The Logical Timestamp Skew Anomaly in Event-Replicated Transaction Schedulers

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

30-06-2021 / 07-07-2021

CITATION

Koutanov, Emil (2021): The Logical Timestamp Skew Anomaly in Event-Replicated Transaction Schedulers. TechRxiv. Preprint. <https://doi.org/10.36227/techrxiv.14880411.v1>

DOI

[10.36227/techrxiv.14880411.v1](https://doi.org/10.36227/techrxiv.14880411.v1)

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

The Logical Timestamp Skew Anomaly in Event-Replicated Transaction Schedulers

Emil Koutanov

Distributed Systems Research Division, Obsidian Dynamics, NSW 2204 Australia

Corresponding author: Emil Koutanov (ek@obsidiandynamics.com)

ABSTRACT To sidestep reasoning about the complex effects of concurrent execution, many system designers have conveniently embraced strict serializability on the strength of its claims, support from commercial and open-source database communities and ubiquitous levels of industry adoption. Crucially, distributed components are built on this model; multiple schedulers are composed in an event-driven architecture to form larger, ostensibly correct systems. This paper examines the oft-misconstrued position of strict serializability as a composable correctness criterion in the design of such systems. An anomaly is presented wherein a strict serializable scheduler in one system produces a history that cannot be serially applied to even a weak prefix-consistent replica in logical timestamp order. Several solutions are presented under varying isolation properties, including novel isolation properties contributed by this paper. We also distinguish between concurrent schedulers based on their propensity to produce deterministic histories. It is further shown that every nondeterministic scheduler is anomaly-prone, every nonconcurrent scheduler is anomaly-free, and that at least one deterministic concurrent scheduler is anomaly-free.

INDEX TERMS Concurrency control, Consistency, Databases, Distributed Systems, Replication, Transaction Processing

I. INTRODUCTION

Strict serializability has been much lauded as the holy grail of transactional isolation levels, particularly in the context of distributed systems. We say ‘particularly’ only because serializability was originally prescribed in the epoch of centralised transactional databases where, with few exceptions [1], execution on a single compute node almost invariably implied that non-concurrent transactions were applied in real-time order, and hence were free of known anomalies. In other words, most centralised systems that guaranteed serializability in fact delivered strict serializability¹, without claiming as much. It was not until transaction processing was distributed that the pitfalls of serializability became apparent [4], prior assumptions were challenged and strict serializability emerged as the new ideal.

To this day, strict serializability is sold as the gold standard of isolation. No one can be certain how this came to pass. It may be in part owing to several sources [2], [3], [4] that have presented strict serializability as a constraint that implies

linearizability, perhaps (and without stating as much) alluding to it being a necessary and sufficient correctness criterion for the design of data-intensive systems. At any rate, strict serializability is a widely adopted isolation model in the context of database systems [8], [22], [27], [29], [31], [32]; such systems approximate serial execution closer than their weaker (isolation-wise) counterparts, while still permitting concurrent execution. Due to its ubiquity and the surplus of literature casting it in a positive light, strong serializability can be easily mistaken for a model that accurately emulates serial execution under all conceivable circumstances, including in distributed systems. That title de facto belongs to actual serial execution—a model that offers perfect isolation, owing to its intrinsic lack of concurrency, however constraining and impractical it may appear in the age of performance-oriented computing.

The formal definitions of various isolation models and the principles by which they relate are unquestionably interesting in an academic sense; however, they are mute to those outside

¹ Occasionally, ‘strict serializability’ is referred to as ‘external consistency’ in literature [8], [10].

of a select few who understand concurrency control theory and can reason about execution histories and other such esoterica. The majority of practitioners²—engineers who are tasked with the construction of real-life, budget-constrained and presumably revenue-generating systems—care little about the intricacies of schedulers and resulting execution histories, commutativity of operations, serial equivalence, anomalies, failure modes, isolation properties, et cetera. Granted, it is unjust to assert that engineers have not been adequately educated, for material on concurrency theory is readily available and its key concepts are routinely taught in tertiary institutions. Prior education notwithstanding, it cannot be safely assumed that the relevant knowledge was carried through their career unabated and duly expanded upon. In this paper, we assume that a database to an engineer is little more than a passive “bucket” for storing records. While this supposition may seem overly generalised, it is the conservative approach that we err on.

In many ways, the broadly perceived definition of serializability has as much to do with actual equivalence to some serial schedule as it has with the following oft-quoted promise, rephrased here: “*if transactions perform correctly when executed serially, they will perform correctly under serializability*”, or this simpler rendition: “*each transaction operates on the database as if it were running by itself*” [5].

In support of the above, consider this statement by Papadimitriou in his seminal JACM paper [9]: “*The appeal of serializability as a correctness criterion is quite easy to justify. Databases are supposed to be faithful models of parts of the world, and user transactions represent instantaneous changes in the world. Since such changes are totally ordered by temporal priority, the only acceptable interleavings of atomic steps of different transactions are those that are equivalent to some sequential execution of these transactions. Another way of viewing serializability is as a tool for ensuring system correctness. If each user transaction is correct—i.e., when run by itself, it is guaranteed to map consistent states of the database to consistent states—and transactions are guaranteed to be intermingled in a serializable way, then the overall system is also correct.*” Papadimitriou’s statement, made in 1979 is still commonly used today in promoting serializability as a sort of composable correctness property.

Evidently, serializability attempts to erase the notion of concurrency from reasoning, eliminating the distinction between concurrent and serial execution. The capacity to safely disregard the effects of concurrency while retaining its benefits is, indeed, a compelling proposition. It encourages a linear way of reasoning about transactions that is approachable by a broad audience with diverse skills and experience, whereas reasoning about complex effects of concurrent

execution without the aid of serializability requires a niche set of skills and is a taxing and error-prone undertaking.

Finally, practitioners understand this reductionistic prose because it is presented as a straightforward obligation from the database vendor to the user. Provided a user fulfils a relatively trivial obligation, the database will fulfil a much more complex one; specifically, if a transaction is safe on its own by design, then the complete system will be safe by implication. This form of composition immediately presents a problem: while the simplified definitions are not incorrect, they erase the subtle implications of the theoretical definition—namely, that transactions may be reordered to conform with a serial schedule. It is precisely this reordering that may surprise the unprepared user, leading to much confusion and resulting in subtle implementation defects.

A. CONTRIBUTIONS OF THIS PAPER

This paper introduces a taxonomy for classifying schedulers based on their capacity to produce deterministic histories. *Deterministic schedulers* always produce conflict equivalent histories for committed transactions, given the same set of transaction inputs, in addition to satisfying some isolation property. *Nondeterministic schedulers* may produce arbitrary histories that satisfy an isolation property. This paper demonstrates three results:

1. That even the strongest isolation model that permits nondeterministic concurrent execution, applied to some primary data store, may result in an anomaly in a serially replicated data store;
2. That this anomaly is absent from every serial execution, where the writes to the primary data store are submitted one after another; and
3. That this anomaly is absent from some deterministic schedulers that permit concurrent execution on the primary data store.

We have named this anomaly **logical timestamp skew (LTS)** after the pathological behaviour that induces the inconsistency on the replica.

For (2), the corollary is that concurrency in the primary scheduler is a necessary criterion to induce the LTS anomaly in a replica.

For (3), the corollary is that concurrency is not sufficient to induce the anomaly. In support of this, we present a novel parametric isolation property—**externalizability**, and its nonparametric special case, **Φ -serializability**—that can be used to specify a deterministic concurrent scheduler that is LTS anomaly-free.

The rest of this paper is structured as follows. Section II presents the necessary theoretical concepts and common definitions that are cited throughout this paper. Section III introduces the system model and describes the LTS anomaly

² We distinguish between the terms ‘practitioner’ and ‘user’ as follows: a user is an arbitrary consumer of a database system, while a practitioner is a moderately educated user who may not be fully aware of the system’s

limitations or other idiosyncratic behaviour. Occasionally we interchange ‘practitioner’ with ‘engineer’ to emphasize the rigour of their education and the propensity for the application of engineering methods.

using a combination of theoretical and practical examples. Section IV offers a thorough analysis of the anomaly, the conditions which are susceptible to this anomaly and the conditions which are anomaly-free, and presents several diverse solutions to the anomaly, complete with the necessary proofs. Section V recounts related work in the field of distributed transaction processing and concurrency control. Section VI summarises this paper.

II. DEFINITIONS AND THEORETICAL FOUNDATIONS

This section summarises known concepts and results of concurrency control theory and introduces some useful definitions as a basis for the following sections.

A. TRANSACTIONS AND HISTORIES

A **transaction**, informally, comprises a set of operations over shared resources. A transaction is **atomic**; i.e., it either completes successfully and all its effects on the resources are made **durable**, or all its effects are undone. In the former, the transaction is said to be **committed**; in the latter, it is **aborted**.

A data item is an elemental, named data resource within the set of shared resources. In practice, a data item may be a tuple in a relational database, identified by its primary key, or a field in a tuple, identified by a combination of the key and column name. Alternatively, it may be a document in a key-value database, identified by a document key, or some attribute of that document. The precise semantics of data access and the granularity of data elements, especially for the purpose of concurrency control, vary between database implementations and we ignore these distinctions in this paper. A transaction may **read** a data item, consuming its state, or **write** to a data item, substituting its state with another. A transaction is **deterministic** if repeat applications over the same set of read inputs produce the same set of writes; i.e., the writeset of a transaction is a deterministic function of its readset.

Like elsewhere in the literature, we use the notation $r_i[x]$ and $w_i[x]$ to denote the execution of a read and write operation, respectively, by a transaction T_i on data item x .

The $\rightarrow_s$ precedence operator denotes a happened-before relationship, where $a \rightarrow_s b$ means that a 's occurrence clearly preceded b 's in some schedule S . If S is obvious from the discussion context, the subscript may be omitted. Formally, the precedence operator is a strict partial order binary relation that is irreflexive ($\neg(a \rightarrow a)$), transitive ($a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$) and asymmetric ($a \rightarrow b \Rightarrow \neg(b \rightarrow a)$).

For convenience, we also define a set-aware precedence operator $U \rightarrow_s V$, where all elements in U precede all elements in V in some schedule S . Formally, $U \rightarrow_s V \Leftrightarrow \forall u \in U, v \in V : u \rightarrow_s v$.

We use c_i and a_i to indicate the commit and abort operations, respectively, of T_i . When the terminal operation exists but is not specified to be one of commit or abort, it is denoted as e_i (ended). Once a transaction has ended, it cannot

introduce any further operations. Formally, a transaction T_i has a single end event e_i ; $e_i = c$ if the transaction committed; $e_i = a$ if the transaction aborted; and for every operation $p_i[x] \in \{r_i[x], w_i[x]\}$, $p_i[x] \rightarrow_i e_i$.

A **complete history** H over a set of transactions $\mathcal{T}$ is a partial order over $\mathcal{T}$ with relation $\rightarrow_H$, defined according to the following axioms:

1. **Projection of transaction-local precedence:** If T_i is in $\mathcal{T}$ and $p_i \rightarrow_i q_i$ then $p_i \rightarrow_H q_i$, for all pairs of events p_i, q_i .
2. **Connectivity of conflicting operations:** If T_i and T_j are in $\mathcal{T}$ then for every pair of conflicting operations $p_i[x], q_j[x]$, either $p_i[x] \rightarrow_H q_j[x]$ or $q_j[x] \rightarrow_H p_i[x]$.
3. **Deterministic aborts:** Let T_i, T_j be transactions in $\mathcal{T}$, where $e_i = a$. If $w_i[x] \rightarrow_H r_j[x]$ then either $e_i \rightarrow_H r_j[x]$ or $r_j[x] \rightarrow_H e_i$. I.e., either T_j read the value after T_i aborted or before T_i aborted; otherwise, the value read by T_j cannot be accurately determined.

A prefix of a partial order P over a set S is a partial order P' over a set $S' \subseteq S$ such that:

- If $q \in S'$ and $p \rightarrow_P q$ then $p \in S'$; and
- If $p, q \in S'$ then $p \rightarrow_P q \Leftrightarrow p \rightarrow_{P'} q$.

A **history** is any prefix of a complete history. We consider prefixes to reason about instantaneous state, where only some of the queued transactions have been fully executed, or even partially executed, for instance, in the event of a failure.

A history is represented as a partial order over a set of operations; e.g., the partial order $\{(r_2[x], w_2[z]), (r_2[y], w_2[z]), (w_2[z], c_2)\}$ over the operations $\{r_2[x], r_2[y], c_2\}$. The set of operations may be omitted as it is trivially inferred from the partial order. A partial order is sufficient to produce a directed graph (digraph) like the one in Fig. 1, where operations are vertices and edges denote precedence.

FIGURE 1. A digraph showing a partial order over a set of operations.

Let $S(H)$ be the set of all topological sorts of H . For the graph in Fig. 1, $S(H) = \{(r_2[x], r_2[y], w_2[z], c_2), (r_2[y], r_2[x], w_2[z], c_2)\}$.

If the digraph is of degree one uniformly throughout, then $|S(H)| = 1$ and the history can be written more compactly using a sequence; e.g., $(w_2[x], w_2[y], w_2[z])$.

