IRON: Internalizing Real-time Ordering to Verify Strictly Serializable Databases

by

Manudeep Herle

May 12, 2024

A thesis submitted to the

Faculty of the Graduate School of

the University at Buffalo, The State University of New York

in partial fulfillment of the requirements for the

degree of

Master's in Computer Science

Department of Computer Science and Engineering

Acknowledgments

I would like to express my deepest gratitude to Dr. Haonan Lu for his invaluable guidance and support throughout my research. His insights and expertise were instrumental in the development of this thesis. I am especially grateful for his course on Distributed Systems, which became one of my favorite subjects and provided a solid foundation for my research. Professor Lu also taught me the principles of conducting rigorous and meaningful research, for which I am profoundly thankful.

Table of Contents

List of Figures

Abstract

As distributed databases have become the norm in most software systems, the need for very strong consistency guarantees has become paramount to avoid business errors, prevent potential security exploits, and simplify application development. This paper explores the challenges of verifying strict serializability in distributed databases, which are complicated by the inherent limitation of physical clocks and network latencies. These challenges prevent reliable verification of transaction orders across different nodes of the database. Traditionally used methods like dependency graphs fall short due to their inability to account for external events that may take place between two transactions and their dependency on unreliable timestamps associated with transactions.

This research introduces IRON, a novel method designed to enhance the verification of strict serializability without relying on assumptions about clock skew or incomplete dependency graphs. By assuming the presence of an external event between any two consecutive transactions on different nodes and simulating these events through an internal message passing channel, IRON creates a comprehensive dependency graph that can highlight any real-time ordering constraints.

Our contributions include an analysis of existing research in this area and the introduction of IRON. This work lays the groundwork for building a workload-agnostic, reliable, and robust verification tool for strongly consistent distributed databases.

Chapter1

Introduction

1.1 Context

As applications have grown, databases have evolved from single-server systems running SQL to geo-replicated, multi-server architectures offering diverse guarantees related to consistency, scalability, availability, and transaction support. However, distributed databases come with some challenges. For instance, when two requests go to two different database servers at almost the same time, figuring out which request should be executed first is non-trivial simply because the two database servers may have different notions of time. This is due to the imperfect nature of physical clocks (clock skew) which makes them slower or faster over time. Synchronization protocols do exist but due to the non-zero latency of networks, they cannot ensure true time on all clocks at all times. This brings the need for consistency models. Consistency models dictate the order in which distributed systems execute parallel user requests arriving at different database nodes.

In the ideal case, we'd like to order transactions according to true time, i.e. transactions that are completed first should be ordered before transactions that start after. This true time ordering paired with transaction isolation is the strictest form of consistency called strict serializability. Strict serializability dictates that every transaction appears to be executed in isolation and respecting the real-time order. i.e. every transaction sees the results of all previously completed transactions. It's important to mention here that when two transactions are running at the same time, they are called concurrent. Concurrent transactions may be ordered in any way by the database without violating strict serializability. A slightly weaker consistency model is linearizability which enforces true time ordering, but on non-transactional workloads. Serializability and sequential consistency are weaker forms of consistency that guarantee a total order but not true time ordering. Serializability applies to transactional workloads and sequential consistency to non-transactional workloads. Although strict serializability is difficult to achieve and often results in lower throughput, it offers various benefits. Strict serializability provides an abstraction of programming in a single-threaded environment, simplifying application development by ruling out concurrency bugs.

Developers trust that the underlying database provides the consistency it advertises when building applications. However, if the database fails to deliver the promised consistency level, the system could yield incorrect results, potentially leading to bugs and security vulnerabilities. Therefore, it is crucial to verify the consistency levels of systems. Formal verification tools (Eg. TLA+) and dependency graphs are two popular methods to evaluate the consistency of distributed databases.

