Distributed transactional reads: the strong, the quick, the fresh & the impossible*

Alejandro Z. Tomsic
Sorbonne Université, Inria, LIP6, Paris

Manuel Bravo
IMDEA Software Institute, Madrid†

Marc Shapiro
Sorbonne Université, Inria, LIP6, Paris

18 September 2018

Abstract

This paper studies the costs and trade-offs of providing transactional consistent reads in a distributed storage system. We identify the following dimensions: read consistency, read delay (latency), and data freshness. We show that there is a three-way trade-off between them, which can be summarised as follows: *(i)* it is not possible to ensure at the same time order-preserving (e.g., causally-consistent) or atomic reads, Minimal Delay, and maximal freshness; thus, reading data that is the most fresh without delay is possible only in a weakly-isolated mode; *(ii)* to ensure atomic or order-preserving reads at Minimal Delay imposes to read data from the past (not fresh); *(iii)* however, order-preserving minimal-delay reads can be fresher than atomic; *(iv)* reading atomic or order-preserving data at maximal freshness may block reads or writes indefinitely. Our impossibility results hold independently of other features of the database, such as update semantics (totally ordered or not) or data model (structured or unstructured). Guided by these results, we modify an existing protocol to ensure minimal-delay reads (at the cost of freshness) under atomic-visibility and causally-consistent semantics. Our experimental evaluation supports the theoretical results.

*Preprint from 19th International Middleware Conference, 10–14 December 2018, Rennes, France

†Work partially done as a student at Université Catholique de Louvain and Universidade de Lisboa

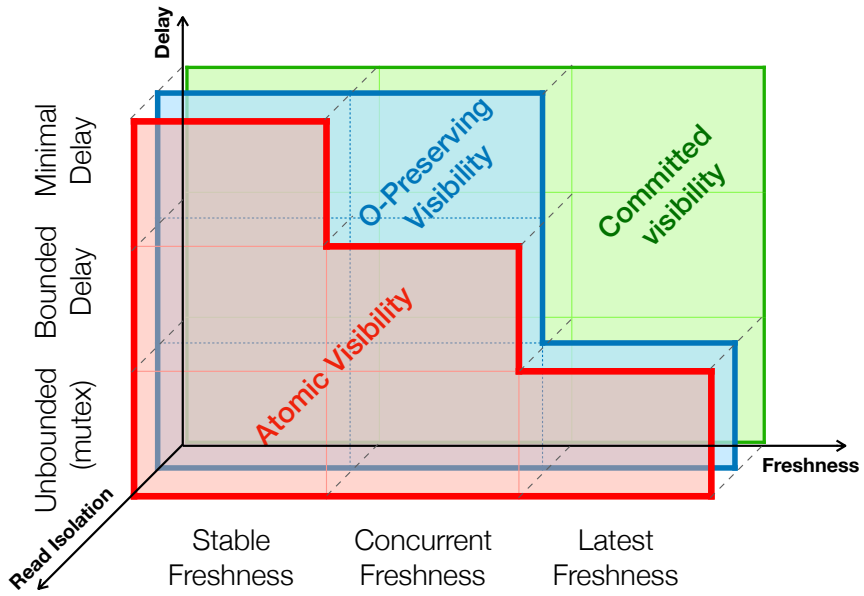


Figure 1 The three-way trade-off. The boxed areas represent possible guarantee/read delay/freshness combinations. Upwards and right is better performance; guarantees are stronger from back to the front planes.

1 Introduction

This paper studies the costs of reading data in a distributed, transactional storage system. In particular, we try to understand whether it is possible to provide strong read guarantees while ensuring both fast performance and fresh data. Intuitively, stronger guarantees will come with higher costs. A recent paper from Facebook (whose performance is strongly read-dominated) states:*"stronger properties have the potential to improve user experience and simplify application-level programming [... but] are provided through added communication and heavier-weight state management mechanisms, increasing latency [...] This may lead to a worse user experience, potentially resulting in a net detriment"* [8]. Is this wariness justified, i.e., is it inherently impossible to combine fast reads with strong guarantees, or can the situation be improved by better engineering? This paper provides a formal and operational study of the costs and trade-offs. Our main finding is that there is a three-way trade-off between read guarantees, read delay (and hence latency), and freshness, and that some desirable combinations are impossible.

It is well known that non-serialisable models, such as Snapshot Isolation [15] or Highly-Available Transactions [13], can improve availability and performance. Therefore, in this paper, we do not necessarily assume that updates are totally ordered.[1] Furthermore, we allow weakening the read

[1] Enforcing a total order of transactional updates enables Consistency under Partition

guarantees: in addition to Atomic Visibility, offered by classical transactional models, we also consider (the weaker) Order-Preserving Visibility, enforced by recent causally-consistent systems [37], and (the weakest) Committed Visibility, offered by Read Committed isolation [11].

Finally, we consider the *freshness* dimension, because (as we show) decreasing the read delay sometimes forces to read a version of the data that is not the most recent. Reading outdated data further incurs multi-version overhead and, under strong consistency, increases aborts and reduces throughput [42].

Figure 1 illustrates the three-way trade-off between the guarantees, delay, and freshness of transactional reads. For instance, under Order-Preserving and Atomic Visibility, it is possible to read with no extra delay (compared to a non-transactional system), but then the freshest data is not accessible. Transactions that access the freshest data with no extra delay are only possible under Committed Visibility. However, we show that minimal-delay Order-Preserving reads allow observing updates of concurrently-committed transactions. As we will see in our evaluation section, this allows for a significant freshness improvement over Atomic Visibility, which forces reading data that was stable (written and acknowledged) before the transaction sends its reads. If, on the other hand, the application requires the freshest data, under either Atomic or Order-Preserving Visibility, this is possible only under a protocol where reads and writes are mutually exclusive, e.g., a read might be delayed (blocked, or in a retry loop) indefinitely by writes, or vice-versa.

This work includes the following contributions:

- A formal study of the trade-offs between the read guarantees, delay, and freshness of transactional reads. We prove which desirable combinations are possible and which are not.
- Guided by the results of our analysis, we modify an existing system that exhibits delays and derive three novel minimal-delay protocols. Each protocol trades freshness for isolation differently. AV provides Atomic Visibility and Stable Freshness, OP provides Order-Preserving Visibility and Concurrent Freshness, and CV Committed Visibility and Latest Freshness. We provide detailed protocol design, including pseudo-code.
- An evaluation of these protocols to empirically validate our theoretical results. In our measurements, we compare the introduced protocols to the protocol from which they derive. Our minimal-delay protocols exhibit similar latency. CV always observes the most recent data, whereas freshness degrades negligibly for OP, and severely under AV.

(CP); but, conversely, Availability under Partition (AP) requires accepting concurrent updates [45].

2 Model and Requirements

In a typical distributed application deployment, we are interested in the database (data storage) tier. The database is distributed, i.e., data is partitioned across many servers within a data centre. Each partition is possibly replicated across several data centres. Clients, representing application logic, access data by contacting the appropriate servers within the data centre. To increase performance, a multi-read/multi-write interface enables the application to access multiple database servers at once in parallel. multi-write operations respectively.

2.1 Transactions

The application consists of *transactions*. A transaction consists of any number of reads, followed by any number of writes, and terminated by an abort (writes have no effect) or a commit (writes modify the store; in what follows, we consider only committed transactions). The transaction groups together low-level storage operations into a higher-level abstraction, with properties that help developers reason about application behaviour.[2] These are often summarised as the ACID properties (Atomicity, Consistency, Isolation, and Durability).

Atomicity ensures that, at any point in time, either all of a transaction's writes are in the store, or none of them is. This guarantee is essential for ensuring some common data invariants, such as equality or complementarity between two data items (e.g., the symmetry of the friendship or the like relationship in a social network application [17]). They are also instrumental to keep materialised views, e.g., when adding a friend to a friends list, by updating a friends count instead of computing this count on reads [14]. *Consistency* is the requirement that each of the application's transactions individually transitions the database from a valid state to another valid state. *Durability* means that later transactions will observe the effect of this transaction after it commits. *Isolation* characterises non-interference between concurrent transactions.

In the interest of performance and availability, we do not assume strong consistency or isolation (e.g., serialisability). Specifically, neither writes nor reads are necessarily totally ordered, and we consider several read guarantees. Interestingly, our results are independent of the write model, e.g., totally ordered or not. Atomicity and durability will be taken for granted in the rest of this paper.

[2] A model that does not support transactions is identical to one where each individual read or write operation is wrapped in a transaction that commits immediately.

For simplicity we assume that a transaction reads a data item at most once (and similarly for writes). The set of item states read by a transaction is called its *snapshot*. Our study distinguishes some important properties of a snapshot, explained in the next few sections: snapshot guarantees, delay, and freshness.

2.2 Snapshot guarantees

Snapshot guarantees constrain the states of the data items that can be accessed by a given snapshot. The stronger guarantees provide higher isolation, and thus facilitate reasoning by the application developer. As we shall see, the weaker ones enable better performance along the freshness and delay metrics. We distinguish three levels, which will be defined formally later (in Section 3.1): Committed, Order-Preserving, and Atomic Visibility.

At the weakest level, **Committed Visibility**, a snapshot may include any updates that have been committed. As it sets no constraints between items, it allows many anomalies. Committed Visibility is equivalent to the read guarantees of Read Committed isolation [11].

Recently, systems have proposed causally-consistent-snapshot reads [37], which strengthen Committed Visibility by ensuring there are no happens-before-order [7] anomalies among the item states included in the snapshot. We consider **Order-Preserving Visibility**, a generalisation of this property. Order-Preserving Visibility ensures that the snapshot preserves *some* (partial or total) order relation O. O might be the (partial) happens-before order, or the total order of updates in the context of a strong isolation criterion such as Serialisability or Snapshot Isolation [15, 16].

Order-Preserving Visibility ensures that transactions do not observe gaps in a prescribed (partial or total) order relation. Consider, in a social network, the data items *photos* and *acl* representing user Alice's photo album and the associated permissions. The set of their states (initially $photos_0$ and acl_0) follows a given order $\rightarrow$. Alice changes the permissions of her photo album from public to private (new state acl_1), then adds private photos to the album (state $photos_2$). Thus, $acl_0 \rightarrow acl_1 \rightarrow photos_2$. Unlike Committed Visibility, Order-Preserving Visibility disallows the situation where Bob would observe the old permissions (acl_0) along with the new photos ($photos_2$), missing out on the restricted permissions (acl_1). This pattern, where the application enforces a relation between two data items by issuing updates in a particular order, is typical of security invariants [45]. It also helps to preserve referential integrity (create an object before referring to it, and destroy references before deleting the referenced object).