Returning to the earlier informal description of a transaction, we now present a formal definition. A transaction T_i is a partial order with the strict ordering relation (irreflexive, transitive, asymmetric) $\rightarrow_i$, defined according to the following axioms (assume T_i reads and writes at most once to a data item, otherwise take $r_i[x]$ and $w_i[x]$ to mean the first read and the last write of x):

1. **A transaction may comprise reads, writes, commits and aborts:** $T_i \subseteq \{r_i[x] : x \text{ is a read item}\} \cup \{w_i[x] : x \text{ is a written item}\} \cup \{a_i, c_i\}$.
2. **A transaction must either commit or abort:** $a_i \in T_i \vee c_i \in T_i$.
3. **All non-terminal operations precede a terminal operation:** for a terminal operation $e_i \in \{a_i, c_i\}$, every other operation $p_i \in T_i \setminus \{e_i\}$, $p_i \rightarrow_i e_i$.
4. **Read and write operations do not commute:** for $r_i[x], w_i[x] \in T_i$, $r_i[x] \rightarrow_i w_i[x] \vee w_i[x] \rightarrow_i r_i[x]$.

A **schedule** is a partial order over a set of events. A history is a schedule. This paper uses schedules to refer to partial orders (and by extension, total orders) outside of transactional operations, for instance, to depict sequences of logical timestamps.

Let S and S' be two sets such that $S' \subseteq S$ and let P be a partial order over a set S . A **projection** of P on a set S' is a partial order P' (a subset of P) consisting of all elements in P involving only elements of S' .

For a complete history H (a partial order $\rightarrow_H$ over a set of transactions $\mathcal{T}$), a projection of H on a set of all operations of some transaction $T_i \in \mathcal{T}$, produces a partial order $\rightarrow_i$ over T_i , complete with all operations of T_i and void of all other operations. A (prefix) history, however, may project a partial history for some transaction that does not conform to the transaction axioms above.

The **commit projection** $C(H)$ of a history H is its projection on its set of committed transactions. Commit projections are used to erase the operations of aborted transactions.

B. CONFLICTS AND EQUIVALENCE OF HISTORIES

Two operations are said to **conflict** if they both operate on the same data item and at least one of them is a write; i.e., $r_i[x]$ conflicts with $w_j[x]$, whereas $w_i[x]$ conflicts with both $r_j[x]$ and $w_j[x]$, for all pairs of transactions T_i, T_j .

Note, while most databases support only reads and writes, several databases support additional operations on data types, for instance, atomic increment/decrement and set addition/subtraction operations of Redis [35], Couchbase [36], MongoDB [37], FoundationDB [38] and Apache Cassandra [39]. To avoid loss of generality, we expand the definition of conflicts and transactions to notionally include the extra operations, as appropriate, subject to a symmetric compatibility matrix that specifies which operations conflict. All examples and formal definitions here are limited to reads and writes; however, the results presented here are not limited to reads and writes; if necessary, they may be expanded upon to include the extra operations. I.e., our assumptions do not limit our results in any substantive way.

Informally, two histories are **view equivalent** if they exhibit the same residual effects after the effects of all aborted transactions have been erased. Formally, histories H and H' are view equivalent if:

1. They contain the same operations.
2. For every T_i, T_j such that $a_i, a_j \notin H$ (hence $a_i, a_j \notin H'$) and for every x , if T_i reads x from T_j in H , then T_i reads x from T_j in H' .
3. For every x , if $w_i[x]$ is the final write of x in H then it is also the final write of x in H' . The final write of x is the operation $w_i[x]$ such that $a_i \notin H$ and for all $w_j[x] \in H$ ($i \neq j$) either $w_j[x] \rightarrow w_i[x]$ or $a_j \in H$.

We also assume that, for all x , some initial value of x is written by a notional progenitor transaction T_{init} omitted from H (and hence H'). As per rule 2, if T_j reads the initial value of x (written by T_{init}) in H then it also reads the initial value of x in H' .

A more restrictive definition of equivalence is **conflict equivalence**, which states that two histories H and H' conflict are equivalent if:

1. H and H' are defined over the same set of operations; and
2. For their commit projections $C(H)$ and $C(H')$ respectively, the order of conflicting operations is identical in both $C(H)$ and $C(H')$. I.e., for pairs of conflicting operations $p_i[x], q_j[x] \in C(H)$, $p_i[x] \rightarrow_{C(H)} q_j[x] \Leftrightarrow p_i[x] \rightarrow_{C(H')} q_j[x]$.

Alternatively, H and H' are **conflict equivalent** if, for some pair of topological sorts, $G \in S(H)$ and $G' \in S(H')$, it is possible to arrive from G to G' through a series of swaps of pairs of adjacent nonconflicting operations in G .

All pairs of histories that are conflict equivalent are also view equivalent.

C. SERIAL HISTORIES, ISOLATION PROPERTIES AND SCHEDULERS

A **serial history** is one in which, for every transaction, the operations of that transaction are contiguous (not interleaved with the operations from any other transaction). Formally, for a serial history H , for all $T_i, T_j \in H$ ($i \neq j$), either $\forall p \in T_i, q \in T_j : p \rightarrow_H q$ or $\forall p \in T_i, q \in T_j : q \rightarrow_H p$. A serial history may be compactly represented by listing the transactions in their serial order; e.g., (T_1, T_2, T_3) . (Assuming the partial order of those transactions' operations is recorded elsewhere.)

An **isolation property** (or **isolation level**, as the terms may be used interchangeably) is a predicate I applicable to an execution history H , such that $I(H)$ iff (if and only if) the property holds.

As it is common elsewhere in the literature, a property's abbreviation is used as a surrogate for the set of all histories permitted by that property. For example, SR is an abbreviation for serializable and is also a surrogate for the set of all serializable histories. Formally, for an isolation property I , its abbreviation $\mathcal{I}$ is the set comprehension $\{H \in \mathcal{H} : I(H)\}$, where $\mathcal{H}$ is the set of all histories.

As preamble for the subsequent definitions, let $Start(T)$ and $End(T)$ resolve the real-time start and end timestamps, respectively, of a transaction T . $Start(T)$ and $End(T)$ are injective: $Start(T_i) = Start(T_j) \Rightarrow T_i = T_j$. Also, transactions take

some time to complete, hence $Start(T_i) < End(T_i)$. $Start$ and End are monotonic increasing with their arguments: if T_i started before T_j , then $Start(T_i) < Start(T_j)$, similarly if T_i ended before T_j , then $End(T_i) < End(T_j)$. The images of $Start$ and End do not intersect; i.e., there is no pair of transactions T_i and T_j for which $Start(T_i) = End(T_j)$.

$Start$ and End can be used to establish a real-time constrained precedence partial order $\rightarrow_{RT}$ over the set of transactions, such that $End(T_i) < Start(T_j) \Leftrightarrow T_i \rightarrow_{RT} T_j$. Note, there is no requirement that $Start$ and End faithfully mirror some reference global clock; only that they correctly impose a temporal order over events. In other words:

- Every timestamp is unique.
- No two timestamps are the same unless they represent the same event.
- A higher-valued timestamp occurs later³ than a lower-valued timestamp.

We say that a transaction T_i **precedes** another transaction T_j if T_j ended before T_i started; i.e., $End(T_i) < Start(T_j) \Rightarrow T_i \rightarrow T_j$.

Transactions are **concurrent** if their lifetimes overlap; i.e., $[Start(T_p), End(T_p)] \cap [Start(T_q), End(T_q)] \neq \emptyset$ for all pairs of concurrent transactions T_p, T_q .

Snapshot isolation (SI) is both an isolation property and a concurrency control mechanism that satisfies this property. Informally, it guarantees that all reads made in a transaction will see the most recent consistent snapshot of the database [24], [25]. Formally, if T_i reads a data item x , then x was written by a committed transaction T_j that precedes T_i , and there is no committed T_k ($j \neq k$) that also precedes T_i that wrote to x after T_j , and T_i either observes T_j 's write to x or its own write to x , whichever is more recent. In other words, writes by concurrent transactions that are active after T_i starts are not visible to T_i [7], [20].

Snapshot isolation prevents the **read skew** anomaly, in which a transaction observes an inconsistent view of the database. For example, suppose T_1 reads x , then T_2 updates both x and y . If now T_1 reads y , it will see an inconsistent state. SI prevents T_1 from observing T_2 's concurrent write to y .

Prefix-consistent snapshot isolation (PC-SI) is a generalisation of snapshot isolation that relaxes the timing constraint [23]. In the generalised variant, a transaction T_i observes a snapshot that is consistent with some prefix of operations involving committed transactions that precede T_i . Consider the precedence order of transactions T_1 – T_4 in the digraph in Fig. 2. The transactions comprise operations $(w_1[x], c_1)$, $(r_2[x], w_2[y], c_2)$, $(r_3[x], w_3[z], c_3)$ and $(r_4[y], r_4[z], c_4)$. T_2 and T_3 thus commute.

FIGURE 2. Transaction precedence for the snapshot isolation example.

Serial histories are topological sorts of the graph in Fig. 2. Assume some serial history H in which, $T_1 \rightarrow_H T_2 \rightarrow_H T_3 \rightarrow_H T_4$. In PC-SI, the set of admissible prefixes comprises $\{(), (T_1), (T_1, T_2), (T_1, T_2, T_3), (T_1, T_2, T_3, T_4)\}$. We later reference a weakened version of PC-SI in the assumed system model.

In its generalised form, snapshot isolation allows pathological orderings. In the most extreme example, the returned snapshot may be consistent with an empty history; i.e., a view of the database that corresponds to its initial state.

Serializability, informally, is an isolation property that states that a concurrent execution of transactions is serializable if its outcome is equal to some outcome in which those transactions execute serially. It serves as the foundation for reasoning about the consistency of a system in the presence of concurrent operations. Serializability is often presented as a necessary correctness condition in the design of concurrent systems.

A history is **(conflict) serializable (SR; is in SR)** if its commit projection is conflict equivalent to some serial history. The qualifier ‘conflict’ is typically omitted; serializable histories are assumed to be conflict serializable. A serial history is serializable.

A history is **view serializable** if the commit projection of every prefix of that history is view equivalent to some serial history. Clearly, from the earlier definition of equivalence, a conflict serializable history is view serializable.

A **blind write** occurs when a transaction writes a value without reading it. A serializable history that is not conflict serializable must contain a blind write.

A **scheduler** controls the concurrent execution of transactions [17]. In practice, a scheduler is a program or a set of programs that form a core part of a database system. A scheduler restricts the order in which read, write, commit and abort operations from different transactions are applied, such that the resulting history conforms to some *a priori* isolation property. Formally, $\mathcal{J}$ schedulers are schedulers that generate histories in $\mathcal{J}$, where $\mathcal{J}$ is the set comprehension of an isolation property I . For example, “serializable schedulers generate histories in SR,” or worded differently, “serializable schedulers generate serializable histories.”

Serializable histories permit certain pathological orderings of transactions in the equivalent serial histories. A serializable scheduler can reorder all transactions comprising only reads towards the beginning of the history, and all transactions

³ This is the only criterion in the entire formulation that relies on the passage of time. The easiest way to satisfy this is via a shared counter that is atomically incremented at each invocation of $Start$ and End .

comprising only writes to the end of the history. Thereby, in the most extreme (but plausible) scenario, all read transactions return the initial state for some value x and the effects of all write transactions on x are discarded. These, loosely speaking, “optimisations,” are not consciously employed in practice in database systems; however, in some databases (particularly in the distributed variety) these effects may inadvertently occur from time to time.

Consider a pair of transactions T_i and T_j that are processed serially (T_j is submitted only after T_i is decided) with no other interfering transactions. For some data item x written by T_i and subsequently read by T_j , $r_j[x]$ is not guaranteed to observe the effects of $w_i[x]$ in serializable histories. I.e., serializability does not offer any real-time, causal or even per-process guarantees.

Strict serializability is an isolation property that includes the serial-emulating property of serializability, in addition to imposing a real-time constraint on nonconcurrent transactions.

A history is **strictly serializable** (S-SR; is in S-SR) if its commit projection is conflict equivalent to a serial history H , and for all pairs of transactions T_i, T_j that are both in H , $End(T_i) < Start(T_j) \Rightarrow T_i \rightarrow_H T_j$.

Strict serializable histories do not permit the pathological orderings that serializable histories are prone to. Specifically, if transactions T_i and T_j are processed sequentially, then for some data item x written by T_i and subsequently read by T_j , $w_i[x] \rightarrow r_j[x]$.

The last isolation property considered here is **commitment ordering** (CO). For a history to be commitment ordered, the order of every pair of conflicting operations in every pair of committed transactions must match the order of their respective commit events [11]. Formally, a history is commitment ordered (CO; is in CO) if for all conflicting operations $p_i[x], q_j[x]$, for all committed transactions T_i, T_j , $p_i[x] \rightarrow q_j[x] \Rightarrow c_i \rightarrow c_j$. CO was devised for use in multidatabase systems and is a necessary and sufficient condition for guaranteeing global serializability [11]. We later refer to CO in one of the solutions presented in this paper.

