T_c would not read its own writes from T_b. It is also expected that successive transactions in the same site read and update the same

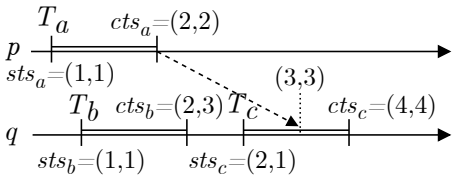


Figure 3. Example of local and global timestamps in TOPSI with: T_a: $write(x)$; T_b: $write(y)$; T_c: $write(y), write(z)$. The dashed arrow represents update propagation.

items, due to locality. Therefore, the alternative is to delay new transactions until the stable snapshot includes the most recent local transaction, as required by *Prefix-Constant Snapshot Isolation* (PCSI) and shown in Figure 2b. In this case, the start of T_c is delayed until updates for T_a, that precedes T_b in total order, have become stable. Note that to ensure that a site always reads its own writes, this would be needed even if T_c was a read-only transaction. In this case, $sts_c = 3$, which is not earlier that the latest update to y done by T_b with $cts_b = 3$, hence it commits.

Our approach stems from the observation that *the problem is the interleaving of local and remote transactions in a global total order, as local writes, even if locally cached and available for reading, are delayed waiting for a potentially lengthy global update and stability protocol.*

Note that local pending transactions (e.g., T_b) and their remote dependencies (e.g., T_a) are known to modify disjoint sets of data items, as they are concurrent and have both been accepted by certification. *Parallel Snapshot Isolation* (PSI) [15], initially proposed in the context of partitioned data stores, would allow T_b to become visible for T_c as long as items updated by concurrent transactions are statically allocated to different partitions (see Section 6).

The challenge is thus to enforce PSI in a system where data is not statically partitioned and assigned to different sites, but globally shared and accessible to all.

Our main insight is that *we can have an alternative local order of the suffix of transactions whose updates are not yet stable, consistent with the causal order allowed by PSI.* Namely, we reorder transactions such as T_a and T_b at site q – taking T_b before T_a – while keeping them in the globally agreed total order – T_a before T_b – elsewhere. This is possible as (1) the local ordering is transient and is eventually reconciled with the global order allowing a shared KVS; and (2) it needs only minimal changes to certification in the TxO, without static data partitioning or vector timestamps. The proposed approach works as follows:

Global and local timestamps. The TxMW in each site keeps two timestamps: *global_t* is incremented whenever the TxO notifies a new stable transaction, thus representing the latest snapshot that can be safely read from the KVS. With GSI, this would be used to assign the starting timestamp to locally initiated transactions. *local_t* is incremented when the TxO replies successfully to a local transaction and also when the TxO notifies that a remote transaction has become stable (but not when a local transaction becomes stable). Pairs of local and global timestamps are used for transaction start and commit timestamps (*sts* and *cts*). In particular, the commit timestamp is formed by the recently incremented *local_t* and the global timestamp provided by the TxO. For a timestamp t, we denote the first local component as $t.local\_t$ and the second global component as $t.global\_t$.[1]

Figure 3 illustrates this with the previous example. Both sites start with $local\_t = global\_t = 1$. At site q, T_b is assigned $sts_b = (1, 1)$. As T_b is certified by the TxO with global timestamp of 3 (after T_a), *local_t* is incremented to 2 and T_b is locally assigned $cts_b = (2, 3)$. When T_c starts, it gets $sts_c = (2, 1)$. Eventually, the notification that T_a is stable arrives to site q and increments *global_t* to 2, and then, upon notification of T_b stable, to 3. When T_c commits, it gets $cts_c = (4, 4)$. Commit timestamps are applied to the corresponding updates in the shared cache, making them available to future local transactions.

Reading and writing. While a transaction executes, an item is initially written to the local cache and kept privately to the transaction by the TxMW. It is later sent to the TxO for certification and, if it succeeds, to be asynchronously applied to the KVS.