Atomic Visibility is the strongest. It is order-preserving, and additionally disallows the "read skew" [15] (also known as broken reads [14])

phenomenon: if the transaction reads some data item written by another transaction, then it must observe all updates written by that transaction (unless overwritten by a later transaction). In a system where atomic updates ensure Alice is in Bob's friends list if and only if Bob is in Alice's, Atomic Visibility ensures a transaction will observe both updates, or none. It also ensures observing materialised views consistently [14]; for instance, observing that the cardinality of a friends list updated atomically with a friends count match.

Atomic Visibility is the read guarantee provided by Serialisability, Cursor Stability and Repeatable Reads ANSI isolation [11], of every strongly-consistent (totally-ordered) criteria (e.g., Parallel Snapshot Isolation [46], Non-Monotonic Snapshot Isolation [42], Snapshot Isolation [15], Update Serialisability [32], and Strict Serialisability [29]), and the weakly-consistent Transactional Causal Consistency [9, 38], and Read Atomicity [14].

2.3 Delay

Perceived low latency keeps users engaged and directly affects revenue [26, 36, 43]. Furthermore, read latency is an important performance metric for services that are heavily read-dominated, such as social networks. For instance, serving a Facebook page requires several rounds, where each round reads many items, and what is read in one round depends on the results of the previous ones; this amounts to tens of rounds and thousands of items for a single page [17]. For read-intensive applications, it is paramount to avoid read delays [8]. Under existing systems, servers often delay a response to a read request. Read delay scenarios include wating for a lock to be released, for physical clocks to catch up, or for a protocol (such as Two-Phase Commit [34]) to finish execution.

The fastest protocol exhibits **Minimal Delay**: distributed reads can address multiple servers in parallel, and any one server always responds with commited data *immediately*, i.e., in a single round-trip, without blocking or coordinating with other servers. Intuitively, this design makes it difficult to ensure strong snapshot guarantees. Examples include Linkedin's Espresso [41] and Facebook's Tao [17]. We will consider Minimal-Delay as our baseline, and will characterise protocols by estimating the *added delay* above this baseline.

Bounded Delay means that parallel reads are not supported, that a bounded number of retry round-trips may occur to read from a server, and/or that a server may block for a bounded amount of time before replying to a read request. An example is Eiger, which requires a maximum of three round-trips to storage servers to read from a snapshot that preserves Atomic Visibility under happens-before order [38].

Mutex reads/writes means that a read might be delayed indefinitely by writes, or vice-versa, because the protocol disallows the same data item from being read and written concurrently to ensure a given isolation property (e.g., Google's Spanner strictly-serialisable transactions [21]).

2.4 Freshness

Another important metric is how recent is the data returned by a read. Users prefer recent data [3]; some strongly-consistent criteria (e.g., Strict Serialisability) might require data to be the latest version; and in others (e.g., Snapshot Isolation) serving recent data makes aborts less likely and hence improves overall throughput [40, 42]. Storing only the most recent version of a data item enables update-in-place and avoids the operational costs of managing multiple versions.

However, many protocols require maintaining multiple versions of a data item to read consistently. Serving an old item state may be faster than waiting for the newest one to become available; indeed, it would be easy for reads to be both fast and consistent, by always returning the initial state.

Freshness is a qualitative measure of whether snapshots include recent updates or not. The most aggressive is **Latest Freshness**, which guarantees that, for every data item that a server stores, it returns the most recent version that it has committed so far. Intuitively, systems like Espresso and Tao [17, 41], which do not make snapshot guarantees, can read with minimal delay under Latest Freshness.

The most conservative is **Stable Freshness**, under which, for any read request in a given transaction, the server returns data taken from a snapshot that is *stable* for that transaction, i.e., all versions included in the snapshot are known to be committed by the time the transaction starts.

The intermediate **Concurrent Freshness** does not necessarily return the latest version, but allows a transaction to read updates committed after its starting point, i.e., not stable. For instance, a transaction T_R can read updates from a committed transaction T_U that ran concurrently with T_R. COPS is a system that exhibits Concurrent Freshness [37].

2.5 Optimal reads

We say a protocol provides optimal reads if it ensures both Minimal Delay and Latest Freshness. An optimal-read protocol is one that supports parallel reads, and where a server is always able to reply to a read request immediately, in a single round trip, with the latest committed version that it stores.

3 The three-way trade-off

In this section, we study the three-way trade-off between transactional reads semantics, delay and freshness. In summary, our analysis concludes the following:

(i) Impossibility of optimal order-preserving reads. Ensuring optimal reads is not possible under Order-Preserving or Atomic Visibility (Section 3.2).

(ii) Order-Preserving Visibility with Minimal Delay and Concurrent Freshness. Order-Preserving Visibility can ensure Concurrent Freshness at Minimal Delay (Section 3.3.2).

(iii) Atomic Visibility with Minimal Delay forces Stable Freshness. To ensure Minimal Delay, Atomic Visibility forces transactions to read from a *stable snapshot*, i.e., a snapshot consisting of updates known to have committed in the past (Section 3.3.3).

(iv) Consistent reads with Latest Freshness. To guarantee reading the freshest data, Order-Preserving and Atomic Visibilities require reads and updates mutually exclusive (Section 3.4).

3.1 Notation and Definitions

Notation. A committed update transaction creates a new version of the data items it updates. For some data item (or object) $x \in \mathcal{X}$, where $\mathcal{X}$ is the universe of object identifiers, we denote a version $x_v \in V$, where V denotes the universe of versions. We assume an initial state $\perp$ consisting of a initial version $x_\perp$ for every $x \in \mathcal{X}$. If versions follow a partial or total order $O = (V, \prec)$, we say a version x_i is more up-to-date (or fresher) than a version y_j when $y_j \prec x_i$.

The database is partitioned, i.e., its state is divided into $P \geq 1$ disjoint subsets, where all the versions of a given object belong to the same partition. Throughout the text, we use the terms partition, server and storage server interchangeably.

Definitions. We define the three types of snapshots introduced in subsection 2.2 formally:

Definition 1 (Committed snapshot) *A committed snapshot S is any subset of V that includes exactly one version of every object $x \in \mathcal{X}$. $\mathcal{S}$ denotes the set of all committed snapshots.*

Definition 2 (Order-Preserving Snapshot) *Given a partial or total order of versions $O = (V, \prec)$, a committed snapshot $S_O \in \mathcal{S}$ preserves O if $\forall x_i, y_j \in S_O$, $\nexists x_k \in V$ such that $x_i \prec x_k \prec y_j$. Intuitively, there is no gap in the order of versions visible in an order-preserving snapshot. We denote $\mathcal{S}_O \subseteq \mathcal{S}$ the set of committed snapshots preserving order O.*

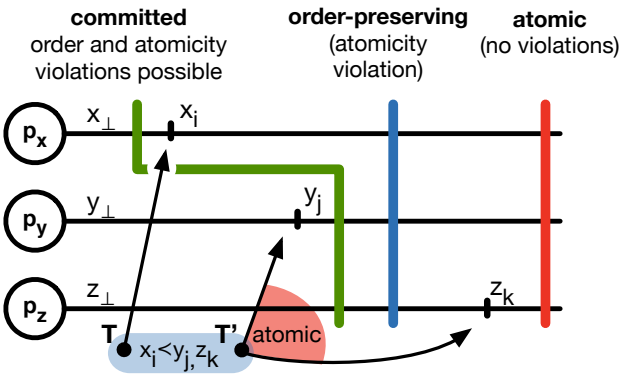


Figure 2 The three snapshot guarantees

Definition 3 (Atomic Snapshot) *Given an order O, an order-preserving snapshot $S_A \in \mathcal{S}_O$ is atomic if $\forall x_i, y_j \in V$ such that x_i, y_j were written by the same transaction, if $x_i, y_k \in S_A$ then $y_k \not\prec y_j$, i.e., it disallows "broken reads". We denote $\mathcal{S}_\mathcal{A}$ the set of atomic snapshots for order O, $\mathcal{S}_\mathcal{A} \subseteq \mathcal{S}_O$.*

Definition 4 (Snapshot guarantee) *Given some order O, we say that a read protocol guarantees Committed (, Order-Preserving or Atomic) Visibility if it guarantees that every transaction reads from a Committed (, Order-Preserving, or Atomic, respectively) snapshot.*

We illustrate the three types of snapshots in Figure 2. The figure shows a system consisting of three partitions, p_x, p_y, and p_z, each storing a single object x, y, and z, in an initial state $x_\perp$, $y_\perp$, $z_\perp$, respectively. Two transactions T and T' have committed updates in order $x_\perp, y_\perp, z_\perp \prec x_i \prec y_j, z_k$. T updates only partition p_x, whereas T' updates p_y and p_z atomically. The figure highlights three possible snapshots. Under Atomic Visibility, only the atomic snapshot is admissible, precluding both order violation (both T and T''s updates are included) and read skew (as both y_j and z_k are included). Under Order-Preserving Visibility, the atomic and the order-preserving snapshots are both admissible. The latter precludes order violations, but not read skews (e.g., the snapshot includes y_j from transaction T', and not z_k). Finally, under Committed Visibility, all three depicted snapshots are admissible because order violations and read skews are allowed. The snapshot at the left of the picture exhibits two anomalies: an order violation, and a read skew. The order violation occurs by reading $x_\perp$ and y_j, as $x_\perp \prec x_i \prec y_j$, and x_i is not read. The read skew occurs by reading $z_\perp$ and y_j, as y_j was created atomically with z_k, which is not read.

3.2 Impossibility of optimal order-preserving reads

Proposition 1 *Order-Preserving (or Atomic) reads cannot ensure optimal (delay and freshness).*

Proof. We prove this proposition by contradiction. Assume that there exists a read-optimal protocol that guarantees Order-Preserving (or Atomic) Visibility, w.r.t. order $O = (V, \prec)$. Consider the execution in Figure 3a where, initially, partition p_x stores $x_\perp$ and p_y stores $y_\perp$. Two transactions T_u and $T_{u'}$ write x_k at p_x and y_j at p_y respectively, establishing the following order: $x_\perp, y_\perp \prec x_k \prec y_j$. For instance, under causal order, this can result from an execution where a transaction reads $x_\perp$, and updates x, creating x_k, and later another transaction reads x_k and updates y, creating y_j. A T_r, running concurrently with T_u and $T_{u'}$, reads objects x and y in parallel from p_x and p_y. T_r reaches p_x before the creation of x_k, and p_y after the creation of y_j. To satisfy read optimality, partitions must reply immediately with the latest version they store, namely $x_\perp$ and y_j, observing an order violation[3] . Contradiction. ■

3.3 Freshness compatible with Minimal Delay

In this subsection we explore which are the maximum freshness degrees achievable for each snapshot guarantee, under the requirement of Minimal Delay.