D. RECOVERABILITY

A transaction T_i **reads from** a transaction T_j in H if, for some data item x , if:

1. T_i read x written by T_j ; i.e., $w_j[x] \rightarrow_H r_i[x]$;
2. T_j did not abort before T_i 's read; i.e., $e_j \rightarrow_H r_i[x] \Rightarrow e_j = c$; and
3. T_j was the last transaction to have written x ; i.e., $\forall T_k, j \neq k : w_j[x] \rightarrow_H w_k[x] \rightarrow_H r_i[x] \Rightarrow a_k \rightarrow_H r_i[x]$.

We write $T_i \vdash_H T_j$ if T_i reads from T_j , omitting the subscript from $\vdash$ if it can be inferred from the context.

A history H is **recoverable** if it guarantees that a transaction in H commits only after the commitment of all transactions (other than itself) from which it read; hence, there cannot be a case in which an uncommitted transaction rolls back a change that was irrevocably observed by a committed transaction. Formally, for a recoverable (REC; in REC)

history H , for a pair of transactions T_i, T_j ($i \neq j$) in H , $T_i \vdash_H T_j \Rightarrow e_j \rightarrow_H e_i \wedge (e_j = a \Rightarrow e_i = a)$.

Aborting T_j leads to a **cascading abort** of T_i if $T_i \vdash_H T_j$ and T_i is active at the time of T_j 's abort.

A history H **avoids cascading aborts** or is **cascadeless** if a transaction in H may only read values written by a committed transaction or by itself. Formally, for a cascadeless (ACA; in ACA) history H , then $T_i \vdash_H T_j \Rightarrow c_j \rightarrow_H r_i[x], i \neq j$.

A history is **strict** if no data item can be read from or overwritten until the transaction that previously wrote to it terminates. Formally, for a strict (ST; in ST) history H , then for T_i, T_j ($i \neq j$) where $T_i \vdash_H T_j, w_j[x] \rightarrow_H p_i[x] \Rightarrow e_j \rightarrow_H p_i[x]$, where $p_i[x] \in \{r_i[x], w_i[x]\}$.

The properties above are progressively stronger; i.e., $ST \subset ACA \subset REC$.

Serializability is orthogonal to recoverability. Specifically, SR intersects all of the sets REC, ACA, and ST, but is incomparable to any of them. All serial histories are in REC.

E. CONCURRENCY CONTROL METHODS

When a scheduler receives an operation, it has the choice of scheduling it immediately, delaying it (inserting it into a queue to schedule later) or rejecting it (thereby aborting the transaction).

Schedulers can be classified based on their propensity to delay or reject operations. An **aggressive scheduler** tends to schedule operations immediately, but in doing so, it foregoes the opportunity to reorder the operations later on. Conversely, a **conservative scheduler** tends to delay operations, which may lead to unnecessarily postponing operations that could be completed immediately. A serial scheduler is an extreme case of a conservative scheduler, in which the operations of all but one transaction are delayed.

Pessimistic concurrency control is a concurrency control method (a set of algorithms) that assumes that conflicts among transactions are frequent and thereby acts conservatively with respect to scheduling. Conversely, **optimistic concurrency control** assumes that conflicts are rare and schedules operations aggressively, looking for indications of conflicts and aborting transactions as appropriate.

Two-phase locking (2PL) is a pessimistic concurrency control method that guarantees serializability. Locks are applied and removed in two phases:

1. **Expanding phase:** locks are acquired, and no locks are released.
2. **Shrinking phase:** locks are released, and no locks are acquired.

2PL can be reduced to a single rule wherein a *lock is never acquired after it has been released*. 2PL is not strict; the basic protocol allows data items to be read from transactions that have not committed. 2PL (and its variants) are deadlock-prone, in that operations may be delayed to a point where none of the queued operations may proceed, pending other queued operations.

Strict two-phase locking (S2PL) is a strengthened variant of 2PL in which write locks held by a transaction are not released until it commits or aborts. As the name suggests, S2PL is strict: data items cannot be read from undecided transactions.

Strong strict two-phase locking (S-S2PL) is a further strengthening of S2PL in which both read and write locks are not released until the transaction terminates. The property gained by S-S2PL is commitment ordering. Note, both Microsoft SQL Server [32] and MySQL InnoDB [31] use S-S2PL in their implementation of the *Serializable* ANSI/ISO isolation level.

Multiversion concurrency control (MVCC) is a family of optimistic concurrency control methods that permit several versions of a data item to coexist, rather than substituting data items for their most recent updates. MVCC avoids contention by presenting a snapshot view of the database to concurrent transactions; changes made by one transaction will not be disclosed to others until the former commits, at which point (or earlier) transactions may be tested for conflicts.

Execution of transactions within a multiversion database can be represented with a multiversion serialization graph [30], with transactions as vertices, and edges indicating precedence in the apparent serial history. Three types of dependencies exist:

1. **write-read dependencies:** if T_i writes a version of a data item, and T_j reads that version, then $T_i \rightarrow T_j$.
2. **write-write dependencies:** if T_i writes a version of a data item, and T_j replaces that version with the next version, then $T_i \rightarrow T_j$.
3. **read-write antidependencies:** if T_i writes a version of a data item, and T_j reads the previous version, then $T_j \rightarrow T_i$, because T_j apparently missed T_i 's update.

Note, multiversion serialization graphs are intrinsically limited to MVCC methods. Pessimistic concurrency control (2PL, etc.) and some optimistic concurrency control methods do not permit antidependencies. This paper refers to multiversion dependencies in introducing serializable snapshot isolation, and later, in the main contribution.

Snapshot isolation (SI), in addition to being an isolation property, is an optimistic MVCC method that allows a transaction T_i to commit if T_i 's writes have not been superseded by another transaction T_j that committed after T_i 's snapshot was taken—the *first-committer-wins* rule. SI is not serializable, being prone to write skew anomalies. Note, despite the lack of serializability guarantees, SI is used by Oracle when the requested isolation level is *Serializable* [7], as it is compatible with the ANSI/ISO definition⁴.

Serializable snapshot isolation (SSI) is a strengthened variant of SI that relies on an abridged form of serialization

graph testing to identify potentially nonserializable executions [7], [26]. It is based on the work by Fekete *et al.* [7] that showed the presence of two consecutive read-write antidependencies in every offending cycle. Thus, rather than detecting the complete cycle, SSI detects a *dangerous structure* containing a pivot transaction T_{pivot} , and a pair of antidependencies coming into and out of T_{pivot} (possibly from the same transaction). SSI is conservative in that it prevents every nonserializable execution, while also preventing some executions that may in fact be serializable. SSI is used by PostgreSQL when the transaction isolation level is set to *Serializable* [29], reverting to SI for the *Repeatable Read* level. Fig. 3 demonstrates the antidependencies detected by SSI.

FIGURE 3. A dangerous structure in SSI.

Precise serializable snapshot isolation (PSSI) is another strengthened variant of SI that, unlike SSI, relies on complete serializable graph testing [28]. PSSI is more permissive than SSI in that it admits a larger set of histories, while still rejecting all nonserializable histories.

III. THE LOGICAL TIMESTAMP SKEW ANOMALY

We now present the **logical time skew (LTS)** anomaly.

A. PRELIMINARIES

We begin by classifying schedulers into two broad types.

A **deterministic scheduler** executes concurrent transactions in a repeatable manner, even if they conflict. Let $D(H)$ be the set of committed transactions in history H ⁵. A scheduler S is deterministic if, given identical transaction inputs (transactions' precedence partial order and operations), every pair of distinct executions of S produces histories H, H' where, if $D(H) = D(H')$ then H and H' are conflict equivalent.

Conversely, a **nondeterministic scheduler** permits an arbitrary ordering of concurrent transactions, such that a repeat execution of the scheduler produces histories that are not guaranteed to be conflict equivalent.

We initially focus on strict serializability as it is commonly labelled as the strongest among isolation properties that permit concurrent execution of transactions [3], [6]. Recall, strict serializability augments serializability with real-time

⁴ ANSI/ISO define serializability as an isolation level that forbids specific anomalies. This style of definition is inherently ambiguous and has been subject to much criticism [24]; it does not, for example, proscribe write skew behaviour. This differs materially from the definition provided by concurrency control theory, which formally constrains the set of admissible

execution histories. The ANSI/ISO definition allows database vendors more leeway in their implementations.

⁵ Equivalently, $D(T)$ is the set of transactions extracted from the commit projection $C(T)$.

constraints on nonconcurrent transactions. At a surface level, strict serializability appears to possess the necessary qualities of erasing the notion of concurrency from users' perception on one hand, while offering most⁶ of the benefits of concurrency on the other. Crucially, it should be noted that concurrent transactions are permitted to contain noncommutative operations, provided they can be reduced to some serial history, and nondeterministic schedulers permit several such histories. (It is these operations that are problematic, as we will later show.) We elaborate with several examples, which will be referred to throughout the rest of the paper.

Take a set of six transactions, T_1 – T_6 , subject to the precedence partial order in Fig. 4.

FIGURE 4. Precedence order of transactions for examples 1–3.

That is, T_1 and T_2 are genuinely serial— T_2 is submitted only after T_1 terminates. Transactions T_3 and T_4 are submitted concurrently, following the termination of T_2 . Then, after the last of T_3 and T_4 terminates, T_5 is submitted, then T_6 .

We are not concerned with collisions among T_1, T_2, T_5, T_6 : these transactions have a clearly defined happened-before relationship, and under strict serializability, their relative order is both *deterministic* and *apparent*. By ‘deterministic’ it is implied that repeat executions of these schedules will result in identical outcomes. In practice, serial transactions may naturally exhibit dependencies; however, these dependencies are faithfully reproduced, owing to the transactions’ one-at-a-time execution. By ‘apparent’ it is implied that their serial order is obvious to the user. This is true by the consequence of their serial submission.

Turning to T_3 and T_4 , there is no defined happened-before relationship, owing to their concurrency. There exists some period, however small, in which both transactions execute concurrently. We permit nondeterminism in their execution. Specifically, if T_3 and T_4 contain conflicting operations, then two identical S-SR schedulers S and S' may emit histories H and H' , respectively, that are both in S-SR but are not conflict equivalent.

We assume that the dependencies between the transactions are arranged such that, in at least one nonfaulty serial execution, all six transactions can commit.

Example 1: Assume that T_3 and T_4 do not conflict, so that there are no dependencies between them in any concurrent execution. This implies that T_3 and T_4 can, among themselves, form two serial sub-histories (T_3, T_4) and (T_4, T_3) . Both sub-histories are concatenated with the surrounding serial sub-histories to form the set comprising two strictly serializable

histories $\{(T_1, T_2, T_3, T_4, T_5, T_6), (T_1, T_2, T_4, T_3, T_5, T_6)\}$, assuming that all transactions committed. The resulting histories are conflict equivalent.

For the subsequent examples, we meddle with T_3 and T_4 to induce conflicts. We use $Readset(T)$ and $Writeset(T)$ to denote the exhaustive set of items read and written by T , respectively.

Example 2: Suppose the transactions were altered such that T_3 writes to a data item that is read by T_4 . Amidst some concurrent execution, a conflict between T_3 and T_4 presents, as $Readset(T_4) \cap Writeset(T_3) \neq \emptyset$. Under S-S2PL, one of the transactions will acquire a lock that is later blocked on by the other. Under SSI, a read-write antidependency from T_4 to T_3 will emerge. T_3 and T_4 are nonetheless serializable, but owing to the nondeterminism of conflicting operations, the resulting histories are not conflict equivalent.

T_3 and T_4 both appear in some histories in different order but are each absent from some other histories. Assuming both transactions commit, the resulting serial histories are $\{(T_1, T_2, T_3, T_4, T_5, T_6), (T_1, T_2, T_4, T_3, T_5, T_6)\}$. Crucially, the two histories are not view equivalent, despite containing the same transactions.

Note that some schedulers, for example, *timestamp ordering* [17], may abort one of the conflicting transactions. We consider the abort case separately in *Example 3*.

A study of *Example 2* reveals some intriguing behaviour under SSI. T_3 updates some value that T_4 has read from an earlier snapshot version and then commits, while T_4 remains active. Intuitively, T_4 appears to be doomed. On the contrary, the scheduler need not abort T_4 , provided the two are acyclic, because T_4 can “slot in” between T_2 and T_3 in some serial history and still respect the real-time constraint. This might appear counterintuitive but is nonetheless valid under S-SR⁷.

Example 3: The transactions are altered such that T_3 reads a value written by T_4 and vice versa. During some concurrent execution, the presence of one precludes the other. Under S-S2PL, each transaction may hold locks that are blocked on by the other. Sooner or later one transaction will be unilaterally aborted. Under SSI, a pair of read-write antidependencies between T_3 and T_4 emerges in opposing directions, forming a cycle in the multiversion serialization digraph, with one of the transactions in a committed state. Although T_3 and T_4 are not serializable in concert, one of the transactions can proceed without the other. This leaves us with the following histories: $\{(T_1, T_2, T_3, T_5, T_6), (T_1, T_2, T_4, T_5, T_6)\}$.

Note, the precise behaviour in *Example 3* is down to the timing of the operations. Clearly, there are runs in which both transactions will commit, in which case *Example 3* trivially reduces to *Example 2*. We intentionally limit *Example 3* to histories in which at least one transaction aborts; otherwise, it serves no purpose.