Dependency graphs represent a history of transactions where each transaction is a node on the graph, and edges represent relations or dependencies between transactions. Any cycles in the dependency graph mean that the transaction history doesn't have a total order. i.e. a cyclic dependency graph implies that different database nodes maintain different execution orders of transactions. An acyclic graph confirms that the transaction history has a total order. When we bring in the edges representing real-time order constraints, dependency graphs can be used to ascertain if the corresponding transaction history is strictly serializable (Or linearizable for non-transactional workloads).

However, existing methods to verify strictly serializability (and linearizability) are inadequate. Works based on dependency graphs [[1], [2]] use timestamps associated with transactions to infer real-time ordering constraints in dependency graphs. To deal with clock skew they propose adding a clock skew threshold. i.e. a specified amount of time depending on the average clock skew is added to the commit timestamp of transactions, and maybe even subtracted from the start timestamp of transactions. This amount of time by which the timeline of the transaction is expanded is called the *clock skew threshold*. However, these techniques are flawed, primarily because of these reasons: a) Adding units of time (clock skew threshold) to commit timestamps may make sequential transactions concurrent. This will result in the verifier failing to detect some or all consistency violations. I.e. this results in false negatives. b) If the clock skew is greater than the clock skew threshold, concurrent transactions may be viewed as sequential transactions by the verifier leading to false positives. I.e. The verifier wrongly flags consistency violations when there aren't any.

Formal verification tools like TLA+ are protocol model checkers and have limited implications on the correctness of the implementation and deployment of the said models. Thus a more robust and flexible technique to verify strict serializability is needed.

1.2 Design insight

Two seemingly unrelated transactions may be related through an external event. These external events imply that the corresponding transactions are sequential. i.e. some units of time exist between the end of one transaction and the beginning of the other during which time the external event occurs. Including these external events in the dependency graph will impose real-time ordering constraints needed to verify strict serializability (or linearizability). However, due to the nature of the external event, it's not possible for the database or any verifier to know about it. To make up for this, timestamps are often used to enforce real-time ordering constraints in the dependency graphs. We saw how the use of timestamps could lead to false positives and false negatives in the previous subsection. Thus we need another approach to add real-time ordering constraints to the dependency graphs.

If we could internalize this external event our dependency graph will be complete and also overcome the pitfalls associated with the use of timestamps. This is our design insight.

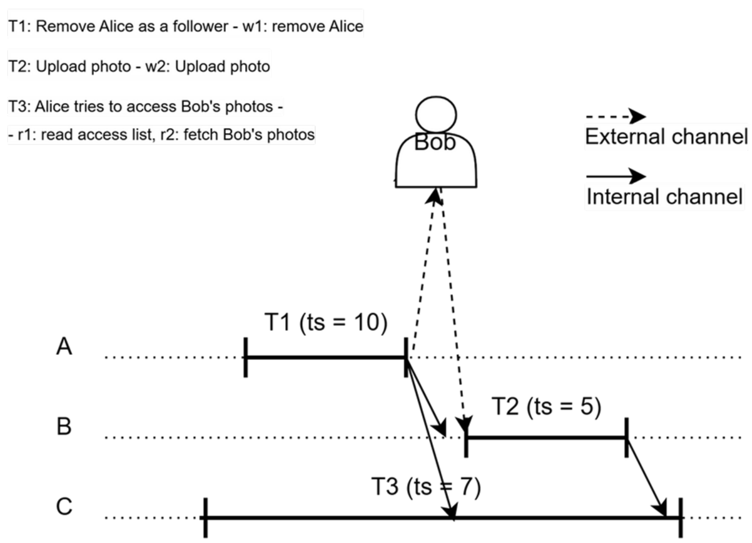

T1: Remove Alice as a follower - w1: remove Alice
T2: Upload photo - w2: Upload photo
T3: Alice tries to access Bob's photos -
- r1: read access list, r2: fetch Bob's photos
Bob
External channel
Internal channel
T1 (ts = 10)
T2 (ts = 5)
T3 (ts = 7)
A
B
C

Figure 1.1: T1 and T2 are indirectly related through the external event. i.e.. T1's response prompts Bob to make a request T2. If we can internalize this external event, node B can be informed of the completion of T1 on node A establishing real-time ordering.