3.3.1 Optimal reads under Committed Visibility

Proposition 2 *A read protocol that guarantees Committed Visibility can be optimal.*

Proof. Committed Visibility imposes no restrictions to the committed versions a transaction can read. Therefore, to serve a request under this model, a partition can reply immediately with the latest object version it stores. ■

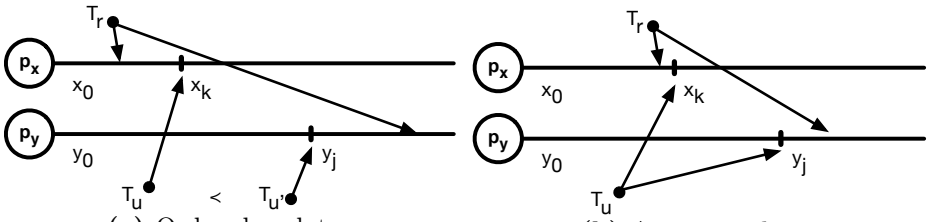
3.3.2 Order-Preserving Visibility and Concurrent Freshness

Proposition 3 *Order-preserving minimal-delay reads can ensure Concurrent Freshness.*

We prove this proposition by sketching a read protocol with such characteristics, followed by a correctness proof. In Section 4, we present a protocol with these characteristics.

Consider a protocol that orders its updates following some order $O = (V, \prec)$, and where reads preserve O. When a read transaction starts, the protocol assigns it an O-preserving *stable* snapshot S_O (subsection 2.4). Read requests are sent to their corresponding partition in parallel. A partition can

[3] A similar situation occurs with the execution of Figure 3b, where reading $x_\perp$ and y_j results in a read skew.



(a) Ordered updates **(b)** Atomic updates

Figure 3 Concurrent distributed read and update transactions

reply immediately with the version in S_O or with a more up-to-date version that is *compatible* with S_O. An object version y_j is compatible with a given order-preserving snapshot S_O if replacing version $y_s \in S_O$ by y_j results in an order-preserving snapshot. Formally:

Definition 5 (Compatible version) *Given an order $O = (V, \prec)$, a version $y_j \in V$ and an order-preserving snapshot S_O, an object version $y_j \notin S_O$ is compatible with S_O, if $\forall x_i \in S_O$, $\nexists x_k \in V$ such that $x_i \prec x_k \prec y_j$.*

Lemma 1 *Given an order-preserving snapshot S_O, replacing any number of versions $x_o \in S_O$ by $x_i \notin S_O$, such that x_i is compatible with S_O, results in an order-preserving snapshot $S_{O'}$.*

Proof. Assume by contradiction that the resulting snapshot $S_{O'}$ is not order-preserving w.r.t. order $O = (V, \prec)$. According to Definition 2, this implies that $\exists x_i, y_j \in S_{O'}, x_k \in V : x_i \prec x_k \prec y_j$. Since S_O is order-preserving, if the versions returned by read partitions were those in S_O, i.e., x_o and y_o, no inconsistency could have been created. Now consider the case where only one compatible version with S_O, e.g., y_j, is more up-to-date than $y_o \in S_O$ ($y_o \prec y_j$). By Definition 5, $\nexists x_k \in V : x_o \prec x_k \prec y_j$. Finally, assume that both $x_i, y_j \notin S_O$ are more up-to-date compatible versions of objects x and y. As they are compatible with S_O, by Definition 5, *(i)*$\nexists x_k : x_o \prec x_k \prec y_j$ and $\nexists y_l : y_o \prec y_l \prec x_i$. Moreover, we know that *(ii)*$x_o \prec x_i$ and $y_o \prec y_j$. *(i)* and *(ii)* imply $y_j \not\prec x_i$ and $x_i \not\prec y_j$. Therefore, there cannot exist $x_k : x_i \prec x_k \prec y_j$. Contradiction. ■

Lemma 2 *The above protocol guarantees Order-Preserving Visibility.*

Proof. This follows directly from Lemma 1. ■

Lemma 3 *The above protocol allows Concurrent Freshness.*

Proof. We prove this lemma by describing a sample execution. Assume transaction T_r starts with stable snapshot $S = \{x_o, y_o\}$. T_r sends a read request for objects x and y to partitions p_x and p_y respectively. Concurrently, update transactions create versions x_u and y_v establishing the following order between them: $x_o, y_o \prec x_u \prec y_v$. T_r's request arrives to p_x after x_u and y_v are committed. By Definition 5, p_x can reply with x_u, a more up-to-date version. However, p_y can only reply with y_o, as y_v is not compatible with S

$(\exists x_u \in V : x_o(\in S) \prec x_u \prec y_v)$. As x_u is committed by an update transaction concurrent to T_r, this execution exhibits Concurrent Freshness. ■

Lemma 4 *The above protocol guarantees minimal delays*

Proof. The protocol reads versions in parallel. In the absence of fresher committed updates than those in S_O, a partition can reply immediately with versions belonging to S_O, which is stable and, therefore, already committed. In the presence of fresher and compatible committed updates, a partition can reply to a request with those, immediately. ■

Proof of Proposition 3. This follows directly from Lemmas 1, 4, 2 and 3. ■

3.3.3 Stable Freshness under Minimal-delay Atomic Visibility

Proposition 4 *A minimal-delay read protocol that guarantees Atomic Visibility requires Stable Freshness.*

The intuition is that, due to the minimal-delay requirement, a partition receiving a read request from a transaction T_r cannot know whether other partitions accessed by T_r are returning updates of a concurrent update transaction T_u or not, which forces it to read from a stable snapshot to avoid Read Skews.

Proof. Assume by contradiction that there exists a minimal-delay protocol that guarantees Atomic Visibility and allows a transaction to read updates committed by other concurrent transactions (Concurrent Freshness). Consider the example execution in Figure 3b. A transaction T_r sends parallel requests to read objects x and y from partitions p_x and p_y respectively. A concurrent transaction T_u commits versions x_k and y_j. Assume that T_r's request reaches p_y after T_u commits. By Definition 3, p_y can return y_j only if it is certain that T_r will read x_k from p_x. Due to the minimal-delay requirements, p_x does not have access to such information, since reads can be executed in parallel and no extra communication among partitions is allowed. Given that T_r can reach p_x before T_u commits x_k, p_y cannot risk returning y_j, and must ignore T_u. Therefore, a partition can only return a version of an update transaction T_u if it knows T_u had committed at all its updated partitions by the time T_r sent its read requests. This implies that T_r has to read from stable snapshot, which contradicts our assumptions. ■

3.4 What is possible under Latest Freshness?

Proposition 5 *Order-Preserving (and Atomic) Visibility require mutually-exclusive reads and updates to guarantee Latest Freshness.*

Lemma 5 *It is not possible to guarantee Order-Preserving or Atomic Visibility with Latest Freshness under Bounded Delay.*

Proof. Consider again the sample execution of Figure 3a (where $x_\perp, y_\perp \prec x_k \prec y_j$). To ensure Latest Freshness, partitions must reply to read requests with the latest committed version they store. If p_x returned $x_\perp$ and p_y returned y_j, T_r would observe an inconsistent result (by missing x_k). The protocol could retry reading from p_x to read x_k, thus ensuring reading a version compatible with y_j. If such request arrived to p_x before T_u created x_k, p_x could block until x_k was applied to read it. During the blocking period, a concurrent update transaction may have written a new version x_m such that $y_j \prec y_w \prec x_m$. To satisfy Latest Freshness, p_x would be forced to reply with x_m, inconsistent with the version read from p_y: y_j. If updates are not stopped, this situation can repeat itself indefinitely, making reading with Bounded Delay impossible. ■

Lemma 6 *A read protocol can ensure Order-Preserving (and Atomic) Visibility and Latest Freshness by enforcing mutually-exclusive reads and updates.*

Proof. We prove this is possible by following the proof of Lemma 5. In the execution of Figure 3a, T_r can retry indefinitely reading the latest versions of x and y until the results belong to an order-preserving snapshot. The equivalent holds for building an atomic snapshot under the execution of Figure 3b. ■

Proof of Proposition 5. This follows directly from Proposition 1, and Lemmas 6 and 5.

3.5 Isolated reads with Bounded Delay and Concurrent Freshness.

Lemma 7 *A read protocol can ensure Order-Preserving (and Atomic) Visibility and Concurrent Freshness under Bounded Delay.*

Proof. Consider again the execution in Figure 3a where read transaction T_r executes concurrently with update transactions T_u and $T_{u'}$. If p_x returns $x_\perp$ and p_y returns y_j, it is possible to issue a second round to force p_x to return x_k, which would ensure Order-Preserving Visibility and Concurrent Freshness. The same holds for ensuring Atomic Visibility in the example execution in Figure 3b. ■

Many existing systems exhibit these properties (see Section 6).

4 Minimal-delay protocol design

In this section, we apply our trade-off analysis to the design of protocols. We design three novel minimal-delay protocols, called CV, OP and AV. Our protocols are a modification of Cure [9]. Cure ensures Transactional Causal Consistency (TCC), i.e., Causal Consistency (where updates follow causal order) and snapshot reads ensure Atomic Visibility. It exhibits Bounded Delays and Concurrent Freshness. We use the insights of the analysis: to remove the delays of Cure one must degrade either read semantics or freshness. AV provides the same isolation guarantees as the base protocol. It achieves Minimal Delay by degrading freshness to stable. OP is the first protocol to provide causally-consistent snapshot reads (Order-Preserving Visibility) and atomic updates. It exhibits Concurrent Freshness. CV ensures Read Committed Isolation, i.e., Committed Visibility and atomic updates. It achieves Latest Freshness.

4.1 Base protocol and changes

Cure associates a snapshot timestamp to a transaction when it starts. When committing updates, a transaction creates object versions which are stored with a commit timestamp. Timestamps are vector clocks sized with the number of sites. Each position in a vector is assigned by the protocol using the physical clocks of servers.

Updates. The protocol can be characterised as *Deferred Update Replication* (DUR), i.e., a transaction buffers updates while executing and sends them to storage servers when committing [44]. A 2PC ensures that the updates of a transaction are applied in an all-or-nothing fashion, and that commit timestamps respect causal order. I.e., that a transaction's commit timestamp is bigger than that of the object versions the transaction read.