⁶ In all serializable models, some histories are rejected because they violate some correctness criteria, which invariably reduces the effective

concurrency factor and impacts throughput. This is an obvious and invariable consequence of correctness.

⁷ This is not permitted under Commitment Ordering, however.

B. SYSTEM MODEL

We model a simple distributed system comprising a strictly serializable **primary database**, an **atomic broadcast channel**⁸ and a **replica database** that conforms to snapshot isolation. A **primary process** operates over the primary database by reading and mutating its state, and atomically broadcasts event records encapsulating the state changes, once accepted by the primary database. Updates are delivered on the **replicator process**, which serially installs them to the replica database in their broadcast order.

The replication model is asynchronous, with no upper bound on message delays. For simplicity, we assume that processes never crash, and network links are nonfaulty. The atomic broadcast primitive ensures that messages are not lost or delivered out of order.

Next, we define **weak prefix-consistent snapshot isolation** (WPC-SI) to be a superset of PC-SI in which nonconflicting operations are allowed to commute. Consider the precedence order of transactions T_1 – T_4 in the digraph in Fig. 2, recalling that T_2 and T_3 commute. For the serial history (T_1, T_2, T_3, T_4) , the set of admissible prefixes under WPC-SI is $\{\emptyset, (T_1), (T_1, T_2), (T_1, T_3), (T_1, T_2, T_3), (T_1, T_2, T_3, T_4)\}$. I.e., T_2 and T_3 are mutually substitutable in WPC-SI. By comparison, the prefix (T_1, T_3) is illegal in PC-SI.

A **viewer process** reads the state of the replica database. We only require read consistency to a level that is at least as strong as WPC-SI. Recall, (conventional) snapshot isolation requires that transactions must see the latest snapshot when it starts execution. This is difficult to achieve in an asynchronous setting [23]. To allow the replica to trail the primary, we relax the real-time constraint, so that the viewer is not required to observe writes that are later than those installed locally by the replicator process. (This is the *prefix-consistent* constraint.) Furthermore, the replica may apply commuting updates out-of-order. (This is the *weak prefix-consistent* constraint.)

The primary process and the primary database reside in the **primary site**. The replicator process, the replica database and the viewer process reside in the **replica site**. The atomic broadcast channel is the only conduit by which the sites communicate. The sites may also be collocated for maximum generality. The system model is depicted in Fig. 5.

FIGURE 5. The system model.

The replica process may optionally be equipped with some transformation function that alters the inbound events prior to installing them, but preserves the semantics of the underlying data, so that the viewer process may take advantage of these transformations without having to apply equivalent transformations after each read. While a useful optimisation in practice, it has no material effect on this study; hence, we omit it from Fig. 5.

The model can easily be generalised to include multiple replica sites, each reflecting some distinct, weak prefix-consistent view of the primary, as illustrated in Fig. 6. It is for this reason that we label the replica sites with the subscript i in Fig. 5.

FIGURE 6. Example system with multiple replicas.

Databases are assumed to always satisfy the guarantees attributed to them; they never crash or lose data. We assume transactions are deterministic and side-effect free; a serial application of transactions always yields a reproducible

⁸ In practice, the atomic broadcast primitive is implemented using a persistent, order-sensitive message queue.

outcome, consistent with every other identically ordered serial application. While the model assumptions may appear overly simplistic, they are sufficient to present our case.

This system model is representative of contemporary event-driven architecture, while transforming between write-centric and read-centric models and offloading some reads to a replica is idiomatic CQRS [40]. We are confident that most practitioners will immediately recognise and identify with these patterns, for countless systems are built this way.

B. IDENTIFYING THE ANOMALY

We turn our attention to the sequence of event records emitted by the primary process. To meet the read consistency requirements on the replica, we only require that the updates installed on the replica are consistent with the observable history on the primary, preserving the partial order of nonconflicting updates. For that, the order of event records emitted by the primary must mirror some serial history. In other words, to maintain the requisite consistency on the replica, the order-sensitive effects of strong serializability must be projected externally in some precise way.

Consider how this might be implemented, at an outline level. To make our case, we assume that most practitioners will not distinguish between two serializable histories that are not conflict equivalent, nor dwell on serializability, for that matter; instead, relying on the correctness of the database system and the chosen isolation level to perform the heavy lifting. We assume that most practitioners will, given the choice, prefer to erase the notion of concurrency from their cognitive process, particularly when this is precisely what the tools of their trade claim. Their behaviour is perfectly rational in this respect.

A straightforward but naive⁹ way to satisfy the stated requirements is as follows.

Algorithm 1 Logical timestamping, execution and event production

For each in the (potentially unbounded) sequence of transactions:

- 1: Await the termination of the transaction's predecessors.
 - 2: Allocate a logical timestamp to the transaction.
 - 3: Submit the transaction to the scheduler. Upon termination, the transaction is internally marked as terminated.
 - 4.1: If the transaction aborted, disregard it and skip the remaining steps.
 - 4.2: Await the completion of all earlier transactions in logical timestamp order: either an abort outcome or the publishing of its event record.
 - 4.3: Publish the event record, disclosing the transaction's writeset and its logical timestamp. Upon acknowledgement from the atomic broadcast channel, the transaction is internally marked as published.
-

Alg. 1 is intentionally specified at a high level, to permit the broadest of implementations. In *Appendix A* of the

Supplementary Material accompanying this paper, we offer a more elaborate version.

Transactions are depicted with the literal T_i ; the subscript i signifying when the logical order of the transaction's arrival at the replica system. $\mathcal{T}$ is the set of all transactions on the primary. The replica process installs the writesets of the transactions serially, in their arrival order. There is an implied nontriviality constraint: the replica may only install updates from transactions in $\mathcal{T}$.

We define a logical timestamp $LTime(T)$ as a unary operation on a transaction T in $\mathcal{T}$. Applications of $LTime$ on pairs of transactions produce a total order, while adhering to the following criteria:

- (1) $\neg (LTime(T_i) < LTime(T_i))$
- (2) $LTime(T_i) < LTime(T_j) \wedge LTime(T_j) < LTime(T_k) \Rightarrow LTime(T_i) < LTime(T_k)$
- (3) $LTime(T_i) < LTime(T_j) \vee LTime(T_j) < LTime(T_i)$
- (4) $T_i \rightarrow T_j \Rightarrow LTime(T_i) < LTime(T_j)$
- (5) $LTime(T_i) = LTime(T_j) \Leftrightarrow T_i = T_j$

Criteria 1–3 mimic the usual irreflexive, transitive and connected properties.

Criterion 4 states that if a transaction T_i has a happened-before relationship directed at T_j , then the logical timestamp of T_i must be strictly less than the logical timestamp of T_j . This is a one-way implication, stating nothing about the logical inverse: if T_i 's timestamp precedes T_j 's, it cannot be claimed that $T_i \rightarrow T_j$ because if T_i is concurrent with T_j , their timestamps may be ordered arbitrarily.

Criterion 5 describes an equivalence between timestamp identity and transaction identity: two logical timestamps are identical *iff* the corresponding transactions are identical. A corollary from 5, logical timestamps are unique; no two distinct transactions may share a timestamp.

Step 1 of Alg. 1 enforces precedence order by ensuring that all transactions that have a happened-before relationship with some successor transaction either abort or commit before the successor is executed.

Step 2—the allocation of a logical timestamp—may be implemented in several ways. One such way is to maintain a monotonically increasing clock outside the transaction processor that is advanced for each incoming transaction in a manner that preserves the relative ordering of the transactions. Each increment of the clock may be durably persisted for fault tolerance. We do not concern ourselves with the particular implementation and assume that $LTime(T_i) \triangleq i$, being the

⁹ Naive only in the context of the assumptions made, which we argue are not unreasonable, given the circumstances.

simplest function that satisfies the abovementioned criteria for $LTime$.

Step 3 is trivial.

Step 4 waits for all preceding transactions to have been dealt with, before sending the event over the atomic broadcast channel. This is straightforward but suboptimal, as waiting for broadcast acknowledgements limits throughput. An alternative is to broadcast events in parallel and out-of-order, relying on the replica process to reassemble the events in logical timestamp order, assuming the timestamps are gap-free, and aborts have been communicated. The implementation details are irrelevant here, provided updates are installed in logical timestamp order on the replica process.

To finally highlight the anomaly, we now turn to our three cardinal examples. Particular attention should be paid to the relative order of the logical timestamps of T_3 and T_4 .

In *Example 1*, the admissible serial histories comprise the set $\{(T_1, T_2, T_3, T_4, T_5, T_6), (T_1, T_2, T_4, T_3, T_5, T_6)\}$. Both histories are conflict equivalent; hence, the relative order of T_3 and T_4 is immaterial to the replication outcome. The user does not know which particular history was emitted by the scheduler, nor do they care. As long as the partial order satisfying $T_1 \rightarrow T_2$, $\{T_2\} \rightarrow \{T_3, T_4\}$, $\{T_3, T_4\} \rightarrow \{T_5\}$, $T_5 \rightarrow T_6$ is preserved, the updates installed on the replica will leave it in a state that is weak prefix-consistent with the primary.

Skipping *Example 2* momentarily, we examine *Example 3*. While the cyclic dependency is interesting on the surface, it does not affect the outcome in the least. The set of admissible serial histories initially contains $(T_1, T_2, T_3, T_5, T_6)$ and $(T_1, T_2, T_4, T_5, T_6)$. Upon execution, however, only one history will be chosen. Note, all histories are void of one transaction—either T_3 or T_4 , owing to their co-dependence and the scheduler's anomaly detection (or avoidance) algorithm that aborts one or the other. Crucially, *each schedule is missing a unique transaction*. The behaviour of the scheduler is not necessarily deterministic—the choice of the aborted transaction is a function of the scheduler implementation and may be down to chance in some cases; however, the abort is trivially detected by step 4.1 of Alg. 1. Therefore, whichever the chosen history, the end result is anomaly-free.

Returning to *Example 2*. Due to the conflict between T_3 and T_4 , we are left with two histories that are not view equivalent but contain the same transactions. We immediately have a problem: if all 6 transactions commit, the algorithm has no obvious way of determining which of the two possible serial schedules was chosen. Both are fair game. It cannot simply pick one at random because they are incompatible.

Example 2 fails to capture precedence in concurrent transactions because the equivalent serial order is not clear before transactions commit. Assume that timestamps were allocated sequentially, such that for $k \in 1..5 : LTime(T_k) < LTime(T_{k+1})$. We have $LTime(T_3) < LTime(T_4)$, although in some dangerous execution of the algorithm, $T_4 \rightarrow_H T_3$ in the equivalent serial history H . Picking that execution, consider

how the update events are broadcast to the replica. The views on the replica assume the following progression:

TABLE I
INSTALLED EVENTS ON THE REPLICA IN EXAMPLE 2

Time on the replica	Updates installed on the replica
1	$\{T_1\}$
2	$\{T_1, T_2\}$
3	$\{T_1, T_2, T_3\}$
4	$\{T_1, T_2, T_3, T_4\}$
5	$\{T_1, T_2, T_3, T_4, T_5\}$
6	$\{T_1, T_2, T_3, T_4, T_5, T_6\}$

The replication algorithm produces an inconsistency at time 3 by attempting to serially install an update from T_3 before T_4 . Queries executed on the replica after time 3 may produce results that are inconsistent with the primary. This is the essence of the LTS anomaly, depicted in Fig. 7.

FIGURE 7. Logical timestamp skew in Example 2.

In contemplating the anomaly, one might assume that the inconsistency is only temporary and will be resolved at time 4 on the replica when both updates are installed. This is not necessarily the case. Recall, under the model assumptions, updates are installed serially on the replica. T_3 conflicts with T_4 . Perhaps T_4 is conditional on some invariant that T_3 has breached. In that case, if T_3 was installed before T_4 on the replica, then the local scheduler will abort T_4 on the replica, blocking replication. The inconsistency will not resolve itself; moreover, either manual intervention or an automated conflict resolution mechanism will be needed to remedy the situation.

Another interesting observation is that the anomaly is only triggered by noncommutative concurrent transactions that exhibit acyclic dependencies in the execution history. When concurrent transactions freely commute, the set of admissible schedules is relaxed to all permutations of the concurrent

transactions, and is anomaly-free. When transactions exhibit a dependency cycle, one of the transactions is aborted. If the cycle involved two transactions, then aborting one results in a user-inferable schedule, as per *Example 3*, and is anomaly-free. However, when a cycle involves three or more transactions and one of those transactions is aborted to break the cycle, the residual schedule may still result in an LTS anomaly.

Theorem 1. *The LTS anomaly is absent if the writes are issued serially.*

Proof. When writes to a serializable scheduler are issued serially, there is precisely one admissible history H —the order that the transactions were submitted in. Therefore, for all $T_i, T_j \in H$, if $LTime(T_i) < LTime(T_j)$ then $T_i \rightarrow_H T_j$. $\square$

On *Theorem 1*: Eliminating concurrency aligns the logical timestamps of transactions with their execution history, which obviously eliminates the logical timestamp skew. Needless to say, the resulting loss of write concurrency is detrimental to the performance of the system; it cannot be considered a viable solution except in the rarest of cases.