When reading an item, the TxMW first looks in the transaction's cache for values already written by it. If not found, it looks in the shared cache for item r tagged with the highest $r.local\_t \leq sts.local\_t$, but with $sts.global\_t \leq r.global\_t$. The first condition enforces the reading snapshot regarding new versions committed by recent local transactions, ensuring that the transaction does not read a value that is in its future. The second condition enforces the reading snapshot regarding new versions committed by remote transactions, by avoiding reading too far in the past just because it is the most recent in the cache.[2] Therefore, if still not found, it looks in the KVS for items tagged with the highest $r.global\_t \leq sts.global\_t$.

Using again Figure 3 as an example, consider the operations issued by T_c that has $sts_c = (2, 1)$ and assume that T_c reads items y and z before updating them. In this case, when reading y, it would be found in the shared cache, as a result of T_b, tagged with $cts_b = (2, 3)$, i.e., compatible with $sts_c = (2, 1)$. As item z is not found in the shared cache, it is retrieved from the KVS with timestamp up to 1.

[1]Note that a site that does not locally execute transactions will simply observe the global total order with $cts.local\_t = cts.global\_t$. Likewise, if stability notification is instantaneous, all sites will also always observe $sts.local\_t = sts.global\_t$.

[2]Interestingly, when $sts.global\_t = r.global\_t$ the value read from the cache is the same as from the KVS, thus we prefer the cache.

To underline the relevance of the second condition for reading from the cache, assume that much later, some transaction T_d at p updates y and then after that value is stable a further transaction T_e at q reads y again. Without the second condition, T_e might still get y from the cache as resulting from T_b, even if a much more recent value for y from T_d is in the KVS.

Submission to the TxO. Upon commit, the TxMW submits the transaction's write-set to the TxO. The TxO knows only global timestamps, thus the TxMW has to provide them for certification. For items that exist in the local cache, the corresponding global timestamp can be efficiently retrieved from there. Otherwise, they are tagged with $sts.global\_t$, as we are sure that there are no concurrent transactions that have produced versions more recent than those in the KVS but older than $sts.global\_t$ in the snapshot. This is also efficient, as it avoids an additional round-trip to the KVS.

Using the example in Figure 3, the write-set for T_c is as follows: y with global timestamp 3, as retrieved from the shared local cache where it was inserted by T_b with timestamp $cts_b = (2, 3)$; and z with timestamp 1, according to the $sts_c.global\_t$ of T_c.

Certification. Aside from having to consider a timestamp for each item, certification by the TxO is mostly the same as for SI, by checking if the newly arrived write-set intercepts in space and time with previously committed transactions. Briefly, it aborts the transaction if any of the modified items has a newer version than what is indicated in the write-set. Otherwise, it commits the transaction, increments the global counter, makes the write-set and timestamp durable,[3] and returns the timestamp to the originating site. The TxO will then asynchronously flush the write-set to the KVS and, when done, notify all sites.

The certification of T_c in the example of Figure 3 would happen as follows: The provided timestamp 3 for y would be compared to 3 – from the point-of-view of the global total order in the TxO is the latest change to y – and accepted. The timestamp of 1 for z would be compared to 1 – not recently modified – and also accepted.

Assume now that T_c also modifies x. As x was not in the cache of site q, x would be submitted with a version of 1 as it has to be retrieved from the KVS by q according to the $sts_c = (2, 1)$. However, this would conflict with T_a, having committed a change to x with global timestamp 2.

Garbage collection. Items can eventually be removed from the cache, that is used for reading according to $local\_t$, as soon as they are included in the prefix of history for which local and global ordering are equivalent. In detail, an item r can be discarded if: $r.global\_t \leq global\_t$; and for all active transactions $r.global\_t \leq sts.global\_t$. The first condition means that the item is no longer useful to a transaction that is about to start. The second means that it is no longer useful for transactions that are already running.

Consider item y inserted in the cache of site q after T_b in the example of Figure 3. At first, it cannot be removed as its global timestamp of 3 is higher than the $global\_t$ of site q, which is still 1. Note that if site q would remain quiescent, it would eventually receive notifications from the TxO and discard y from the cache.