Reads. When reading, a transaction sends a read request to a partition including its snapshot timestamp. A partition receiving a request must reply with the version with the maximum commit timestamp smaller than the snapshot timestamp of the request. In Cure, the snapshot timestamp associated to transactions is not necessarily stable. A partition that receives a read request must delay a response when (i) an instance of 2PC is committing a transaction wich needs to be included in the snapshot, and/or (ii) the clock at the partition is behind the snapshot timestamp.

Changes. Our protocols remove the blokcing situations of Cure by applying the following modifications:

- AV degrades freshness. Specifically, it ensures that the snapshot assigned to a transaction is stable.

- OP degrades visibility to Order-Preserving. Given an initial stable snapshot, a partition is allowed to return a more up-to-date compatible version (Definition 5).
- CV degrades read guarantees to Committed Visibility, allowing a partition to return the latest committed version.

To minimise delays of update transactions, all protocols implement a low-latency 2PC protocol, called presumed commit [35]. Under this optimisation, a transaction is considered to be committed after a successful prepare phase, i.e., after every involved partition has persisted a prepare record including its updates. A client can, then, receive a response after a single round-trip of communication with updated partitions.

4.2 Protocols

Setting and notation. All designs consider M fully-replicated sites. Every site partitions data into N partitions. All sites follow the same partitioning scheme, i.e., for each partition $p_n^m, n = 1...N$ at a given site m, there exists a partition p_n^k storing the same objects at every other site $k : k = 1...M, k \neq m$. Updates are replicated across sites asynchronously. We refer to partitions storing the same set of objects at different sites as *sibling* partitions. Each transaction executes entirely within a site.

API. The data access APIs are multi reads and multi updates. A client can group any number of read or update operations respectively, and execute them against the storage servers in parallel. A transaction comprises any number of such multi-operations.

All algorithms share a general skeleton. In what follows, we look at their points in common. In Section 4.3, we address each algorithm's particularities. There are two types of processes involved in a transaction's execution: a *transaction coordinator (TC)*, and the *partition* servers.

4.2.1 Transaction Coordinator Algorithm

The server that receives a new transaction starts a new TC process. The TC lives throughout the execution of the transaction and terminates once the transaction commits or aborts.

Init. A TC initialises (Algorithm 1, Lines 30-35) on a client's first read or update request. It initialises the write set for the transaction WS_T, the associated snapshot ss_T, the transaction's commit time ct_T, and the dependency vector clock dep_T (used for creating a causal order of updates).

Reads. When receiving a read request for a list of object keys (Line 1), the TC splits them by partition by calling the GET_PARTS function (Line 4)

Algorithm 1 Transaction Coordinator tc at site n

```
1: function READ_OBJECTS(keys)
2:     If (!initiated) INIT( )
3:     result = ∅
4:     read_partitions_T=GET_PARTS(keys)
5:     for all ⟨p, keys_p ⟩ ∈ partitions_T do
6:         send ⟨read, keys_p, ss_T⟩ to p
7:     for all ⟨p, keys_p ⟩ ∈ partitions_T do
8:         receive ⟨partition_result⟩ from p
9:         result = result ∪ {partition_result}
10:    If (protocol == OP)
11:        commit_vc=MAX_v(v.commit_vc ∈ result)
12:        dep_T=MAX_v(dep_T, commit_vc)
13:    return result.values
14: function UPDATE_OBJECTS([⟨key, update⟩])
15:    If (!initiated) then INIT( )
16:    WS_T = WS_T ∪ {[⟨key, update⟩]}
17:    return ok
18: function COMMIT( )
19:    ct_T=dep_T[n] +1
20:    updated_partitions_T = GET_PARTS(WS_T.keys)
21:    for all ⟨p, updates_p ⟩ ∈ updated_partitions_T do
22:        send ⟨prepare, updates_p, dep_T, ct_T⟩ to p
23:    for all ⟨p, keys_p ⟩ ∈ updated_partitions_T do
24:        receive ⟨prepared, time_p⟩ from p
25:        If (protocol ≠ CV) then ct_T=MAX(ct_T, time_p)
26:    send ok to client
27:    for all ⟨p, updates_p ⟩ ∈ updated_partitions_T do
28:        send ⟨commit, ct_T⟩ to p
29:    TERMINATE( )
30: function INIT( )
31:    WS_T=∅
32:    If (protocol == OP or AV)
33:        ct_T = ⊥
34:        dep_T = ss_T = GET_STABLE_VECTOR( )
35:        initiated = true
```

and sends a read request to each partition in parallel (Line 6). Once it receives all responses (Line 7), it returns the read values to the client.

Updates and commit. When a client submits an update, the transaction coordinator buffers it (Line 16) in its write set. If the transaction updates the same data item multiple times, they are applied (by the commit protocol) in the order they are submitted. This is useful when updating data structures such as counters, sets, lists, etc. When the client calls commit (Line 18), if the transaction updated multiple partitions, the TC starts an instance of the 2PC protocol among the updated partitions (Lines 21-28). In the prepare phase, the TC sends, in parallel, a prepare message containing partition's updates to each updated partition (Line 22). The participants reply with a prepare time, a proposed commit timestamp for the transaction. Once it has received the response from every participant, the TC i) computes the commit time of the transaction as the maximum proposed prepare time, ii) replies to the client confirming that the transaction has committed, and iii) sends the commit instruction, including the commit timestamp, to all participants. If the transaction updated a single partition, we collapse the prepare and commit messages. This optimisation is not depicted in the pseudocode. An abort will occur only if requested by the client, or in case of a failure. The abort path discards the transaction updates. The protocol, identical to that of presumed commit [35], is not depicted in the pseudocode.

4.2.2 Partition Servers Algorithm

A partition server stores versions of the objects in its partition and replies to requests from transaction coordinators that access those objects.

Reads. When a partition receives a read request from a TC, it replies with the most up-to-date version of each requested object that satisfies the algorithms' visibility criteria (Lines 17-20).

Updates and commit. A partition server participates in instances of 2PC coordinated by some TC. In the first phase, the partition server receives a prepare message containing updates (Line 7). It persists those updates to stable storage (not depicted in the pseudocode), and replies with a *prepared* message; a positive vote to commit the transaction. At this stage, these updates are persisted but not accessible by other transactions, as the transaction has not yet committed. Later, if the partition server receives a commit message (Line 13), it makes updates visible to future readers.

Update propagation. Under replication, a partition server propagates committed updates asynchronously to sibling partitions at remote sites. When a partition receives a remote transaction's updates, it applies them locally and makes them available to future read operations. CV makes

Algorithm 2 Partition m at site n p_m^n

```
1: upon receive ⟨read, keys,ss_T⟩ from tc
2:     result=∅
3:     for all k ∈ keys do
4:         v= newest k_i ∈ ver[k] : COND(k_i, ss_T)
5:         result = result ∪ {v}
6:     send ⟨result⟩ to tc
7: upon receive ⟨prepare, upd, dep_T, ct_T⟩ from tc
8:     If (protocol ≠ CV)
9:         time = MAX(READ_CLOCK( ), ct_T)
10:        prepared = prepared ∪ ⟨tc, upd, dep_T, time⟩
11:    Else prepared = prepared ∪ ⟨tc, upd⟩
12:    send ⟨prepared, time⟩ to tc
13: upon receive ⟨commit, ct_T⟩ from tc
14:    prepared = prepared \⟨tc, upd, dep_T, time⟩
15:    UPDATE_VERSIONS(⟨upd, dep_T, ct_T, n⟩)
16:    to_send=to_send ∪ {⟨upd, dep_T, ct⟩}
17: function COND(k_i, dep_T)
18:    If (protocol==CV) then return true
19:    If (protocol==OP) then return k_i.dep ≤ dep_T
20:    Else return k_i.cv ≤ dep_T
21: function UPDATE_VERSIONS(upd, dep_T, ct, n)
22:    for all ⟨k,val⟩ in upd do
23:        If (protocol = CV) then ver[k] = {val}
24:        Else ver[k] = ver[k]∪ {⟨val, dep_T, ct, n⟩}
```

updates visible as they arrive. The stabilisation protocol run by AV and OP ensures a remote update is made visible to readers respecting causal consistency, as explained in the next section.

4.3 Correctness

We discuss here the differences between the three protocols and focus on their correctness.

Consistent Reads. OP and AV provide across-object isolation. When a TC executes the first read (or update), it assigns the transaction a snapshot timestamp ss_T, which is used as a pivot to compute versions consistent with the target isolation level. ss_T is assigned the partition's stable vector SV_n^m, which denotes the latest stable snapshot known by the partition where the TC runs (Line 34). In Section 4.4, we describe how each partition maintains this vector.

A TC includes ss_T in each read request it sends to partition servers (Line 6). When a server receives a request, it responds with the most up-to-date version that complies with the requested snapshot, according to a $COND$ function parametrised by protocol:

- Under CV, $COND$ returns the latest committed version.
- Under AV, $COND$ ensures that versions do not violate an atomic snapshot that preserves causal order. $COND$ returns the newest version with a commit vector smaller than ss_T (Alg. 2, Line 20). It ensures causal consistency, since the snapshot is stable, i.e., all updates with $cv \leq ss_T$ have been applied. It satisfies Minimal Delay, as partitions can reply immediately. It also guarantees Atomic Visibility (the absence of fractured reads) because all updates of a given transaction commit with the same commit timestamp (as explained later in this section).
- Under OP, $COND$ ensures that versions belong to a causal-order-preserving snapshot. A partition server returns either the version belonging to the stable snapshot ss_T, as above, or with a more up-to-date version compatible with ss_T, if available (see Definition 5). A version is compatible if its associated dependency vector is not larger than ss_T (Alg. 2, Line 19).

AV and OP provide causal consistency, which ensures all session guarantees [48]. We explain how these are ensured in Appendix A.