C. A REAL-LIFE SCENARIO

To describe the problem in plain English, we turn to an anecdotal example of a real-life system, on the good account of the author. It is set in a fintech organisation that is well-known for its rigorous (and by some accounts, draconian) hiring process that limits it to 99th percentile engineers, to use a much overstated and highly subjective criterion. The company enforces a code review process and retains several staff engineers who claim familiarity with the subject matter.

The system under study has been highly simplified to both avoid divulging sensitive material and to make the example approachable to a general audience. The reader might understandably question whether this scenario took place; nonetheless, it is easy to imagine this scenario and many others like it unfolding routinely, in the course of a somewhat fashionable push toward distributed computation, ostensibly to make systems more decoupled, scalable, resilient and, lest it be forgotten, agile.

The main transactional system tracks orders and executes trades in various financial markets. A market has a status of *OPEN* and *CLOSED*; trading can only occur in an open market and must cease once the market closes—the *no-more-trades* rule. The period between market open and close events is called a *trading session*. Reports are produced for the regulator, summarising each trading session. For simplicity, assume the regulator only cares about the *no-more-trades* rule. Trades and market status updates are streamed from the primary database to a replica, using a durable message queue, in a manner comparable to Alg. 1. As per contemporary event-driven architecture practices, operational reporting occurs on the replica.

Assume a pair of transactions T_{300} and T_{400} executing on the primary database, depicted in pseudo-SQL in Listing 1 and 2.

Listing 1 Transaction T_{300}

```
-- T300 --
1: BEGIN
2: UPDATE market SET status = 'CLOSED'
3: SELECT 'market closed'
4: COMMIT
```

Listing 2 Transaction T_{400}

```
-- T400 --
1: BEGIN
2: status ← SELECT status FROM market WHERE ...
3: IF status = 'OPEN' THEN
4:   INSERT INTO trades VALUES (...)
5:   SELECT 'trade entered'
6: ELSE
7:   SELECT 'trade rejected'
8: END IF
9: COMMIT
```

Simultaneously to the abovementioned transactions, transaction T_{999} (Listing 3) executes on the replica database.

Listing 3 Transaction T_{999}

```
-- T999 --
1: BEGIN
2: status ← SELECT status FROM market WHERE ...
3: count ← SELECT count(*) FROM trades WHERE ...
4: SELECT concat(status, ' with ', count, ' trades')
5: COMMIT
```

Assume an initial state with 100 trades before T_{400} . Consider the set of serializable outcomes for valid permutations of the above transactions, depicted in Table II.

TABLE II
ADMISSIBLE OBSERVATIONS IN THE TRADING EXAMPLE

#	Serial order	Outputs (unordered)
1	$T_{300}, T_{400}, T_{999}$	'market closed', 'trade rejected', 'closed with 100 trades'
2	$T_{300}, T_{999}, T_{400}$	'market closed', 'closed with 100 trades', 'trade rejected'
3	$T_{400}, T_{300}, T_{999}$	'trade entered', 'market closed', 'closed with 101 trades'
4	$T_{400}, T_{999}, T_{300}$	'trade entered', 'open with 101 trades', 'market closed'

5	$T_{999}, T_{300}, T_{400}$	'open with 100 trades', 'market closed', 'trade rejected'
6	$T_{999}, T_{400}, T_{300}$	'open with 100 trades', 'trade entered', 'market closed'

Observations containing the following outputs are considered to be illegal, as they do not conform to any admissible serial order: $\{\text{'closed with 100 trades'}, \text{'trade rejected'}\}$ and $\{\text{'closed with 101 trades'}, \text{'trade entered'}\}$. Logically, either the trade proceeds before the market is closed—in which case the report accounts for it, or the trade is rejected afterwards—in which case the report omits it. Either way, the report must consistently reflect the status of the market and the trade count, by the definition of weak prefix-consistent snapshot isolation.

We can now see nondeterministic scheduling in all its glory: with the read-only transaction erased, there are two distinct sets of conflict equivalent serializable histories: $\{\#1, \#2, \#5\}$ and $\{\#3, \#4, \#6\}$. (Refer to row numbers in Table II.) Both sets contain only correct histories, in that none violate the *no-more-trades* invariant; however, they are not mutually equivalent.

Transactions T_{300} and T_{400} in this scenario are the practical analogues of T_3 and T_4 in *Example 2*. Suppose they were assigned logical timestamps 300 and 400, respectively; i.e., $T_{300} \rightarrow_L T_{400}$ in the logical timestamp order L . The scheduler on the primary may sequence them such that in the resulting serial history H , $T_{400} \rightarrow_H T_{300}$. This produces the outputs $\{\text{'trade entered'}, \text{'market closed'}\}$; hence, we expect there to be 101 trades in the recent session.

On the replica, writes are applied in strict logical timestamp order, meaning that the market appears to close before the trade is entered. If T_{999} executes between the applications of T_{300} and T_{400} on the replica, the result is *'closed with 100 trades.'*

IV. SOLUTIONS TO THE LOGICAL TIMESTAMP SKEW ANOMALY

This section offers a thorough analysis of the anomaly, the conditions which are susceptible to it and the conditions which are anomaly-free. It subsequently presents several diverse solutions to the anomaly.

A. ANALYSIS

The naive application of logical timestamps in the replication of transactional updates can lead to an LTS anomaly, even under strict serializability, as demonstrated earlier. The anomaly is triggered by a reordering of concurrent transactions by the scheduler due to conflicts. It cannot be easily corrected or accounted for, because strict serializability (and by implication, serializability) is only concerned with producing a history that is consistent with some prominent ordering (real-time constrained serial or plain serial, depending on the model strength). Strict serializability and its weaker variant are not obligated to reveal the resulting

schedule to the user or to behave in a manner where the user might deterministically infer the schedule.

This leads to an important question. Is strict serializability sufficiently strong to isolate transactions such that serial behaviour is completely emulated across multiple schedulers?

The answer is a categorical *no*. In the general case, strict serializability is a necessary but not sufficient criterion to completely emulate serial execution with multiple schedulers. Consider Alg. 1: would the resulting anomaly have occurred had the primary scheduler been truly serial? No, because it has been shown by *Theorem 1* that an LTS anomaly cannot occur in a system comprising only serial writes.

To clarify, it is not claimed that strict serializability has a subtle flaw that makes it behave differently from its stated guarantees, nor is the LTS anomaly attributed solely to strict serializability. We claim that the guarantees offered by strict serializability in one scheduler are insufficient to correctly emulate serial execution in an event-replicated system. This is substantiated with a single counterexample that highlights a material difference in the behaviour of two systems—one operating serially and another emulating serial execution. This difference was later exploited to break an invariant on an external system that would have otherwise been preserved in the serial case. In that respect, our observations are harmonious with the result of Gligor and Popescu-Zeletin [18] that targets atomically coordinated multidatabase systems.

Notably, a serializable database does not need to know the final serial history in order to uphold its stated guarantees. Its only commitment is that serializability anomalies do not occur, which can be accomplished either with S-S2PL in systems with pessimistic concurrency control, or with anomaly detection in systems with optimistic and multi-version concurrency control. For the latter, cycle detection is both necessary and sufficient to identify anomalies [7]. There are some MVCC systems, such as PostgreSQL [29], that conservatively detect serializability anomalies by looking for *dangerous structures* [7]. On top of anomaly detection (graph testing) or avoidance (locking), a system that conforms to strict serializability also needs to honour the real-time ordering constraint. A serializable system is not required to retain the final serial history, although it typically tracks the execution history in a commit log, and from that history, a serial history may be derived. Moreover, if the database observes commitment ordering (as in all S-S2PL systems), the schedule is trivially derived by observing the commit events [11], [12].

We also stress that the “linear reasoning” instilled by proponents of serializability may be precarious under certain circumstances. It erases a subtle, yet crucial point in the serializability guarantee; namely, that concurrent executions are not necessarily commutative and may be reordered to conform to a prominent schedule. When the system in its entirety comprises a single (strict) serializable scheduler, then operations against that scheduler will adequately emulate seriality while the underlying serial history can safely remain

concealed. However, the information emitted by the system may not appear serializable to an external agent.

The anomaly only applies to systems comprising multiple schedulers. If the read-only transaction (like the one in the trading example) is colocated on the primary scheduler, the resulting history is serializable, even if the read-only transaction executes after T_3 commits but before T_4 commits, in the case where $T_4 \rightarrow T_3$ in the serial history and the transactions commitment order is reversed. (This is possible under SSI due to lack of commitment ordering.) *Appendix B* of the *Supplementary Material* elaborates.

C. SOLUTIONS WITHIN STRICT SERIALIZABILITY

In this subsection, we explore whether strict serializability on its own, perhaps with some enhancements, can be *reasonably* used to eliminate the LTS anomaly; that is, without introducing undue complexity or resulting in significant performance degradation.

1) PRETESTING TRANSACTIONS FOR CONFLICTS

In one solution, all concurrent candidate transactions could be tested for the possibility of conflicts before submitting them to the scheduler, identifying T_3 's conflict with T_4 in *Example 2* and either executing them serially or eliminating one of the transactions. This implies emulating some of the intricate machinery already present in serializable schedulers—a nontrivial proposition for all but the most knowledgeable and committed of users. This method also assumes that the transactions' operations are known in advance or can be easily determined, which calls into question its feasibility. Nonetheless, pretesting transactions for conflicts would have identified and prevented the LTS anomaly.

Theorem 2. *A history void of conflicting concurrent transactions is LTS anomaly-free.*

Proof. For all pairs of concurrent, nonconflicting transactions $T_i, T_j, i \neq j$, all histories differing only in the order of T_i 's and T_j 's operations are conflict equivalent. Therefore, if $LTime(T_i) < LTime(T_j)$ then in some conflict equivalent serial history, $T_i \rightarrow T_j$. $\square$

Returning to the contract between the user and the database system, the overarching supposition is that practitioners, even moderately skilled in the art, will revert to the “linear reasoning” model because it is convenient to do so. The viable solutions to the anomaly problem should take users' skills into account.

With only strictly serializable isolation at their disposal, and in the absence of schedule disclosure, this solution is essentially asking the user to perform dynamic analysis on their transactional workload stream to reduce the set of

histories *before* submitting transactions for execution. We believe that it is extremely unlikely that many practitioners would *a priori* identify this as a pressing concern, let alone commit to addressing it at this level. The lack of awareness around the LTS anomaly, and others like it, is not unexpected on the practitioners' part, given the confidence instilled by strict serializable systems.

2) SERIALIZABLE COUNTERS

It is known that autoincrementing fields and sequences are generally exempt from serializability checks¹⁰, rendering them useless for logical timestamp generation in concurrent transactions that exhibit conflicts, as shown in *Example 2*. Could a logical timestamp be generated with serializable semantics? Suppose each transaction in $\mathcal{T}$ increments some shared integer field f by reading it and writing back the updated value. Thus, for every transaction $T_i \in \mathcal{T}$, $\{r_i[f], w_i[f]\} \subseteq T_i$.

The observed read of f equates to a logical timestamp that reveals the serial schedule to the user. Unfortunately, this method severely restricts the concurrency level in both pessimistic and optimistic concurrency control systems. Under S-S2PL, and assuming *SELECT FOR UPDATE* is used, access to f will lock it until the transaction completes, blocking other transactions when they reach this point. (The locking schedule without *SELECT FOR UPDATE* is more complex and may lead to deadlocks due to lock upgrades.) The S-S2PL regime imposes a penalty on all concurrent transactions in $\mathcal{T}$ that touch f , not just the normally conflicting ones. Under SI (and SSI), updates to f will conflict, and for n conflicting concurrent transactions, $n - 1$ will be forced to retry. Again, the penalty is indiscriminate.

Theorem 3. *Taking a logical timestamp from a serializable counter prevents the LTS anomaly.*

Proof. As previously noted, $\forall T_k \in \mathcal{T} : \{r_k[f], w_k[f]\} \subseteq T_k$. As T_i conflicts with T_j , in every serial history H containing both committed transactions, either $T_i \rightarrow_H T_j$ or $T_j \rightarrow_H T_i$. In both cases, $T_i \rightarrow_H T_j \Leftrightarrow LTime(T_i) < (T_j)$ and $T_j \rightarrow_H T_i \Leftrightarrow LTime(T_j) < LTime(T_i)$. $\square$

Due to its limitations, this method cannot be recommended except in the rarest of cases where concurrency is a nonrequirement. One might assume that it is easier to just submit transactions serially, knowing they will be executed serially all the same. Suppose, however, that transactions were submitted from separate processes. In the absence of inter-process coordination, forcibly serializing transactions through the scheduler becomes a viable option.

¹⁰ That is, among the database products that we are aware of.

3) PROMISING TO DISCLOSE THE SERIAL ORDER

Intuitively, every solution must possess some knowledge of, or influence over the serial schedule—either to derive logical timestamps that are ordered equivalently to the serial history, correct noncompliant timestamps, or otherwise abort transactions that do not conform to the timestamp schedule. This compels us to revisit the concept of nondeterministic schedulers.