After T_c has started, y cannot be removed from the cache as its global timestamp of 3 is after $sts_c.global\_t = 1$. This is particularly important as q receives updates and increments its $global\_t$ to 3 while T_c is executing. If not, T_c might read both versions 3 and 1 (according to its $sts_c.global\_t$) of item y. After T_c completes, both conditions are false and y can be removed from the cache by site q.

As for the KVS, it only needs to keep an object r if it is the most recent version or if $r.global\_t \geq$ the minimum $global\_t$ of any active transaction in the system, for which a global stability detection protocol is needed.

5 Preliminary Results

To assess the impact of TOPSI, we build a proof-of-concept implementation that can be configured as offering:

- **`TOPSI`** – independently evolving snapshots implemented with the usage of the $global\_t$ and $local\_t$ timestamps;
- **`GSI`** – snapshots evolve using total commit order in all sites, although transactions may start without all local predecessors being visible;
- **`PCSI`** – same as GSI, with the additional restriction that the snapshot contains all preceding local transactions;
- **`SI`** – snapshots evolve using total commit order and at the same time[4] in all sites; a snapshot contains all preceding transactions globally applied.

The implementation uses C++ for middleware and TxO (gcc-7.5.0) and PostgreSQL 12 as a database engine. As a workload, we use the TPC-C benchmark[5] with a scale factor of 32 warehouses. The tests are executed on Google Cloud Engine virtual machines (Ubuntu 18.04 LTS), with client (4 N1 vCPUs), sites (4 N1 vCPUs, 8GB RAM, 100GB persistent SSD) and the centralized TxO (4 N1 vCPUs, 8GB RAM, 100GB persistent SSD) deployed on different instances in the same local network. Different numbers of client threads are tested, with each test running for three minutes (average of three runs). In the multiple site tests, both client threads and TPC-C warehouse access is evenly shared across all sites (e.g. with 16 clients, the first site processes clients 1-4, and clients 1-4 only access the warehouses 1-8), to simulate applications with data locality access patterns.

[3] Depending on the actual implementation of the TxO, this can be done implicitly by a consensus protocol or explicitly with a centralized log.

[4] Due to message delay, two different sites can differ, at most, in one transaction application.

[5] https://github.com/Percona-Lab/sysbench-tpcc/

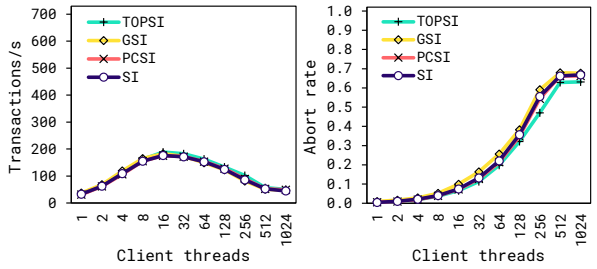


Figure 4. Throughput (left) and abort rate (right) of the different isolations with one site.

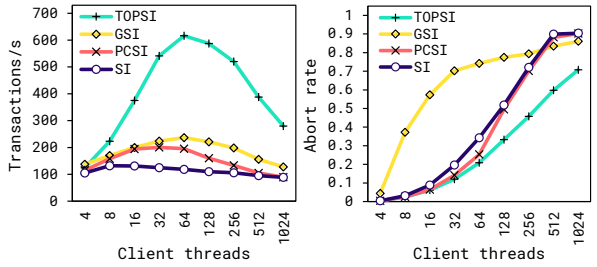


Figure 5. Throughput (left) and abort rate (right) of the different isolations with four sites.

To better understand the different methods, two deployments are considered: single and multiple (four) sites. Figure 4 shows that with a single site there is not much difference between the different isolation criteria. This is expected, as all transactions are local and there is no update delay. Figure 5 shows that increasing the number of sites leads to reduced throughput with `SI` and similar with `PCSI`, due to blocking. `GSI` also has reduced useful throughput, but due to a large number of aborted transactions. In contrast, by reducing the impact of update delays, `TOPSI` scales much better than the alternatives, and betters exploits the additional sites. In short, enforcing total order between recent local and remote transactions leads to higher delays (`PCSI`, `SI`) or higher abort rates (`GSI`) in a distributed environment, which are avoided by `TOPSI`.