Causal order of updates. OP and AV ensure updates are causally ordered. A transaction creates an object version with a commit timestamp, and a dependency vector *dep*. *dep* indicates a version is ordered causally after all versions with commit vector $cv \leq dep$. A commit vector cv is created by replacing, in *dep*, the entry of the site where the version was committed by its

Algorithm 3 Stabilisation for AV and OP at p_m^n

```
1: periodically
2:     If (prepared ≠ ∅)
3:         stable_n = MIN(time ∈ prepared) - 1
4:     Else stable_n = READ_CLOCK( )
5:     vec_p[n] = stable_n
6:     send ⟨stable, vec_p⟩ to p_k^n, ∀k ∈ P, k ≠ m
7:     If (to_send ≠ ∅)
8:         for all ⟨upd_p, dep, ct⟩ ∈ to_send : ct ≤ stable_n do
9:             send ⟨updates, upd_p, dep, ct⟩ to p_m^j j ≠ n
10:    Else send ⟨heartbeat, stable_n⟩ to p_m^j, j ≠ n
11: upon receive ⟨stable, vec_p⟩ from all p_k^n, k ≠ m
12:    SV_m^n=MIN_v(vec_p), # ∀p_m^j
13: upon receive ⟨updates,updates, dep, ct⟩ from p_m^j
14:    UPDATE_VERSIONS(updates, dep, ct, n)
15:    vec_p[j] = ct
16:    # update known committed transactions from site j
17: upon receive ⟨heartbeat,stable_j⟩ from p_m^j
18:    vec_p[j] = stable_j
19:    # update stable time from site j
```

commit timestamp ct. To establish a correct causal order, these algorithms must ensure a version's cv is larger than its dep. The transaction coordinator ensures this by picking a ct which is larger than the transaction dependencies (in Alg. 1, Line 19). To ensure that dep is larger than the cv of its predecessors:

- Under OP, a transaction may read a version with cv larger than ss_T. After receiving a read response, the TC updates the transaction's dependency vector dep_T to the maximum cv of a version read (Alg. 1, Line 12).
- AV's read algorithm ensures that a transaction will never read a version with cv larger than ss_T. Therefore, it suffices to assign ss_T to the transaction's dependency vector dep_T (Alg. 1, Line 34).

As Cure, OP and AV ensure that every update of a transaction is assigned the same cv by choosing a transaction's ct as the maximum proposed time by an updated partition (Alg. 1, Line 25). This is used by AV's read protocol to ensure Atomic Visibility.

4.4 Stabilisation protocol

Under OP and AV, a transaction uses the knowledge of a stable snapshot to read consistently and with Minimal Delay. A stabilisation protocol among all partitions in the system computes stable snapshots at each site periodically. Algorithm 3 describes this protocol. It includes the following steps:

1. When a partition server commits a transaction, it adds the transaction's updates to to_send, the list of updates to be sent to sibling partitions at remote sites (Alg. 2, Line 16).
2. Periodically, updates in to_send are propagated to sibling partitions (Line 9). A partition sends updates in commit timestamp (ct) order. A prepared transaction can commit with ct bigger than the prepared time proposed by the partition. A local stable time $stable_n$ is set to be smaller than the minimum prepared time of the transactions prepared at the partition server (Line 3). To ensure updates are sent in ct order, only updates with $ct \leq stable_n$ are sent to sibling partitions (Line 8). If to_send is empty, the partition sends a heartbeat message (Line 10).
3. Each partition server maintains a vector vec_p with an entry per site. An entry j indicates that partition p of site i has delivered locally all updates with commit time $ct \leq vec_p[j]$ from its sibling partition at site j. The local entry of the vector is set to $stable_n$, as the ct of prepared transactions is not defined.
4. Periodically, each partition server sends its vec_p to the other partition servers of its site (Line 6).
5. each partition p_m^n computes SV_m^n, the latest stable snapshot known by p_m^n, as the minimum of all vec_p received (Line 12).

This protocol ensures that the snapshot is stable since (i) no local partition will commit a transaction with commit time smaller than $vec_p[i]$, and (ii) no remote update will be received from a remote site j with commit time smaller than $vec_p[j]$.

5 Evaluation

We empirically explore how the results of the three-way trade-off in Section 3 affect real workloads. We evaluate Cure and the three minimal-delay protocols presented in Section 4: CV, OP and AV.

5.1 Implementation

All protocols were built on the Antidote database [4], an open-source platform built using the Erlang programming language. Antidote uses the Riak-core distributed hash table (DHT) to partition the key-space evenly across physical servers [47]. Object versions are stored in a per-partition in-memory key-value store. During the evaluation, updates were not persisted to durable storage.

Under OP, AV and Cure, a linked list of recent updates is stored for each key. Old versions are garbage collected by a naive mechanism that truncates version lists that contain more than 50 versions, keeping the latest 20.

5.2 Setup

Hardware. All experiments were run on a cluster located in Rennes, France, on the Grid5000 [31] experimental platform using fully-dedicated servers, where each server consists of 2 CPUs Intel Xeon E5-2630 v3, with 8 cores/CPU, 128 GB RAM, and two 558 GB hard drives. Nodes are connected through shared 10 Gbps switches. The ping latency measured within the cluster during the experiment was approximately 0.15 ms.

Configuration. Within the cluster, we configured two logical sites consisting of 16 machines each. Each site is comprised of 512 logical partitions, scattered evenly across the physical machines. Nodes within the same DC communicate using the distributed message passing framework of Erlang/OTP running over TCP. Connections across separate DCs use ZeroMQ sockets [5], also over TCP, where each node connects to all other nodes to avoid any centralisation bottleneck. The stabilisation protocol is run every 10 ms for OP, AV, and Cure. This value has a negligible impact in throughput and freshness, as noted as well in other systems with similar stabilisation mechanisms [9, 28].

Workload generation. The data set used in the experiments consists of 100k keys per server, totalling 1.6 million keys. Objects are registers with the last-writer wins (LWW) policy [33], where updates generate random 100-byte binary values. All objects were replicated at all sites. A custom version of Basho Bench is used to generate the workload [1]. Google's Protocol Buffer interface is used to serialise messages between Basho Bench clients and Antidote servers [18]. To avoid across-machine latencies, two instances of Basho Bench run at each server, which issue requests to the Antidote instance running on it. Each run of the benchmark was run for two minutes, the first minute being used as a warm-up period. A variable number of clients repeatedly run read-only and update transactions. We run first instances with high update rate and number of clients. This rapidly populates the store up-to a point where memory utilisation remains constant due to garbage collection, and response times stabilise. Operations within a transaction select their keys using a power-law distribution, where 80% of the operations are directed to 20% of the keys.

5.3 Experiments

We expect to observe a similar latency response for all minimal-delay protocols, and a slight degradation for Cure, which may block for a small amount of time under clock skew or due to reading concurrent updates. We expect to observe lower latency for CV, as it does not incur the overheads of multi-versioning. Moreover, we expect to observe Order-Preserving visibility to exhibit significantly better freshness than Cure and AV, which implement

Atomic Visibility. We run each experiment in two configurations: the cluster configured as a single DC, or logical split into two DCs. We observed very similar results under both configurations. In what follows, we only present the results of the latter.

Workloads. We run experiments under two workloads which run different read-only transactions. Under the first workload, a read-only transaction reads a number of objects in parallel, in a single round, i.e., in a single call. In the second workload, we try to mimic Facebook's multi-read operations, by issuing read-only transactions that make many calls, each reading a number of objects in parallel. Under both workloads, each client repeatedly executes a read-only transaction followed by an update transaction in a closed-loop (zero think time). An update transaction performs blind updates over a subset of the objects read by the previous transaction. Under every evaluated protocol, this is equivalent to running a single transaction that executes reads followed by updates. We use the separation to simplify the latency measurement of a transaction's read phase.

We vary the number of client threads and measure how latency, and freshness change as load is added to the system. Throughput is measured in operations per second, where each operation is a read or write in a transaction. We measure staleness as follows. When a partition server receives a read request, it logs asynchronously the number of versions needed to skip to guarantee the required isolation property. For Cure, it also logs the cases where it has to wait due to clock skew or for other transactions to commit. We process these logs offline.

Single-shot read-only transactions. A single-round transaction performs 100 reads in parallel. Update transactions perform two, 10 or 100 updates, generating an update rate of approximately 2, 9 and 50% respectively.

Multi-shot read-only transactions. The multi-round experiment mimics the Facebook social network, where a transaction reads thousands of objects in tens to dozens of rounds, and updates represent 0.2% of the workload [17]. In this workload, a read-only transaction executes 10 rounds of 100 parallel reads each (totalling 1000 reads). An update transaction performs two, 100 or 1000 updates, generating an update rate of approximately 0.2, 9 and 50% respectively.

5.3.1 Throughput results.

Due to space limitations, we do not include throughput figures. We explain briefly the behaviour we observed during both experiments. Graphs and a detailed explanation can be found in an extended version of this work [49].

CV exhibits the highest throughput. The other protocols must manage multiple versions; since CV only requires the most recent version, it does not have this overhead. All other protocols exhibit throughput similar to one another throughout the evaluation space. The difference in throughput, between CV and the other protocols, is negligible for workloads with a low rate of updates. It grows to 15% for a workload with ≈10% of updates, and to 35% for 50% of updates. Under Cure, AV and OP, a higher proportion of updates increases (i) the frequency of garbage collection, and (ii) the number of updates running concurrently with a read operation. The latter increases the overhead of traversing version lists to find a version that satisfies a given snapshot.

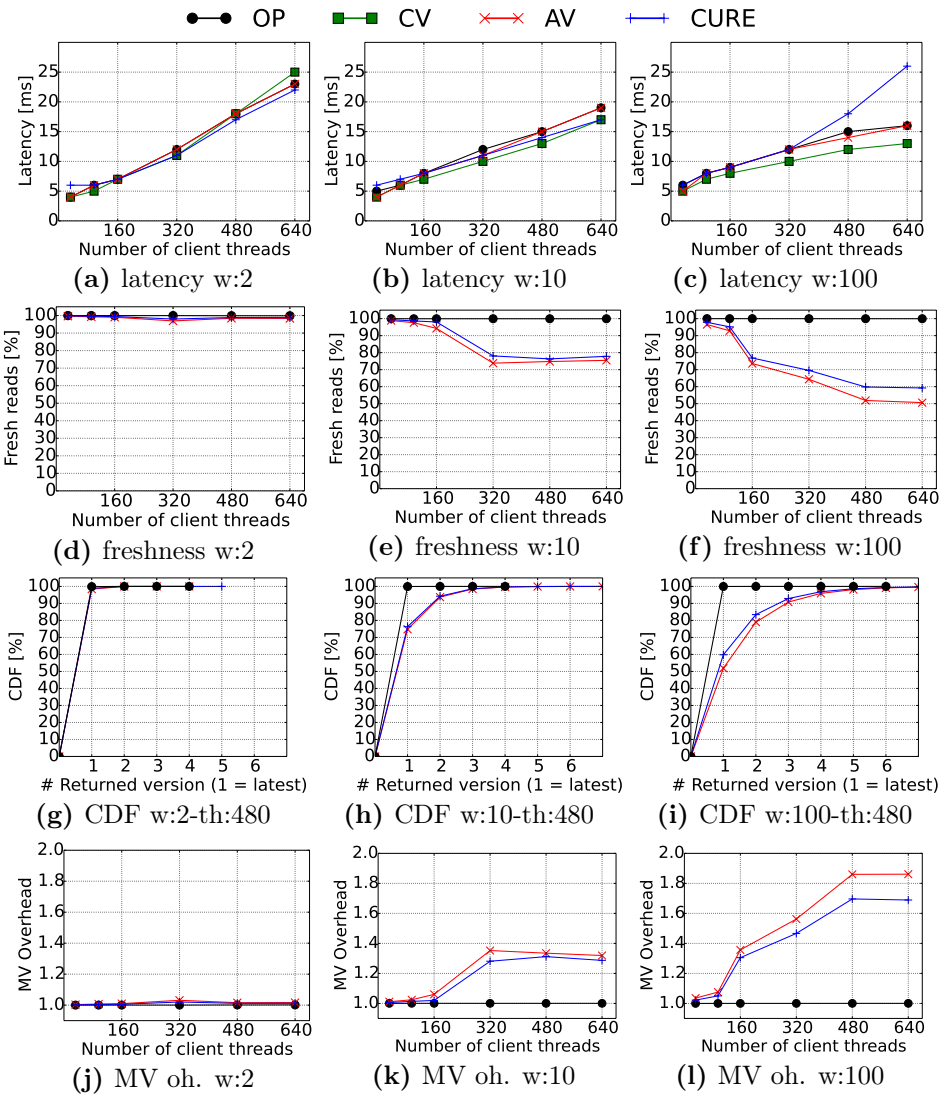
5.3.2 Single-shot read-only transactions

In this section, we present the results of latency and freshness of the first experiment.