Theorem 4. *Some run of Alg. 1 will produce an LTS anomaly in all schedulers that permit nondeterministic concurrent execution of transactions.*

Proof. Suppose a nondeterministic serial-emulating scheduler S is LTS anomaly-free. S executes transactions according to a precedence partial order P . Transactions T_i and T_j are concurrent (in P) and conflicting. Assume that for some execution of Alg. 1, $LTime(T_i) < LTime(T_j)$; i.e., $T_i \rightarrow_L T_j$ for some logical timestamp schedule L . S admits several non-equivalent serializable histories containing both T_i and T_j . Pick one such history H in which $T_j \rightarrow_H T_i$. We have a contradiction in which $T_i \rightarrow_L T_j$ and $T_j \rightarrow_H T_i$; therefore, S cannot be LTS anomaly-free. $\square$

One viable solution within strict serializability involves changes to the scheduler to disclose at least one serial schedule that is conflict equivalent to the serial execution history. Ideally, this is a total order that maps directly to logical timestamps. In the simplest such implementation, *COMMIT* conveniently returns the logical timestamp to the user.

There are glaring problems with this approach, particularly in the context of relational and other SQL databases:

1. *COMMIT* would need to return a logical timestamp, contrary to the current SQL standard.
2. Transactions may be implicit, requiring potentially all DML statements to return a logical timestamp in lieu of an explicit *BEGIN–COMMIT* block.
3. The serial order should be recoverable sometime after the transactions commit; for example, if the issuing process crashes after committing T and later needs to determine T 's position in the serial schedule.
4. The scheduler may not know the serial order to begin with. S-S2PL systems are commitment-ordered [11] and the serial history can be trivially determined as such. Also, some MVCC systems (e.g., Bohm) impose a total order on transactions prior to execution [13]. Most MVCC (e.g., SSI and PSSI) systems, however, do not know the serial history—only that the history is serializable.
5. The serial order of a transaction may not be known at the point of commitment; it may remain undetermined for some time, until all concurrent transactions commit.

Both points 1 and 2 require changes that are not backwards compatible with the ANSI/ISO SQL standard, posing serious logistical problems for implementers. Perhaps the DML statements communicate the serial order in some other way... This could be via a built-in function; for example, *SELECT serial_order()*. Naturally, this would only work for explicitly demarcated transactions. The latter also solves point 3: the result of calling this function could be used in a subsequent *UPDATE* statement, thereby persisting the logical timestamp. Assuming point 4 is addressable without a significant performance impact¹¹, the real trouble with returning a logical timestamp within a transaction is point 5—it may be unknown even after commitment, particularly in some MVCC systems where transactions may be subject to more aggressive reordering [11]. The obvious way of solving this is to abort offending transactions—a crude measure that is costly, throughput-wise. Another way is to delay the commits until the order is known, but that fails to satisfy point 3—one cannot use an unknown value inside of a transaction.

The less-obvious way is to return a **promise**—an opaque reference that materialises at some point after the transaction commits, when the serial order has been unequivocally determined. We refer to this as the **maturity point**. A promise requires a new data type, for example, *serial_promise*—that can be used for transporting the timestamp's reference over the database connection, assignment between statements, and eventual persistence. For the latter, assume that in addition to updating the shared state, every transaction also writes to a dedicated tuple, keyed by the corresponding transaction ID, where the promise may be durably persisted to. After the promise matures, it can be resolved, revealing the logical timestamp. (It is not implied that the promise must be persisted for it to be resolved; however, persistence is a likely use case.) Moreover, owing to its durability, the promise may be queried by any process, at any time in the future. Listing 4 shows how one might use promises.

Listing 4 Use of promises in SQL databases

```
-- get a promise and persist it
1: INSERT INTO txn VALUES (t_id, serial_capture())
-- later, resolve the promise
2: SELECT serial_resolve(txn.promise)
   FROM txn WHERE txn.id = t_id
```

A promise obtained via a built-in function, for example, *serial_capture()* is later resolved by presenting it to another built-in function, for example, *serial_resolve(serial_promise)*. Assume a transaction T_i to which the promise applies. The resolution may block if the serial order of T_i is as yet undermined at the time of the call. T_i may not have committed,

¹¹ The reality is that the cost of determining the serial order is highly implementation-dependent. S-S2PL and pre-ordered MVCC schemes will likely fare better than others.

or there may exist a transaction T_j that precedes T_i in the serial history and T_j has not terminated.

To elaborate, CO-serializable schedulers (e.g., the S-S2PL variety) are guaranteed not to block due to pending transactions, provided the resolution attempt was made after T_i committed. Some MVCC-based strict serializable schedulers (e.g., SSI) may block even after T_i 's commitment, if T_i contains a non-empty writeset and there are concurrent transactions with T_i that are still in flight. Unless T_i is read-only, some other transaction T_j that overlaps with T_i may, at any point after T_i 's commitment, present a read-write antidependency directed upon T_i , implying that either $T_j \rightarrow T_i$ in the serial history or that T_j must unilaterally abort. Finally, in both pessimistic and optimistic concurrency regimes, the resolution will fail if T_i was aborted.

Alg. 1 can now be easily amended to take advantage of promises, producing Alg. 2.

Algorithm 2 Amendment of Algorithm 1 to use promises

For each in the (potentially unbounded) sequence of transactions:

- 1: Await the termination of the transaction's predecessors.
 - 2: Record the transaction's submission order.
 - 3: Submit the transaction to the scheduler. Upon termination, the transaction is internally marked as terminated.
 - 4.1: If the transaction aborted, disregard it and skip the remaining steps.
 - 4.2: Await the completion of all earlier transactions in submission order: either an abort outcome or the publishing of its event record.
 - 4.3: Resolve the logical timestamp of the transaction.
 - 4.4: Publish the event record, disclosing the transaction's writeset and its logical timestamp. Upon acknowledgement from the atomic broadcast channel, the transaction is internally marked as published.
-

The changes are captured in step 2 and steps 4.2–4.4. Under the revised algorithm, the logical timestamp remains unassigned until sometime after the transaction's commitment. In Alg. 2, step 4.3 may block.

D. SOLUTIONS WITHIN COMMITMENT ORDERING

Commitment ordering, or CO-serializability, is a strengthening of strict serializability such that the serial history is compatible with the order of transactions' commit events [11]. The question is whether this strengthened model is conducive to avoiding the anomaly.

As previously established, Alg. 1 is LTS anomaly-prone in all serial-emulating schedulers that permit nondeterministic execution. We require a modified version of the algorithm that either constrains the serial ordering or resolves it. A CO-serializable scheduler guarantees that the serial order of a transaction is established before commitment [11]. If the scheduler can communicate this order to the user process by some means, then the problem is trivially solved—we adapt Alg. 2 to use the scheduler-disclosed order. Note, because CO-serializability implies strict serializability, our promise-based solution is equally applicable here.

What if the scheduler does not explicitly communicate the serial schedule? Most CO implementations we know of are S-S2PL, being common among relational databases that support the serializable isolation level [31], [32], and the latter are not especially known for expressly communicating the serial schedule [14]. A CO-serializable scheduler implicitly communicates the serial schedule in the order of its commit operations; however, for an algorithm to exploit this, it must have a precise way of measuring the relative order of those operations. In other words, the user process must establish causality among commit events inside the scheduler.

Real-time clocks are not particularly useful for inferring causality. First, transactions might be submitted by multiple asynchronous or semi-synchronous processes. Second, the order in which commit operations are acknowledged on the user process may not be the same as their true order in the scheduler. Assume, for example, that transactions T_i and T_j are both blocked at the user process, awaiting completion from the scheduler. The scheduler may commit T_j before T_i , but the user process might observe T_i 's commit operation returning first and wrongly assume that $T_i \rightarrow T_j$.

An alternative approach is to inspect the commit log for the presence of commit operations [12]. Whether this is practical largely depends on the users' circumstances. Tailing the commit log requires an additional process deployed alongside the database server with read access to the underlying storage subsystem, and the user may simply lack sufficient access privileges at that level. The commit log is also proprietary—for example, MySQL's *binlog* differs from PostgreSQL's *WAL*; therefore, this solution is not portable. Furthermore, the format of the commit log may change between database versions, making it brittle. Finally, the log tailing option may not be available altogether; for example, in Cloud-hosted SaaS products, where the database system is fully managed and precludes low-level interference from the user.

On first thought, it may appear that a user cannot precisely measure the commitment times of concurrent transactions without somehow “peeking” into the scheduler or the commit log. But as it turns out, we can exploit a certain property of a specific CO scheduler to obtain a total order of commit events.

Recall, some write operations are inherently conflict-free; for example, sequences cannot cause conflicts and are exempt from serialization graph testing. For that reason, they neither block (in S-S2PL and weaker schedulers), nor lead to aborts or reordering (in optimistic schedulers). Such operations do not interfere with transaction execution, despite having observable and even persistent side effects. Not only are they reasonably performant, but also tremendously useful for instrumenting transactions, as it turns out.

We define a **nondiscord operation** to be one that either does not conflict with another nondiscord operation, or conflicts in a way that makes no material difference to the perceivable outcome. For example, two transactions might each insert a distinct item into a common collection and conflict on the underlying index page. Another example is

when two transactions modify distinct data items, but due to lock coarsening¹², they appear to conflict. In a third example, two transactions append distinct items to an array that is treated as an unsorted set by the user process. Although the persistent outcomes differ depending on their relative order, the end-effect does not. In these examples, transactions appear to have a precedence order in the serial schedule but are anomaly-free if their declared effects are applied in reverse order. Restating the earlier definition, a pair of transactions are nondiscord if they can be applied in either order to achieve outcomes that are equivalent although not necessarily identical.

We now define a **subcritical region** to be any contiguous block of inherently nondiscord statements in a transaction between the last conflict-prone operation and the *COMMIT* statement, as per Listing 5 (the area enclosed by lines 51–98).

Listing 5 A subcritical region

```

1:  BEGIN
    -- start of conflict-prone operations
2:  SELECT ...
3:  INSERT ...
    -- more conflict-prone operations
    ...
    ...
50: UPDATE ...
    -- end of conflict-prone operations
    -----
    -- subcritical region
    -----
99:  COMMIT

```

Each transaction is augmented with a sequence increment operation in its subcritical region, for example, *SELECT nextval('ser_order')*, where *ser_order* is an existing sequence. The resulting scalar equates to a logical timestamp that is consistent with commitment order for a subset of conflicting transactions that established a precedence order prior to their subcritical regions. To persist the timestamp, the augmented statement may be revised, for example, to *UPDATE txn SET order = nextval('ser_order') WHERE txn.id=t_id*¹³.

Nonconflicting transactions may enter the subcritical region simultaneously, as there nothing prevents one transaction from venturing into its subcritical region before some other nonconflicting transaction commits. As a result, their logical timestamps may disagree with the serial order. This is of no consequence as the transactions commute—the histories are conflict equivalent. Also, transactions may conflict in the subcritical region; however, this implies that they did not

conflict before the subcritical region (where it matters) and their later conflicts are nondiscord.

This method works with an S-S2PL scheduler by a simple consequence of its locking protocol. In the shrinking phase, locks are not released until the point of termination; therefore, without loss of generality, should a transaction T_i overtake another transaction T_j in accessing some data item d , upon attempting to access d in a way that fails to satisfy lock compatibility¹⁴ T_j will be blocked until T_i terminates. When T_i enters its subcritical region, T_j will be blocked in some operation that precedes T_j 's subcritical region. At this point, there is a materialised $T_i \rightarrow T_j$ relationship that persists until either transaction terminates¹⁵. T_i will atomically increment the sequence before T_j . When T_i eventually commits, T_j will be unblocked and will eventually arrive in its subcritical region, locking out its successors, and so on.

Regrettably, this technique only works on S-S2PL, relying on a combination of resource locking and the shrinking phase of 2PL to function as a synchronization barrier. It may be justly said that this technique is also brittle, leaning on an implementation subtlety, which may change with little notice, rather than on the stable guarantees of the isolation level.

Generally, every conflict serializable schedule can be made CO without aborting any transaction, by delaying commit events to comply with the transactions' precedence partial order. Although most CO schedulers are S-S2PL based, it is conceivable that an MVCC system may offer commitment ordering. In such systems, CO requires deferring the commit operation to align with the serial schedule but does not block the rest of the statements [11]. The subcritical region technique, however, relies on mutual exclusion; it loses its efficacy in systems based on optimistic concurrency control because conflicting data access operations no longer block.

E. SOLUTIONS WITHIN DETERMINISTIC ISOLATION

The solutions presented thus far were based on aligning the logical timestamp total order with the execution history, ensuring the two are conflict equivalent; the main problem being accurately determining the serial schedule in the absence of a transparent scheduler. (We considered both modifying the scheduler to disclose the serial order and instrumenting transactions to deduce the serial order.) We now contemplate the reverse: a scheduler that produces histories that are conflict equivalent to the user-specified logical timestamp total order, transferring the responsibility of alignment from the user to the scheduler.

All history-constraining isolation properties thus far were *nonparametric*. Schedulers producing nonparametric histories operate only on the supplied set of transactions, accepting no additional instructions from the user.

¹² A concurrency control optimisation in which multiple fine-grained locks are merged into a coarse-grained lock to minimise the number of lock objects that must be maintained.

¹³ The PostgreSQL dialect is used here for sequence manipulation.

¹⁴ A necessary and sufficient condition for transactions to conflict in S-S2PL.