1.3 Design

This paper presents IRON, a novel approach for verifying strict serializability in distributed database systems. We propose internalizing hypothetical external events between consecutive transactions each one executing on a different database server (node). Since there's no way for the system to know if an external event took place or not, we assume that such events invariably occur between any two consecutive transactions. We simulate these external events by internally passing messages between different nodes, which allows us to build a complete dependency graph. This methodology ensures that all identified inconsistencies are genuine violations of strict serializability, effectively eliminating the false positives typically associated with assumptions of clock skew in other serializability verification tools, and with a quick enough internal network (internal channel), we can eliminate false negatives to a large extent. Moreover, we wouldn't have to place any restrictions on the transaction type either enhancing IRON's applicability and flexibility.

The contributions of this paper include:

- We present IRON, a new mechanism to test distributed storage systems that can check for strict serializability violations without using time stamps and without placing any restrictions on transaction types.
- Proposing a set of solutions for implementation details.
- A comprehensive review of existing approaches to testing distributed database systems and relevant research on the same.

Chapter2

Background

Distributed databases are systems in which data is stored across multiple servers connected over a network. A distributed database provides better availability, scalability, and lower latency compared to traditional single-server databases. By distributing the data and the workload across multiple nodes, these systems can handle higher traffic loads and provide better performance.

Consistency models define how data is read and written across distributed systems. The goal is to ensure that all nodes have a consistent view of the data. There are several types of consistency models [3] [4]. Here are some popular ones arranged in increasing order of their strength:

- Eventual consistency: Ensures that, given enough time, all nodes will converge to the same value. This model allows for temporary inconsistencies but guarantees eventual agreement.

- Sequential Consistency: Guarantees that operations will appear in the same order across all nodes, although this order may not reflect the real-time order of the operations.

- Linearizability: A stronger form of consistency for non-transactional workloads. Ensures that all operations have an order consistent with their real-time order.

- Serializability: Ensures transaction isolation along with a total order of transactions across all nodes.
- Strict Serializability: The strongest consistency model, combining serializability with real-time ordering, ensuring that transactions appear to be executed in isolation and respecting their real-time order.

To achieve consistency, the various nodes of a distributed database need to work in sync with each other, ensuring that each node is aware of updates made by others. The stronger the consistency provided by the database, the more closely the nodes must coordinate to update each other about the transactions they execute. Strict serializability is the strictest consistency model, offering an isolated, total order of transactions across all nodes that is consistent with the real-time ordering of the transactions. This means that every node sees the transactions in the same order, and each transaction can see all transactions that were completed before it.

However, stronger consistency models result in lower throughput and are complex to implement. This complexity arises because transactions on different keys and different nodes may need to be ordered, which can be challenging due to each server having a different notion of time caused by clock skew. Specialized techniques like distributed optimistic concurrency control, transaction reordering, and distributed two-phase locking are needed to ensure strict serializability. These techniques are fairly complex, making it possible for their implementation to have bugs, leading to consistency violations that are equally complex to detect.

Chapter3

Design

3.1 Setup

We add a few additional components on top of the database to perform some key functions - An *internal channel*, *informers*, *loggers*.