Latency. Figures **4a**, **4b** and **4c** show the latency of single-shot read-only transactions. All protocols exhibit a similar trend: increasing the offered load increases latency. CV, AV and OP exhibit minimal delays and, therefore, exhibit very similar latency. CV does not incur the overhead for searching for a compatible version, and has slightly lower latency than OP and AV. Cure exhibits delays. For low update rates, Figures **4a** and **4b** show that, under small number of client threads, Cure exhibits extra (up to 1.9X) latency due to clock skew between servers. At high update rate (Figure **4c**), read operations in Cure wait for update transactions to commit frequently. This causes the latency gap between this protocol and the remaining ones when the number of client threads is large. In Appendix B, we analyse these effects in more depth.

Freshness. Figures **4d**, **4e** and **4f** show the freshness response as the number of client threads increases, for different update rates. Plots display the percentage of read operations that returned the most up-to-date version available at the contacted server. CV is not present in the figures as it always returns the latest version. Figures **4g**, **4h** and **4i** display a CDF showing how stale a read version is under 480 client threads for each update rate. We observe that:

- OP outperforms the other protocols under all workloads. Its freshness response remains nearly constant: over 99.8% of reads observe the latest version under all configurations. This shows that Concurrent Freshness allows this protocol to behave nearly optimally.
- Under the 2%-writes workload (Figure **4d**), OP exhibits, 0.2% of stale reads in the worst case, while Cure 2% (10X), and AV 3% (15X). Figure **4g** shows how fresh reads were under 480 client threads. All protocols



(a) latency w:2 **(b)** latency w:10 **(c)** latency w:100

(d) freshness w:2 **(e)** freshness w:10 **(f)** freshness w:100

(g) CDF w:2-th:480 **(h)** CDF w:10-th:480 **(i)** CDF w:100-th:480

(j) MV oh. w:2 **(k)** MV oh. w:10 **(l)** MV oh. w:100

Figure 4 Single-shot read-only-transactions

read, most of the times, the latest or second most recent version. OP read, in the worst case, the third most recent version, while the remaining two protocols, the fourth. Cure exhibits, under the same conditions, 1.2% (6X) and, AV, 1.8% (9X) stale reads, meaning that potentially every transaction observes stale versions.

- Under 10% of writes (Figure **4e**), OP does not further degrade its freshness, and reads observe the same percentage of stale versions, whereas freshness degrades significantly for AV, which shows 25% (125X) stale reads in the worst case, and Cure, which shows 22% (110X). Figure **4h** shows that OP read mostly fresh versions and, in the worst case, the fourth most-recent version. Cure and AV show a higher frequency or returning the second (≈

20%), third (≈ 5%) and fourth (≈ 2%) most recent versions. As each read transaction executes 100 reads, this means that this potentially affects every transaction. In the worst case, these two protocols read the 12th-to-latest version, not shown in the picture to display them more clearly.

- Under 50% of writes (Figure **4f**), we observe the biggest difference between all protocols: OP still suffers from up to 0.2% of stale reads, while Cure from up to 41% (205X) and AV 49% (245X). Figure **4i** shows that, for 480 threads OP read, in the worst case, the 6th-to-latest version. Cure the 18th and AV the 19th. Cure and AV frequently show significantly stale versions, up to the sixth (≈ 2%) version is potentially observed by every transaction. The oldest version returned by AV was the 19th to latest, and by Cure, the 18th to latest.

Multi-version overhead. We compute the multi-version overhead as the extra work required, for a read operation, to find and store a version that respects a required isolation level, with respect to a single-versioned protocol (e.g., CV). For instance, under this metric, a read that returns the second-to-latest version has an overhead of 2X over the baseline. This metric is computed in practice as the area over the lines of the CDFs. We compute this metric as the overall overhead observed during the entire execution. Figures **4j**, **4k** and **4l** show the results under this workload.

For all workloads, OP shows a very low overhead, under 1.002X over an optimal protocol. AV presents a maximum overhead of 1.03X, 1.35X, and 1.87X under 2, 10 and 100 updates per transaction, respectively, while Cure 1.02X, 1.31X and 1.7X.

Conclusion. We have observed the effects of the three-way trade-off in action. Under this workload, strengthening the semantics from Committed to Order-Preserving visibility incurs a negligible overhead in terms of latency and freshness. However, strengthening the semantics to Atomic Visibility penalises freshness significantly. Both protocols we have experimented with exhibited a high degradation in freshness. Cure exhibits better freshness than AV at a latency cost that increases with contention.

5.3.3 Multi-shot read-only transactions.

We perform the same analysis under the multi-shot workload. Figure 5 shows the results. Results follow the same trend as those of the previous workload. However, some effects get diminished while others get augmented. In what follows, we refer to the differences between results.

Latency. Figures **4a**, **4b** and **4c** show the latency response of all protocols under this workload. These transactions exhibit significantly higher —around 10X— latency than single-shot transactions, as they incur 10 rounds of 100

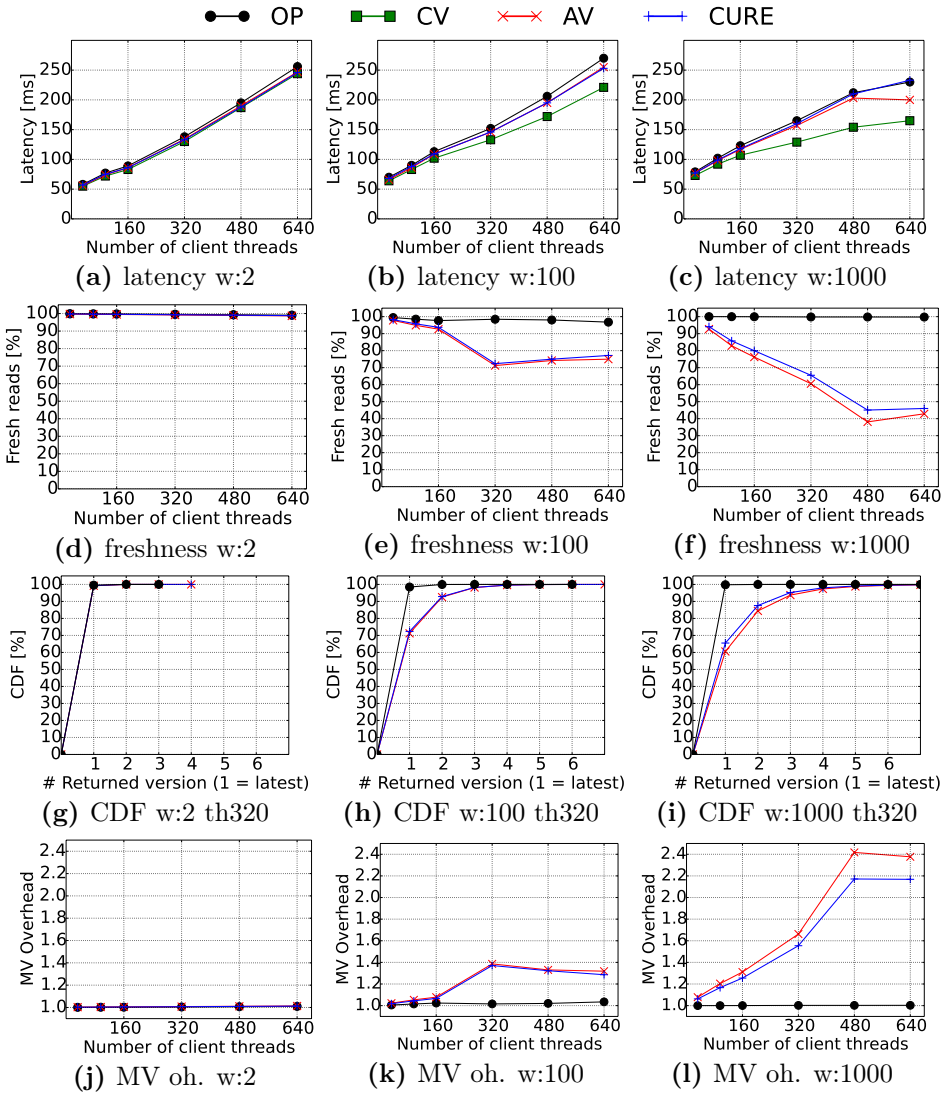
reads each. The trend of all systems is very similar to that of single-shot transactions: CV outperforms the remaining protocols, and the difference becomes larger as update rate augments. One difference with respect to single-shot transactions is that OP exhibits slightly worst performance than the other systems. This happens because of OP's mechanism for enforcing causal order: every time a transaction coordinator receives a read response, it must recompute the transaction's causal dependencies (Algorithm 1, Lines 11 and 12). Under this workload, each transaction coordinator performs this computation 1000 times. Nevertheless, the protocol could be modified to avoid this situation by performing such computation in parallel with subsequent read operations, which we have not experimented with.