¹⁵ T_j may terminate even in a blocked state if, say, it times out or trips a deadlock detector.

We now define a parametric isolation property that is stronger than strict serializability, named **external order serializability**, or simply **externalizability**. We believe it to be the first isolation property to be defined in such manner; i.e., parametrised. Externalizability is orthogonal to commitment order; while both are stronger than strict serializability, neither is stronger than the other, in that each may permit histories barred by the other.

We first recall some earlier definitions before formally introducing externalizability.

A history is serializable (SR) if its commit projection is conflict equivalent to some serial history.

A history is strictly serializable (S-SR) if its commit projection is conflict equivalent to a serial history H , and for all pairs of transactions $T_i, T_j \in H$, $\text{End}(T_i) < \text{Start}(T_j) \Rightarrow T_i \rightarrow_H T_j$.

Let $\mathcal{X}$ be the set of all strict total order binary relations (irreflexive, transitive, connected) over $\mathcal{T}$. I.e., every relation $<_X \in \mathcal{X}$, for every triplet of transactions $T_i, T_j, T_k \in \mathcal{T}$, satisfies the properties:

(irreflexive) $\neg (T_i <_X T_j)$

(transitive) $T_i <_X T_j \wedge T_j <_X T_k \Rightarrow T_i <_X T_k$

(connected) $T_i <_X T_j \vee T_j <_X T_i$

A history is externalizable (XTN($<_X$); is in XTN($<_X$)) if its commit projection is conflict equivalent to some serial history H in S-SR, and all transactions in H are ordered by $<_X$. I.e., for all pairs of transactions T_i, T_j that are both committed in H , $T_i <_X T_j \Rightarrow T_i \rightarrow_H T_j$.

Theorem 5. *Externalizability is stronger than strict serializability for all external total orders; i.e., $\forall <_X \in \mathcal{X} : \text{XTN}(<_X) \subset \text{S-SR}$.*

Proof. It suffices to show that (1) $\text{XTN} \subseteq \text{S-SR}$ and (2) S-SR permits schedules that are not in $\text{XTN}(<_X)$, for some $<_X \in \mathcal{X}$.

For (1), the obligation is a consequence of the definition of XTN. It only permits conflict equivalence with S-SR and no new histories are introduced.

For (2), a history is in S-SR if its commit projection is conflict equivalent to some serial history that satisfies the real-time precedence partial order. Pick a pair of concurrent transactions T_p and T_q , which are exempt from the real-time constraint because $\text{Start}(T_q) < \text{Start}(T_p) < \text{End}(T_q) \vee \text{Start}(T_p) < \text{Start}(T_q) < \text{End}(T_p)$. Let the pair conflict, such that $T_q \rightarrow T_p$ in some serial histories in S-SR that include both committed transactions. Pick G to be one such history. Let $T_p <_X T_q$, implying that T_p must precede T_q in all serial histories in XTN containing both committed transactions. Thus, G exists in S-SR but is absent from XTN. $\square$

Theorem 6. *An externalizable scheduler is deterministic.*

Proof. Let $D(H)$ be the set of committed transactions in history H . It suffices to show that every history H produced by an externalizable scheduler S is conflict equivalent to every other history H' produced by S for the same set of inputs, if $D(H) = D(H')$. For every pair of committed transactions T_p and T_q in H , either $T_p <_X T_q$ or $T_q <_X T_p$. Without loss of generality, take $T_p \rightarrow_H T_q$. This implies that either (1) $T_p <_X T_q$, or (2) T_p does not conflict with T_q . For (1), this implies that $T_p \rightarrow_{H'} T_q$ by the definition of externalizability; thus, H and H' are conflict equivalent. For (2), H is implicitly conflict equivalent with H' . $\square$

We now define another useful isolation property, **first-in serializability**, abbreviated to **Φ -serializability**, that is a special case of externalizability where the parameter $<_X$ is bound to the real-time order of start events. I.e., $T_i <_X T_j \triangleq \text{Start}(T_i) < \text{Start}(T_j)$. Conversely, externalizability is a generalisation of Φ -serializability.

A history is Φ -serializable (Φ -SR; is in Φ -SR) if it is in some $\text{XTN}(<_X)$, where $\text{Start}(T_i) < \text{Start}(T_j) \Rightarrow T_i <_X T_j$.

Remarkably, the strength of Φ -serializability is such that Alg. 1 becomes trivially correct with no modifications.

Before contemplating the complete proof, consider the scheduling behaviour under Φ -serializability for our canonical examples. First, we lead with a lighter proof.

Theorem 7. *If an execution constrained by an isolation property M is A anomaly-free, it remains A anomaly free under M' that is at least as strong as M .*

Proof. If the set of histories in M does not contain some history H that admits A , then $M' \subseteq M$ implies that $H \notin M'$. Therefore, M' is A anomaly-free. $\square$

Both *Example 1* and *Example 3* were anomaly-free under strict serializability, and will remain as such under every isolation property that is at least as strong as strict serializability, as per *Theorem 7*.

In *Example 1* the concurrent transactions T_3 and T_4 are conflict-free; therefore, every pair of histories in SR over the same committed transactions is conflict equivalent, and the relative serialization order of T_3 and T_4 is immaterial. (S-SR and Φ -SR permit both histories because the transactions commute.)

In *Example 3*, in all serializable histories where T_3 and T_4 are cyclically dependent, possibly through transitive dependencies with other transactions, only one will be committed; therefore, their relative serialization order is once again irrelevant. If no cycle is formed and T_3 and T_4 both commit, then *Example 3* reduces to *Example 2*.

Example 2 produces the dangerous serial history $(T_1, T_2, T_4, T_3, T_5, T_6)$ under strict serializability, which is incompatible with the logical timestamp order assigned by Alg. 1. In other

words, in some histories in S-SR containing both committed transactions, $T_4 \rightarrow T_3$. Under Φ -serializability, if $T_3 <_X T_4$, then every admissible history containing both committed transactions must be conflict equivalent to a serial history H , such that $T_3 \rightarrow_H T_4$. Note, some histories in Φ -SR may contain just one transaction or none.

Theorem 8. *The LTS anomaly cannot occur under Φ -serializability when transactions are submitted in logical timestamp order.*

Proof. Transactions are submitted in logical timestamp order L , such that for all pairs of transactions T_i, T_j , $T_i \rightarrow_L T_j \Leftrightarrow \text{Start}(T_i) < \text{Start}(T_j)$. Every history in Φ -SR containing committed T_i and T_j is conflict equivalent to some serial history H wherein $\text{Start}(T_i) < \text{Start}(T_j) \Rightarrow T_i \rightarrow_H T_j$. Thus, $T_i \rightarrow_L T_j \Leftrightarrow T_i \rightarrow_H T_j$, a necessary and sufficient condition for LTS anomaly freedom. $\square$

Theorem 8 presents a weaker result than indiscriminately claiming that the anomaly cannot occur in every XTN($<_X$), or even more generally, under every deterministic scheduler. It was established in *Theorem 4* that the LTS anomaly can occur in every nondeterministic scheduler. The logical inverse, sadly, does not lead to a contrapositive in this case, because an XTN($<_X$) scheduler respects some total order $<_X$ but not necessarily the logical timestamp order. It is easy to imagine a scenario where such a claim would be void. Suppose a fictitious externalizable scheduler orders transactions according to a topological sort of their precedence partial order, in which concurrent transactions are further ranked by the lexicographical order of some unique, user-defined identifier, $\text{Id}(T)$. Then for a pair of transactions T_i, T_j , where $\text{Id}(T_j) < \text{Id}(T_i)$ every history containing both T_i and T_j in its commit projection will be conflict equivalent to some serial history H wherein $T_j \rightarrow_H T_i$, contradicting $T_i \rightarrow_L T_j$, where L is the logical timestamp order.

Theorem 9. *Some run of Alg. 1 will produce an LTS anomaly in some schedulers that permit deterministic concurrent execution.*

Proof. By the counterexample presented earlier. $\square$

The practicality of Φ -serializability over externalizability is immediately apparent because the former is parameterless. Φ -serializability can be retrofitted to any standards-compliant scheduler without breaking prior interface contracts, unlike generalised externalizability, which requires additional instructions (logical timestamps or some other ordering) from the user. Conversely, externalizability is a more flexible property that could be exploited with some interesting ordering; however, that would require a more committed and aware user.

V. RELATED WORK

This section recounts related work in the field of distributed transaction processing and concurrency control, relating the results of this paper to other contributions that preceded it.

A. MULTIDATABASE TRANSACTIONS

Externally observable anomalies in serializable systems have been known for several decades and were the subject of several notable studies. Gligor and Popescu-Zeletin [18] were perhaps the first correspondents of this in 1985. In particular, anomalies were observed in multidatabase systems wherein a central coordinator serializes transactions using an atomic commitment protocol and multiple subordinate databases acquire resource locks and subsequently install updates at the coordinator's instruction. The basic problem is explained very well by Georgakopoulos *et al.* in [14]. We recount their explanation here with minor clarifications and expansions, with the aid of Fig. 8, which has been adapted from the authors' original account.

FIGURE 8. Global serialization failure in multidatabase transactions.

Consider two global (multidatabase) transactions G_1 and G_2 that are executed by a centralised coordinator, operating over two autonomous databases L_1 and L_2 , by way of an atomic commitment protocol such as two-phase commit. Global transactions have local subtransactions in L_1 and L_2 . Additionally, a local transaction T_1 executes on L_2 . In Fig. 8, an arc from a data item x to a transaction G_i denotes that x is read by G_i ; conversely, an arc from G_i to x denotes that G_i writes to x .

In L_1 , G_1 writes a and G_2 reads it. Due to the conflict in L_1 , $G_1 \rightarrow G_2$. In L_2 , G_1 and G_2 access different items: G_1 reads b and G_2 writes c . T_1 writes b and reads c ; thus, G_1 and G_2 conflict indirectly in L_2 . The resulting histories are:

On L_1 : $(w_{G1}[a], r_{G2}[a])$, implying that $G_1 \rightarrow G_2$.

On L_2 : $(w_{T1}[b], r_{G1}[b], w_{G2}[c], r_{T1}[c])$, implying that $G_2 \rightarrow T_1 \rightarrow G_1$ ¹⁶.

Intuitively, the problem may be summed up as follows. Since the coordinator has no knowledge of local transactions and the indirect conflicts they may cause, it cannot interfere, for example, to induce a conflict between G_1 and G_2 . Also, the

¹⁶ The history on L_2 is conflict equivalent with the serial history $(w_{G2}[c], w_{T1}[b], r_{T1}[c], r_{G1}[b])$.

subordinates are *autonomous*, meaning no concurrency control information is shared with others, except via the atomic commitment protocol. Consequently, coordinators are unaware of the discrepancies between each other's serial schedules.

Georgakopoulos *et al.* demonstrate that the anomaly occurs even if G_1 and G_2 are executed serially [14]. Indeed, considering the histories above, there is nothing suggesting that G_1 and G_2 are concurrent. Since the pair does not conflict, the serializable schedulers in the subordinates permit their reordering.

We now consider a special case of the multidatabase problem in which subordinates are strict serializable, to verify that the problem persists under stronger model assumptions. It does in fact, but only if G_1 and G_2 are concurrent. Intuitively, there is a clear happened-before relationship between the pair, hence the pair will be scheduled in real-time precedence order on L_2 . Consequently, (G_2, T_1, G_1) cannot be a valid serial history.

Work in this area has been voluminous, presenting solutions that include stronger isolation properties (i.e., Commitment Ordering in [11]), inducing conflicts within the subordinates [14], disclosing the serial order [15], using strict schedules [19] (which was later shown to be inadequate [14]).

This paper shares some similarities with the (multidatabase) research above, in that both sets of problems relate to discrepancies in serial orders across multiple schedulers. Notably, in the multidatabase scenario, the presence of a centralised coordinator and a synchronous atomic commitment protocol is assumed, and the commit/abort decision is based on the subordinate states. The differences are in the following:

1. A centralised coordinator in the multidatabase scenario may be equated to the *primary process* and the *primary database* in this paper. The multidatabase scenario employs a synchronous coordination protocol for the subordinate databases, in which the subordinates actively partake in decision making. By comparison, the anomaly presented in this paper relies on nothing more than asynchronous replication, in which the replicas are passive processes that vacuously reproduce the events from the primary. This also implies that the LTS anomaly is inherently latent; i.e., it may be observed sometime in the future, potentially long after writes on the primary have quiesced.
2. In the multidatabase scenario, serialization anomalies are surfaced in workloads that contain disjoint writesets incident upon multiple subordinate databases. The specific anomalies described therein are not possible where the writeset is incident upon a single database. By comparison, the anomaly presented in

this paper is possible even when the writeset concerns a single database.

3. Subordinates in the multidatabase scenario are autonomous and may process read-write transactions outside of the coordinator's instructions, which is the main reason that the serial schedules disagree in the first place. The anomaly presented in this paper is observable on a replica that only processes writes at the primary's instruction; the remaining transactions are read-only or nonexistent¹⁷.

Both sets of works demonstrate anomalies that are external to the serializable scheduler. In both cases, the respective anomalies would not have been observable if all transactions occurred within the confines of a single serializable scheduler.