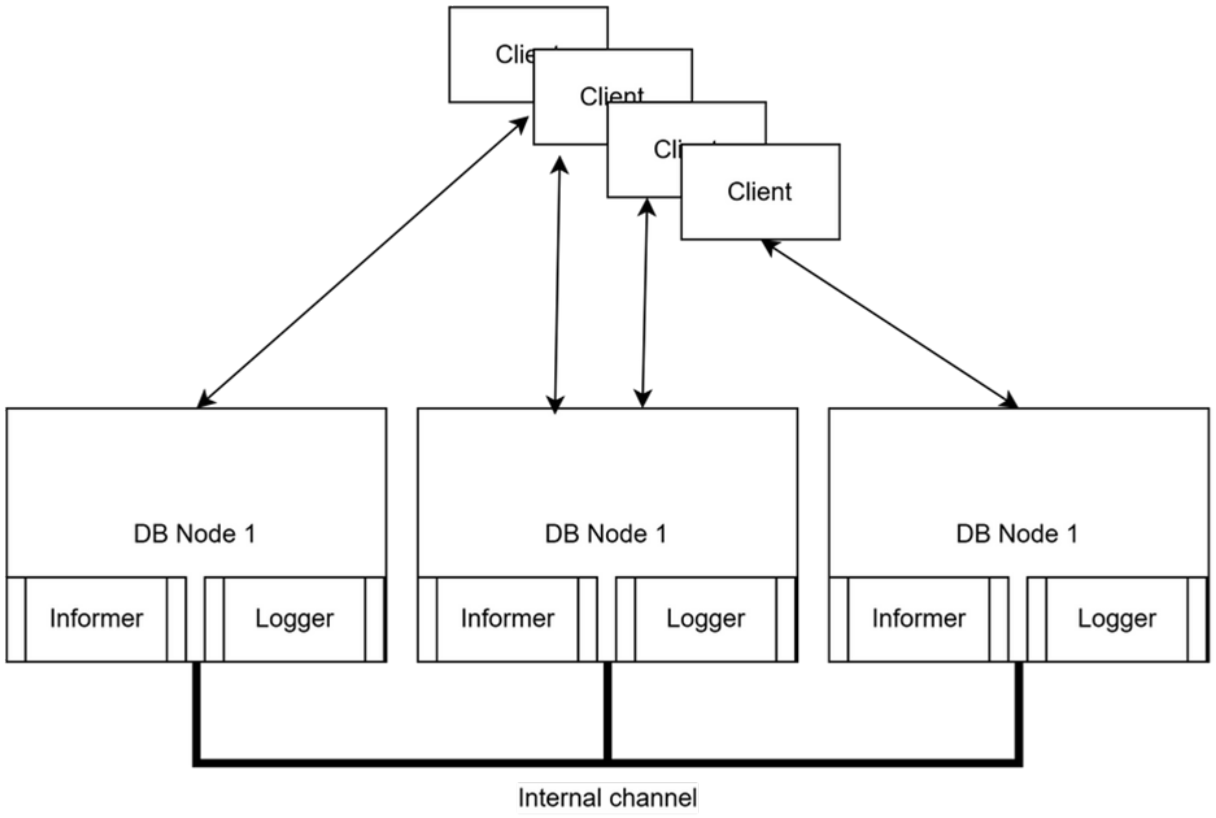


Figure 3.1: Iron's architecture. The verifier is off the critical path and hasn't been included in this image.

The *Internal channel* connects all the database nodes and will be used to internalize the possible external events. This could be a separate network for the database nodes to send internal messages to each other. This network has to be quicker than the underlying system network. This property of the internal channel directly affects the number of false negatives that we might see. For instance, if the internal channel is always slower than the underlying system network, the system will not flag any violations of Strict serializability, and the ideal case is when the internal channel is always quicker than the underlying network, in which case IRON will flag every violation of strict serializability (zero false negatives). While it may seem feasible for the underlying network of the database to utilize a faster network if such technology were readily available, the internal channel proposed in this study has distinct advantages that justify its separate implementation. These include: a) Smaller size of internal messages b) Special hardware like RDMA may be deployed temporarily [5] c) Fewer number of machines on this network d) Fewer messages on the internal channel

These factors combined provide a strategic edge, allowing the internal channel to operate faster than the standard network and more effectively verify strict serializability by minimizing false negatives.

Informers running on each node send out messages through the internal channel when a transaction completes on the host node. One-to-all notifications will be sent as soon as a transaction is committed, in parallel to sending out the responses to the client.

Loggers sit on every database node and collect the internal messages arriving at the host node and any new transaction requests from the clients. This information will be critical to complete the dependency graph during the verification step.