Freshness and multi-version overhead. Figures **5d**, **5e** and **5f** show the freshness response as the number of client threads increases, for different update rates. Figures **5g**, **5h** and **5i** display a CDF showing how stale a read version is under 320 client threads for each update rate. The trend is similar to that of single-shot read-only transactions: OP outperforms the remaining protocols under all configurations, and Cure and AV degrade freshness significantly as contention is added to the system. For all protocols, the effect of staleness gets magnified with respect to that of single-shot transactions. This occurs because transactions are long lived, which renders interleaving with update transactions more frequent. The worst-case scenarios are 5% of stale updates for OP, while 62% for AV, and 55% for Cure. In terms of oldest versions read under contention (50% of updates and maximum client load), OP returned, at most, the 7th-to-latest version, while Cure the 28th and AV the 31st. Figures 5j, 5k, and 5l show the multi-version-overhead results under this workload. Graphs follow a similar-but-magnified trend too as that of single-shot reads, where overhead peaks at 1.05X for OP, 2.2X for Cure, and 2.4X for AV.

Conclusion. Under this workload, as transactions live long time, all protocols exhibit similar latency, including Cure, which is not latency optimal. In terms of freshness, we observe that protocols with Atomic Visibility get highly penalised as contention increases.

6 Related Work

Impossibility results. The CAP theorem proves that, in a replicated system, it is impossible to guarantee strong consistency (C), high availability (A), and partition tolerance (P) at the same time [30]. Under partition, system designers must decide between remaining available but weakly-consistent (AP), or remaining strongly-consistent but not available (CP). Strong consistency requires, at the minimum, ensuring that single object updates respect



(a) latency w:2 **(b)** latency w:100 **(c)** latency w:1000

(d) freshness w:2 **(e)** freshness w:100 **(f)** freshness w:1000

(g) CDF w:2 th320 **(h)** CDF w:100 th320 **(i)** CDF w:1000 th320

(j) MV oh. w:2 **(k)** MV oh. w:100 **(l)** MV oh. w:1000

Figure 5 Multi-shot read-only transactions

a total monotonic order, which requires synchronous replication. This result has motivated designs that foster latency to eschew both strong consistency and read isolation [8], properties frequently mixed. This effect has been exacerbated by the fact that all existing AP designs that enforce isolation exhibit delays. This work shows that the performance of read isolation is orthogonal to update-order enforcement. Moreover, the presented algorithms are available under partition and exhibit minimal delays.

Lu et al. prove that no protocol can achieve Strict Serialisability (which implies Atomic Visibility) and Minimal Delay [39]. Therefore, their work proves that the upper-left corner of Figure 1 is unachievable under this model. Strict Serialisability requires that, if an update transaction T_U commits

updates at time t_U, all transactions that start after t_U, in real time, observe T_U's updates. This requirement disallows that a transaction reads from an outdated snapshot. Intuitively, knowing the latest set of committed transactions at a given point in time requires coordination, which implies added delay. Their result is complementary to ours.

Didona et al. prove that a minimal-delay causally-consistent (order-preserving) distributed read protocol forces expensive updates [25]. Their work considers a more-restrictive model; a client, which coordinates a transaction, is not allowed to contact a partition server (and vice-versa) outside the scope of the transaction. This forbids a client to receive asynchronous notifications regarding stable snapshots. This is instrumental to achieve Minimal Delay and progress in our protocols. Without this restriction, that result is similar to Proposition 1.

Attiya et al. determined bounds on the response times of Sequential Consistency and Linearisability, consistency guarantees of single objects [12]. The results of that work can be used to help determining the delays of implementations that rely on blocking to ensure a given isolation property (e.g., Strict Serialisability requires linearisable objects).

Systems. We relate the properties of existing systems to the trade-off. Table 1 summarises the discussion.

Weak Isolation. Systems that are designed for high-availability and low latency under replication are those that do not require a per-object monotonic total order of updates, and thus avoid synchronous replication. Surprisingly, minimal-delay designs with read isolation are missing from the literature, as they all incur delays.

No Isolation. Espresso [41], Tao [17], Yahoo's PNUTS [20], Amazon's Dynamo [24], Twitter's Rainbird [2], and Google's BigTable [19] ensure optimal reads but no atomic updates or read isolation. Cassandra offers atomic updates and no isolation. Reads are only prevented from observing partial updates within a row [23].

Committed Visibility. In MySQL cluster, reads may block waiting for a transaction to commit [6]. We introduce CV, a protocol providing atomic updates and optimal committed reads.

Order-Preserving Visibility. All systems we discuss provide snapshot reads that preserve causal order. None supports all-or-nothing updates. Similarly to Cure [9], GentleRain [28], Orbe [27] and ChainReaction [10] block to wait for concurrent transactions to commit and for clocks to catch up. COPS [37] incurs multiple read-rounds. In Contrarian [25], a transaction executes a communication round to any partition server to obtain a stable snapshot, followed by the read phase. COPS-SNOW [39] offers Minimal Delay and Concurrent Freshness. It removes the second round of reads in COPS by

System	Atomic Updates	Update Ordering	Model	Read Isolation	Delay	Freshness
PNUTS	☐	-	-	-	minimal	latest
Dynamo	☐	-	-	-	minimal	latest
Rainbird	☐	-	-	-	minimal	latest
BigTable	☐	-	-	-	minimal	latest
Espresso	☐	-	-	-	minimal	latest
Tao	☐	-	-	-	minimal	latest
Cassandra	☑	-	-	-	minimal	latest
MySQL Cluster	☑	-	RC	Committed	bounded	latest
CV (this paper)	☑	-	RC	Committed	minimal	latest
COPS	☐	Partial	Causal	O. Preserving	bounded	concurrent
COPS-SNOW	☐	Partial	Causal	O. Preserving	minimal	concurrent
GentleRain	☐	Total	Causal	O. Preserving	bounded	concurrent
Orbe	☐	Partial	Causal	O. Preserving	bounded	concurrent
ChainReaction	☐	Partial	Causal	O. Preserving	bounded	concurrent
Contrarian	☐	Partial	Causal	O. Preserving	bounded	stable
OP (this paper)	☑	Partial	Causal	O. Preserving	minimal	concurrent
Cure	☑	Partial	TCC	Atomic	bounded	concurrent
Eiger	☑	Partial	TCC	Atomic	bounded	concurrent
RAMP	☑	Partial	RA	Atomic	bounded	concurrent
AV (this paper)	☑	Partial	TCC	Atomic	minimal	stable
Jessy	☑	Partial	NMSI	Atomic	bounded	concurrent
Blotter	☑	Partial	NMSI	Atomic	bounded	concurrent
Walter	☑	Partial	PSI	Atomic	bounded	stable
Occult	☑	Partial	PSI	Atomic	mutex R/W	concurrent
GMU	☑	Partial	US	Atomic	bounded	concurrent
Clock-SI	☑	Monot. TO	SI	Atomic	bounded	concurrent
CockroachDB-SI	☑	Monot. TO	SI	Atomic	bounded	concurrent
CockroachDB-SSI	☑	Monot. TO	Ser.	Atomic	mutex R/W	concurrent
Spanner ROTX	☑	Monot. TO	Ser.	Atomic	bounded	stable
Spanner	☑	Monot. TO	Stict Ser.	Atomic	mutex R/W	latest
Rococo	☑	Monot. TO	Stict Ser.	Atomic	mutex R/W	latest
Rococo-SNOW	☑	Monot. TO	Stict Ser.	Atomic	mutex R/W	latest

Table 1 Guarantees, delay and freshness for several published systems.

rendering updates expensive. An update operation must update data structures of all the objects it causally depends on. Both of these systems rely on metadata sized with the number of objects in the system to causally-order updates. The introduced OP has the same read semantics without incurring such costs, and furthermore providing atomic updates.

Atomic Visibility. Cure exhibits the blocking scenarios introduced in Section 4.1. The remaining considered systems incur multiple rounds of reads. Examples include Eiger [38] and RAMP [14]. This work introduced AV, the first weakly-consistent protocol that achieves minimal delays and Atomic Visibility. It implements TCC.

Strong Isolation. Cassandra offers single-object strong consistency through synchronous replication, and no cross-object isolation [22].

Walter, Occult, Blotter, Jessy and GMU ensure causal order and enforce a per-object total order by avoiding conflicting writes. They achieve Concurrent Freshness with delays. Walter retries reads. Blotter, Jessy and GMU read sequentially. Occult exhibits Unbounded Delays. A transaction attempts to read from an atomic snapshot from sites that might be in an inconsistent state. It aborts when it detects an inconsistency.

Clock-SI provides Snapshot Isolation. Its read algorithm is very similar to that of Cure and GentleRain; it blocks in the case of clock skew or waiting for transactions to commit. CockRoachDB offers Snapshot Isolation and Serialisability, both ensuring Atomic Visibility. Under the former, reads might block waiting for a transaction to commit. Under the latter, a read-only transaction might abort (indefinitely) when it detects a serialisation conflict.

Spanner features two kinds of transactions. Strictly-serialisable transactions rely on locks to ensure mutually-exclusive reads and writes (by Proposition 5, this is unavoidable). Spanner's read-only transactions exhibit Bounded Delay as a server might need to wait for physical clocks to advance past a transaction's snapshot time. Rococo provides Strict Serialisability (Atomic Visibility) under Latest Freshness. Its read algorithm issues an unbounded number of rounds to ensure its desirable guarantees. Rococo-SNOW issues a bounded number of read rounds, and blocks updates when these rounds do not attain Atomic Visibility and Latest Freshness. Once updates are stopped, these are guaranteed.

7 Conclusion

We have explored the three-way trade-off between a transactional read algorithm's isolation guarantees, its delays, and its freshness, and analyse a spectrum of possible points in the design space. Interestingly, order-preserving minimal-delay reads can be fresher than (the strongest) atomic. Moreover,

reading the most up-to-date data and ensuring isolated reads, required by strict serialisability, is only possible by implementing mutually-exclusive reads and updates, which may delay reads indefinitely. We have used these results to guide protocol design. Departing from an existing transactional protocol exhibiting delays, we have created three minimal-delay variants: one maintains its read guarantees by degrading its freshness, while the other two improve freshness in different degrees by degrading read guarantees. The evaluation of these three protocols supports the theoretical conclusions of the trade-off.

Acknowledgments

This research is supported in part by European FP7 project 609 551 SyncFree (2013–2016), by European H2020 project LightKone #732 505 (2017–2020), and by the RainbowFS project of *Agence Nationale de la Recherche*, France, number ANR-16-CE25-0013-01.

References

[1] Basho bench. http://github.com/SyncFree/basho_bench.

[2] Rainbird: Real-time analytics@ twitter. https://www.slideshare.net/kevinweil/rainbird-realtime-analytics-at-twitter-strata-2011.

[3] Twitter, inc. https://twitter.com/.

[4] AntidoteDB. http://syncfree.github.io/antidote/, 2015.