6 Discussion

The proposed approach, TOPSI, focuses on reducing the impact of waiting for remote updates in *Snapshot Isolation*, while at the same time ensuring that transaction history as seen by all sites shares a common totally ordered prefix. This combination should benefit data management systems in a geo-distributed setting and with shared or disaggregated storage systems, such as Amazon's Aurora [17]. Our preliminary evaluation shows that TOPSI results in significant performance improvements as the number of sites grows.

An interesting aspect of TOPSI is that it does not need a static partitioning of data items: Each site is allowed to execute transactions reading and writing any item. It is however often the case that applications, especially geo-distributed ones, exhibit access locality. Accessing the same item in successive transactions in the same site is precisely the worst-case scenario for systems based on GSI, as it leads to false conflicts and aborted transactions. In contrast, this is precisely the scenario that is solved by TOPSI, which performs best when there is access locality, without the need for any static configuration.

Moreover, it is interesting to compare TOPSI to Walter [15], as it also provides PSI. In contrast to TOPSI, Walter assumes a statically partitioned system where each site is responsible for the certification of a subset of data. It uses vector clocks to obtain consistent snapshots and certify operations, each index representing the number of transactions applied in the snapshot of each site. Besides the additional complexity of certification based on vector timestamps, the same object in different sites does not have a common timestamp making Walter's solution not viable in shared storage systems.

By offering PSI, TOPSI can be used with the same range of applications as Walter, with the same ease of use. However, TOPSI provides a restricted form of PSI, opening up the research question of whether formal specification of the isolation criteria offered by TOPSI can be shown to be genuinely stronger than PSI, namely, as TOPSI always applies remote transactions based on their global commit order while PSI does not. Moreover, an implementation of TOPSI in a more complete data management system is needed to experimentally evaluate it in depth. Finally, despite presenting TOPSI with a sequential TxO for simplicity, it should be possible to further improve it with internal parallelism.

Acknowledgments

Partially funded by project AIDA – Adaptive, Intelligent and Distributed Assurance Platform (POCI-01-0247-FEDER-045907) co-financed by the European Regional Development Fund (ERDF) through the Operacional Program for Competitiveness and Internationalisation (COMPETE 2020) and by the Portuguese Foundation for Science and Technology (FCT) under CMU Portugal.

References

[1] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, 163–172.

[2] Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Sigmod Record*, Vol. 24. 1–10. https://doi.org/10.1145/568271.223785

[3] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[4] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[5] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder-A Transactional Record Manager for Shared Flash.. In *CIDR*, Vol. 11.

9–20.

[6] Prima Chairunnanda, Khuzaima Daudjee, and M Tamer Özsu. 2014. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *Proceedings of the VLDB Endowment* 7, 11 (2014), 947–958.

[7] Mike Curtiss. 2020. *Why you should pick strong consistency, whenever possible.* Retrieved 2021-01-11 from https://cloud.google.com/blog/products/gcp/why-you-should-pick-strong-consistency-whenever-possible

[8] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. 2005. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 73–84.

[9] Masoomeh Javidi Kishi and Roberto Palmieri. 2020. On Reading Fresher Snapshots in Parallel Snapshot Isolation. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1205–1206.

[10] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 265–278.

[11] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 401–416.

[12] Vinit Padhye and Anand Tripathi. 2012. Causally coordinated snapshot isolation for geographically replicated data. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, 261–266.

[13] Kazunori Sato. 2012. An inside look at google bigquery. *White paper, URL: https://cloud.google.com/files/BigQueryTechnicalWP.pdf* (2012).

[14] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[15] Yair Sovran, Russell Power, Marcos Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP'11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 385–400. https://doi.org/10.1145/2043556.2043592

[16] Robert H Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)* 4, 2 (1979), 180–209.

[17] Alexandre Verbitski et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052. https://doi.org/10.1145/3035918.3056101

[18] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[19] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 449–462.

[20] WonderNetwork. 2020. *Global Ping Statistics.* Retrieved 2020-07-24 from https://wondernetwork.com/pings

[21] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*. 75–87.