The operations of the informers and the loggers are quite simple and thus shouldn't result in a lot of overhead on their respective host nodes. We can always scale up the individual database nodes to accommodate these additional components if the need arises.

We'll also need the transaction order as seen by the database. For instance, FoundationDB adds versionstamp (the transaction's commit version) a unique, monotonically increasing value to each write operation at the commit time, This versionstamp or any alternative that orders the transactions can be leveraged to determine the sequence of the transactions as perceived by the database.

3.2 Verification

The *Verifier* will use the transaction history along with the logs collected by the loggers on each node to check for any strict serializability violations. We first build a graph with the transaction order as seen by the database - *base graph*. The base graph will contain execution edges and all the transactions so far on the database. For each transaction *ti* on the base graph, We'll check the logs on the DB node that received *ti* for the immediately prior internal message *ip*. Let the Transaction corresponding to the internal message *ip* be *tp*. Since we know that *ip* arrived before the request for *ti* we can infer that *tp* executed before *ti*. Thus, we draw an edge from *tp* to *ti*. This edge will represent the real-time ordering constraint between *tp* and *ti*. If this real-time edge inverts the existing execution edges (i.e. a cycle is formed in the dependency graph), then the transaction history is not strictly serializable.

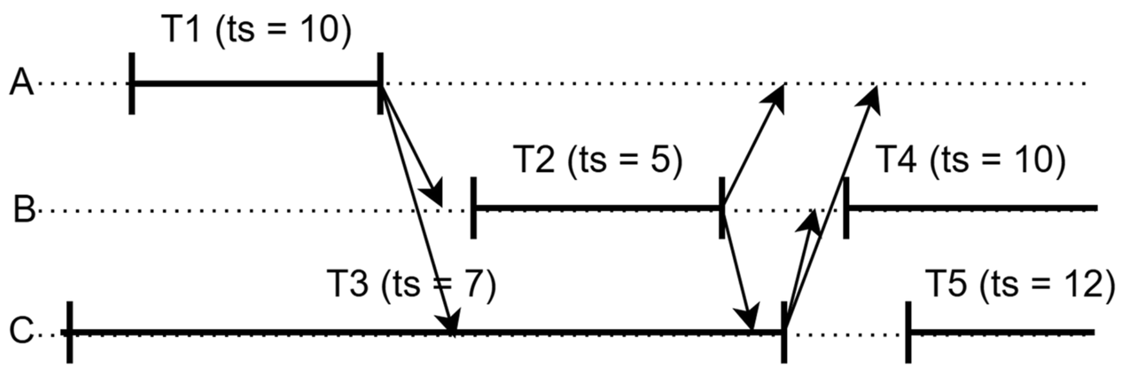


Figure 3.2: Sample timeline with 3 nodes and 5 transactions. The arrows indicate internal messages arriving through the internal channel.

Figure 3.2. shows a minimal example with 3 nodes, and 5 transactions. The base graph using the transaction ordering as seen by the database has been depicted in Fig 3.3 a.

The verifier then completes the graph by checking the logs at nodes that received requests for T2, T3, T1, T4, and T5 and in that order.

$$\begin{tikzcd}
\text{T2} \arrow[r] & \text{T3} \arrow[r] & \text{T1} \arrow[r] & \text{T4} \arrow[r] & \text{T5}
\end{tikzcd}$$

(a) Base graph. Based on the transaction order as seen by the database.

$$\begin{tikzcd}
\text{T2} \arrow[r] & \text{T3} \arrow[r] & \text{T1} \arrow[r] \arrow[ll, dashed, bend right] & \text{T4} \arrow[r] & \text{T5}
\end{tikzcd}$$

(b) Edge added from T1 to T2 when the verifier looks at the logs of node B and finds that an internal message informing about T1 arrived before the request for T2.

$$\begin{tikzcd}
\text{T2} \arrow[r] & \text{T3} \arrow[r] \arrow[rr, dashed, bend right] & \text{T1} \arrow[r] \arrow[ll, dashed, bend right] & \text{T4} \arrow[r] & \text{T5}
\end{tikzcd}$$

(c) Edge added from T3 to T4 when the verifier looks at the logs of node B and finds that an internal message informing about T3 arrived before the request for T4.