[5] ZeroMQ. http://http://zeromq.org/, 2015.

[6] MySQL :: MySQL NDB Cluster :: Limits Relating to Transaction Handling in NDB Cluster. https://dev.mysql.com/doc/mysql-cluster-excerpt/5.7/en/mysql-cluster-limitations-transactions.html, 2018.

[7] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995. doi: 10.1007/BF01784241. URL http://dx.doi.org/10.1007/BF01784241.

[8] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *HOTOS*, pages 13–13, Berkeley, CA, USA, 2015. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2831090.2831103.

[9] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 405–414, Nara, Japan, June 2016. doi: 10.1109/ICDCS.2016.98. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.2016.98.

[10] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A causal+ consistent datastore based on chain replication. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pages 85–98, Prague, Czech Republic, 2013. doi: 10.1145/2465351.2465361. URL http://doi.acm.org/10.1145/2465351.2465361.

[11] ANSI. X3. 135-1992, American National Standard for Information Systems-Database Language-SQL, 1992.

[12] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability (extended abstract). In *SPAA '91*, SPAA '91, pages 304–315, New York, NY, USA, 1991. ACM. ISBN 0-89791-438-4. doi: 10.1145/113379.113407. URL http://doi.acm.org/10.1145/113379.113407.

[13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, November 2013. doi: 10.14778/2732232.2732237. URL http://dx.doi.org/10.14778/2732232.2732237.

[14] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD*, pages 27–38, New York, NY, USA, 2014. ACM. doi: 10.1145/2588555.2588562. URL http://doi.acm.org/10.1145/2588555.2588562.

[15] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223785. URL http://doi.acm.org/10.1145/568271.223785.

[16] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-10715-5.

[17] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, San Jose, CA,

2013. USENIX. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

[18] Protocol Buffers. Google's data interchange format, 2011.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816. URL http://doi.acm.org/10.1145/1365815.1365816.

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008. doi: 10.1145/1454159.1454167. URL http://dx.doi.org/10.1145/1454159.1454167.

[21] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 251–264, Hollywood, CA, USA, October 2012. Usenix. URL https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf.

[22] DATASTAX. Configuring data consistency in cassandra. http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html, 2018.

[23] DataStax. How are Cassandra transactions different from RDBMS transactions? https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlTransactionsDiffer.html, 2018.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, October 2007. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/1294261.1294281.

[25] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Causal consistency and latency optimality: Friend or foe? *Proc. VLDB*

Endow., 11(11):1618–1632, July 2018. ISSN 2150-8097. doi: 10.14778/3236187.3236210. URL https://doi.org/10.14778/3236187.3236210.

[26] Phil Dixon. Shopzilla site redesign: We get what we measure. In *Velocity Conference Talk*, 2009.

[27] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pages 11:1–11:14, Santa Clara, CA, USA, October 2013. Assoc. for Computing Machinery. doi: 10.1145/2523616.2523628. URL http://doi.acm.org/10.1145/2523616.2523628.

[28] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Symp. on Cloud Computing*, pages 4:1–4:13, Seattle, WA, USA, November 2014. Assoc. for Computing Machinery. doi: 10.1145/2670979.2670983. URL http://doi.acm.org/10.1145/2670979.2670983.

[29] David Kenneth Gifford. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford, CA, USA, 1981. AAI8124072.

[30] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/564585.564601.

[31] Grid'5000. Grid'5000, a scientific instrument [...]. https://www.grid5000.fr/, retrieved April 2013.

[32] R. C. Hansdah and Lalit M. Patnaik. Update Serializability in Locking. In *ICDT '86*, pages 171–185, London, UK, UK, 1986. Springer-Verlag. ISBN 3-540-17187-8. URL http://dl.acm.org/citation.cfm?id=645497.658206.

[33] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976. URL http://www.rfc-editor.org/rfc.html.

[34] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. January 1979. URL https://www.microsoft.com/en-us/research/publication/crash-recovery-in-a-distributed-data-storage-system/.

[35] Butler W. Lampson and David B. Lomet. A New Presumed Commit Optimization for Two Phase Commit. In *VLDB*, pages 630–640, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. ISBN 1-55860-152-X. URL http://dl.acm.org/citation.cfm?id=645919.672675.

[36] Greg Linden. Make data useful, 2006.

[37] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043593.

[38] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 313–328, Lombard, IL, USA, April 2013. URL https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf.

[39] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW Theorem and Latency-optimal Read-only Transactions. In *OSDI'16*, pages 135–150, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026889.

[40] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. ICDCS, pages 455–465, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4685-8. doi: 10.1109/ICDCS.2012.55. URL http://dx.doi.org/10.1109/ICDCS.2012.55.

[41] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. On Brewing Fresh Espresso: Linkedin's Distributed Data Serving Platform. In *SIGMOD*, pages 1135–1146, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465298. URL http://doi.acm.org/10.1145/2463676.2465298.

[42] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 163–172, Braga, Portugal, October 2013. IEEE Comp. Society. doi: 10.1109/SRDS.2013.25. URL http://dx.doi.org/10.1109/SRDS.2013.25.

[43] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[44] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable Deferred Update Replication. In *DSN*, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8. URL http://dl.acm.org/citation.cfm?id=2354410.2355159.

[45] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. Consistency in 3D. In Josée Desharnais and Radha Jagadeesan, editors, *Int. Conf. on Concurrency Theory (CONCUR)*, volume 59 of *Leibniz Int. Proc. in Informatics (LIPICS)*, pages 3:1–3:14, Québec, Québec, Canada, August 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany. doi: 10.4230/LIPIcs.CONCUR.2016.3. URL http://drops.dagstuhl.de/opus/volltexte/2016/6188/pdf/LIPIcs-CONCUR-2016-3.pdf.

[46] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043592.

[47] Basho Technologies. Riak KV. http://basho.com/products/riak-kv/.

[48] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

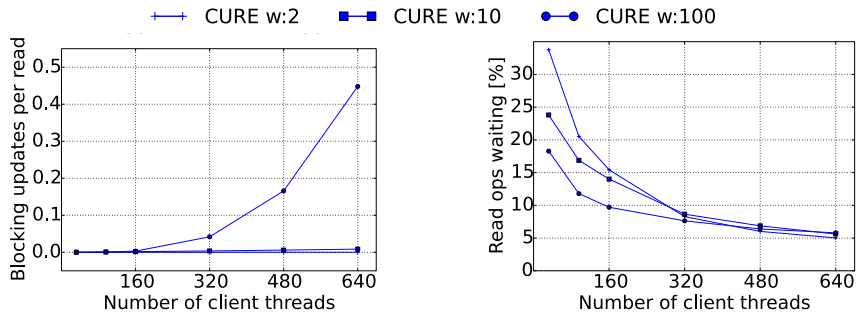
[49] Alejandro Z. Tomsic. *Exploring the design space of highly-available distributed transactions*. PhD thesis, Sorbonne Université, 2018.

A Protocols: session guarantees

Read your writes. The algorithms presented in Section 4 do not ensure the *read your writes* session guarantee, required by causal consistency. The problem arises because updates must undergo the stabilisation process to be available to further transactions. If a client's transaction performs some updates, a subsequent transaction by the same client might miss its latest updates because they are not stable yet.

Read your writes can be enforced by a client caching its latest updates. When receiving a read response, the client compares the version received with the cached one (if any). If the cached version is fresher, the one returned by the system is discarded. After a transaction finishes, a TC returns the latest stable vector SV it is currently aware of. A client can invalidate all updates with $cv \leq SV$.

Monotonic Reads. Under AV and OP, monotonic reads are ensured if a client always connects to the same server. However, if the server fails or



(a) Wait due to committing update **(b)** Wait due to clock skew

Figure 6 Cure blocking scenarios, single-shot workload

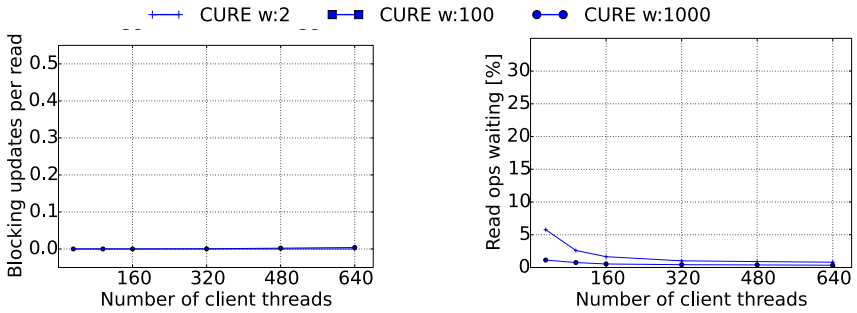
becomes unreachable, a client might connect to a server where SV is behind that of the ss of the client's last transaction. This might lead to observing less up-to-date data than what the client previously observed. This violates the monotonic-reads session guarantee.

To ensure monotonic reads, a client informs a TC of its latest ss. When a transaction commits, the TC returns the transaction's ss. If a TC detects that its SV is behind the client's ss ($SV < ss$), it catches up as follows: if the TC runs at the same site as the client's latest transaction, it updates its SV to ss immediately. This is possible because the snapshot was previously observed as stable by another partition at the same site. If the TC runs at a different site, TC has to block until the stabilisation protocol validates $SV \geq ss$.

B Evaluation of blocking in Cure

Single-shot RO transactions. The design of Cure is not latency optimal. We plot, in Figure **6b**, the percentage of read operations that blocked due to clock skew. This effect is frequent under small number of client threads, and dissipates as the system becomes more loaded. Under high load, the time it takes to process a received read-request message is larger, and during this time, lagging clocks can catch up. Figure **6a** shows how, when keys become highly contended (i.e., at high update rate and number of client threads), waiting for an update operation becomes frequent. At maximum contention —640 threads and 50% of updates— a read operation waits, on average, for 0.45 update operations to finish or, equivalently, each transaction waits for an average of 45 updates to commit.

Multi-shot RO transactions. Under this workload, where transactions execute for a long time, the blocking cases of Cure are significantly reduced with respect to those of single-shot transactions. Figure **7a** shows the percentage of read operations that blocked due to clock skew under Cure. As



(a) Wait due to committing update **(b)** Wait due to clock skew

Figure 7 Cure blocking scenarios, multi-shot workload

we see, the effect practically disappears —below 6% of reads block— under all workloads. If we consider that each read round takes approximately 10ms, rounds after the first one are very unlikely to block due to clock skew, where most of waiting is expected to happen. The same occurs with blocking due waiting for update transactions to commit, as shown in Figure **7b**. Under maximum contention, under 1% of read operations block due to this effect.