B. CONCURRENCY CONTROL ALGORITHMS

Those familiar with concurrency control theory will notice that Φ -serializability shares similarities with the Timestamp Ordering (TO) technique [17], [21]. The latter is a concurrency control mechanism that emits histories satisfying $Start(T_i) < Start(T_j) \Rightarrow p_i[x] \rightarrow q_j[x]$ for a pair of conflicting operations $p_i[x]$, $q_j[x]$ of transactions T_i and T_j , respectively. The basic algorithm is specified below.

Each data item x is assigned separate read and write timestamps, denoted by $RTime(x)$ and $WTime(x)$, respectively.

A transaction T reading from x checks:

- That $Start(T) < WTime(x)$, if so, T aborts, rolling back all prior writes;
- Otherwise, T proceeds with the read and assigns $Max(Start(T), RTime(x))$ to $RTime(x)$.

A transaction T writing to x checks:

- That $Start(T) < RTime(x)$, if so, T aborts, rolling back all prior writes;
- That $Start(T) < WTime(x)$, if so, T skips the write;
- Otherwise, T proceeds with the write and assigns $Start(T)$ to $WTime(x)$, taking note of the prior x value.

The difference between TO and Φ -serializability is in the level of abstraction. TO is a family of algorithms in the class of optimistic concurrency control. Also, basic TO is not recoverable without the addition of the strictness constraint. Φ -serializability, on the other hand, is an isolation property. Relating the two, we can now say that a strict TO scheduler produces histories in Φ -SR.

¹⁷ In Example 2, T_4 preceded T_3 in serialization order but was reversed on the replica. If T_3 breached some invariant that T_4 relied upon, replication

may have ceased after installing T_3 ; thus, the anomaly is surfaced without resorting to other transactions on the replica.

VI. SUMMARY AND CONCLUSIONS

This paper classifies transaction schedulers into two types: *deterministic schedulers* that always produce conflict equivalent histories given the same set of transaction inputs, and *nondeterministic schedulers* that may produce arbitrary histories that conform to some overarching isolation property.

We show that a nondeterministic concurrent scheduler is insufficient for ensuring that replicated systems are anomaly-free. This claim has been substantiated with a counterexample: a scenario in which a strict serializable scheduler produces a history that cannot be serially applied to a replica database in logical timestamp order, without the loss of weak prefix consistency. The anomaly is named *logical timestamp skew* (LTS), after the conditions under which it occurs. The anomalous scenario initially targets S-SR schedulers; it is subsequently shown to apply indiscriminately to all nondeterministic schedulers, while being absent from all nonconcurrent schedulers and some deterministic ones.

The identified anomaly has far-reaching implications. The system model described herein is used broadly in the construction of contemporary distributed systems, for example, event-driven microservices and similar architectures that rely on event streaming, stream processing, command-query responsibility segregation, state machine replication, CQRS, OLAP applications, event sourcing, and so on. Many of these systems can now be labelled as “broken” as a result of this paper.

One key takeaway of this paper is that practitioners ought to remain sceptical of highly generalised claims that casually extrapolate from important results in concurrency control theory without a justifiable theoretical basis, particularly when those results are composited over multiple schedulers.

In particular, we refer to strict serializability, which states that the resulting history’s commit projection is conflict equivalent to some serial history constrained by the transactions’ real-time precedence partial order. The supposition that we find most concerning, yet one that is abundant in the industry [8], [22], [27], [33], [34], is that strict serializability is a touchstone for correctness, alluding to it perhaps being correct in the general case¹⁸. We already know this to be false, owing to the research in multidatabase systems and global serializability [11], [14], [15], [18], [19]. We now reveal this to be also false in event-replicated schedulers.

Several solutions are proposed to avoid the anomaly, ranked here by the strength of the isolation property to which they apply (starting with the weakest).

Under strict serializability:

1. *Analyse transactions for potential conflicts before submitting transactions to the scheduler*, eliminating histories that do not conform to the logical timestamp

order. While this approach does not require changes to the scheduler, it is highly involving on the user’s part.

2. *Use serializable counters*. The performance of this method is comparable to serial execution; nonetheless, it may be used when performance is a nonrequirement and in the absence of inter-process coordination.
3. *Capture the serial schedule using promises that can be resolved at some future point after the transaction commits*. This requires substantial changes to the scheduler in favour of relatively few changes in the user’s code.

Under commitment ordering: *Exploit the ‘subcritical region’ technique for instrumenting transactions to determine their serial order*. This technique only works with S-S2PL schedulers and requires relatively few changes to the user’s code. It cannot be ported to other scheduler implementations.

Finally, this paper demonstrates that concurrency is a necessary but not sufficient factor in inducing the LTS anomaly; some concurrency-permitting deterministic schedulers are LTS anomaly-free. We present a parametric isolation property, *externalizability*, that arranges conflicting transactions according to some *a priori* total order. Externalizability is subsequently refined to produce Φ -serializability—a nonparametric special case that preserves the transactions’ logical timestamp order. We show that the system is LTS anomaly free when the primary scheduler is Φ -serializable.

We close with a remark: There are serious problems with the “linear thinking” approach that is endemic to the industry. It is easy to abstract oneself away from the underlying model when the surface claims are both simple and convenient, and when the underlying details are complex and esoteric. Such behaviour is rational and even advantageous if the claims are accurate, and their limitations are well-known and articulated. This is not always the case, however, and we suggest that designers of complex systems should exercise great care and diligence in adopting lesser models.

REFERENCES

- [1] J.M. Falcão, A. Thompson and D.J. Abadi, “Lazy Evaluation of Transactions in Database Systems,” in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 15–26 Jun. 2014.
- [2] K. Kingsbury, “Consistency Models,” available: <https://jepson.io/consistency>, accessed 27 Jun. 2021.
- [3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J.M. Hellerstein and I. Stoica, “Highly Available Transactions: Virtues and Limitations,” *Proceedings of the VLDB Endowment*, vol. 7, issue 3, pp. 181–192, Nov. 2013.
- [4] M. Herlihy and J.M. Wing, “Linearizability: A Correctness Condition for Concurrent Objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12(3), pp. 463–492, Jul. 1990.
- [5] P. Bailis, “When is ‘ACID’ ACID? Rarely,” available: <http://www.bailis.org/blog/when-is-acid-acid-rarely/> (accessed 27 Jun. 2021),

¹⁸ These sources fail to mention the adverse effects of combining multiple strict serializable schedulers.

- Jan. 2013.
- [6] P. Viotti and M. Vukolić, "Consistency in Non-Transactional Distributed Storage Systems," *ACM Computing Surveys*, vol. 49:1, Dec. 2015.
 - [7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha, "Making snapshot isolation serializable," *ACM Transactions on Database Systems*, vol. 30:2, pp. 492–528, 2005.
 - [8] Google LLC. Cloud Spanner, "TrueTime and external consistency", <https://cloud.google.com/spanner/docs/true-time-external-consistency>, accessed 27 Jun. 2021.
 - [9] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates," *Journal of the ACM*, vol. 26:4, pp 631–653, Oct. 1979.
 - [10] D. Gifford, "Information Storage in a Decentralized Computer System," *Published by Xerox Corporation Palo Alto Research Centers*, 1981.
 - [11] Y. Raz, "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment," *VLDB '92: Proc. of the 18th International Conference on Very Large Data Bases*, pp. 292–312, Aug. 1992.
 - [12] K. Daudjee and K. Salem, "Inferring a Serialization Order for Distributed Transactions," *22nd International Conf. on Data Engineering*, IEEE, Apr. 2006.
 - [13] J.M. Faleiro and D.J. Abadi, "Rethinking serializable multiversion concurrency control," *Proc. of the VLDB Endowment*, vol. 8:11, pp. 1190–1201, Jul. 2015.
 - [14] D. Georgakopoulos, M. Rusinkiewicz, and A.P. Sheth, "On serializability of multidatabase transactions through forced local conflicts," *Proceedings of the Seventh International Conference on Data Engineering*, pp. 314–323, 1991.
 - [15] C. Pu, "Superdatabases for composition of heterogeneous databases," *Proc. of the Fourth International Conf. on Data Engineering*, pp. 548–555, Feb. 1988.
 - [16] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6:2, pp. 213–226, 1981.
 - [17] P.A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.
 - [18] V. Gligor, R. Popescu-Zeletin, "Concurrency Control Issues in Distributed Heterogeneous Database Management Systems," *Distributed Data Sharing Systems*, North Holland, pp. 43–56, 1985.
 - [19] A. Wolski and J. Veijalainen, "2PC Agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase," *Proceedings of PARBASE-90 Conference*, Feb. 1990.
 - [20] A. Fekete, E. O'Neil and P. O'Neil, "A read-only transaction anomaly under snapshot isolation," *ACM SIGMOD Record*, vol. 33:3, pp.12–14, 2004.
 - [21] R. Shapiro and R. Millstein, "Reliability and fault recovery in distributed processing," *OCEANS '77 Conference Record*, pp. 425–429, 1977.
 - [22] D.J. Abadi and M. Freels, "Serializability vs 'Strict' Serializability: The Dirty Secret of Database Isolation Levels," available: <https://fauna.com/blog/serializability-vs-strict-serializability-the-dirty-secret-of-database-isolation-levels> (accessed 27 Jun. 2021), Feb. 2019.
 - [23] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Generalized Snapshot Isolation and a Prefix-Consistent Implementation," *Proc. of the 24th IEEE Symposium on Reliable Distributed Systems*, 2004
 - [24] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil. "A critique of ANSI SQL isolation levels," *Proc. of the ACM SIGMOD International Conference on Management of Data*, ACM, New York, pp. 1–10, Jun. 1995.
 - [25] A. Cerone, G. Bernardi and A. Gotsman, "A Framework for Transactional Consistency Models with Atomic Visibility", *Proc of the Concur Conference*, Madrid, Spain, Sep. 2015.
 - [26] M.J. Cahill, "Serializable Isolation for Snapshot Databases," *ACM Trans. on Database Systems*, vol. 34:4, pp. 1–42, Dec. 2009.
 - [27] E. Weaver, "A Comparison of Scalable Database Isolation Levels," Fauna website, available: <https://fauna.com/blog/a-comparison-of-scalable-database-isolation-levels>, accessed 27 Jun. 2021.
 - [28] S. Revilak, P. O'Neil and E. O'Neil, "Precisely serializable snapshot isolation (PSSI)," *Proc. of the 27th Int. Conf. on Data Eng.*, pp. 482–493, Apr. 2011.
 - [29] D.R.K. Ports and K. Grittnr, "Serializable Snapshot Isolation in PostgreSQL," *Proc. of the VLDB Endowment*, vol. 5:12, pp. 1850–1861, Aug. 2012.
 - [30] A. Adya, B. Liskov and P. O'Neil, "Generalized isolation level definitions," *Proc. of the 16th Int. Conference on Data Engineering*, pp. 67–80, Feb. 2000.
 - [31] MySQL 8.0 Reference Manual, "InnoDB Transaction Isolation Levels," available: <https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html>, accessed 27 Jun. 2021.
 - [32] Microsoft SQL Server 2019 Documentation, "Transaction Locking and Row Versioning Guide," available: <https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver15> (accessed 27 Jun. 2021), Oct. 2020.
 - [33] A. Matei, "CockroachDB's Consistency Model", available: <https://www.cockroachlabs.com/blog/consistency-model/> (accessed 27 Jun. 2021), Jan. 2019.
 - [34] A. Colyer, "Distributed Consistency and Session Anomalies," available: <https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies> (accessed 27 Jun. 2021), Feb. 2016.
 - [35] Redis Reference Manual, "INCR key operator," available: <https://redis.io/commands/INCR>, accessed 27 Jun. 2021.
 - [36] Couchbase Reference Manual, "Sub-document API," available: <https://docs.couchbase.com/server/5.1/developer-guide/sub-doc-api.html>, accessed 27 Jun. 2021.
 - [37] MongoDB Reference Manual, "\$inc operator," available: <https://docs.mongodb.com/manual/reference/operator/update/inc/>, accessed 27 Jun. 2021.
 - [38] FoundationDB Features, "FoundationDB Atomic Operations," available: <https://apple.github.io/foundationdb/features.html#atomic-operations>, accessed 27 Jun. 2021.
 - [39] Apache Cassandra Reference Manual, "Data Types," <https://cassandra.apache.org/doc/3.11.3/cql/types.html#counters>, accessed 27 Jun. 2021.
 - [40] A. Debski, Bartłomiej Szczepanik, M. Malawski, S. Spahr and D. Muthig, "A Scalable, Reactive Architecture for Cloud Applications," *IEEE Software*, vol. 35:2, pp. 62–71, doi: 10.1109/MS.2017.265095722, Mar. 2018.

EMIL KOUTANOV received his B.E. degree *summa cum laude* from the University of Technology Sydney (Australia), in 2007. He is a consulting software architect and part-time researcher.

His professional career has spanned several diverse industries including wagering, telecommunications, construction and finance, focusing on the design and implementation of reliable and scalable distributed systems. He has authored several articles on open-source middleware, application design and architecture, as well as a reference book on the subject of event streaming. His research interests include the analysis and design of distributed algorithms and models of computation, fault-tolerance and continuous availability, formal methods, concurrent and parallel computing, consistency models and consensus.