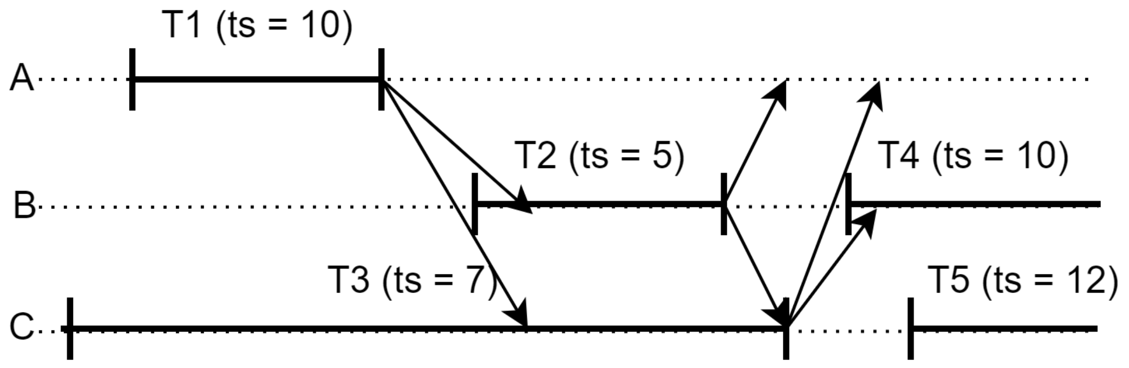
$$\begin{tikzcd}
\text{T2} \arrow[r] \arrow[rrrr, dashed, bend right] & \text{T3} \arrow[r] \arrow[rr, dashed, bend right] & \text{T1} \arrow[r] \arrow[ll, dashed, bend right] & \text{T4} \arrow[r] & \text{T5}
\end{tikzcd}$$

(d) Edge added from T2 to T5 when the verifier looks at the logs of node C and finds that an internal message informing about T2 arrived before the request for T5.

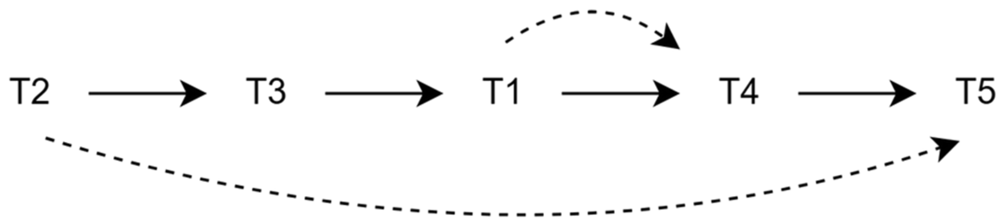
Figure 3.3: Verifier incrementally adding edges to the dependency graph for the transaction depicted in Fig 3.2.

If the internal messages arrive late at other nodes, i.e. the internal channel is slower than the external channel then false positives will occur. However, this delay doesn't cause any real-time order edges that invert the base graph, thus, it will not result in any false positives.

The verifier will be off the critical path and not cause overheads to the underlying system.



(a) False negatives when the internal channel is slower than the external channel. The internal message between T1 and T2 is late which prevents IRON from detecting a real-time ordering violation.



(b) Dependency graph for the transaction history depicted in fig 3.5 (a).

Figure 3.4: An example showing possible false negatives when the internal message is late.

Chapter4

Related work

Many past works use dependency graphs to check for real-time ordering violations. However, this is the first work that proposes using an internal channel to add missing information in the dependency graphs which can internalize external events and thus establish real-time order between transactions. In this section, we will briefly discuss related research and their impact on our paper.

An important strict serializability violation we based our design around comes from the timestamp inversal pitfall highlighted in the NCC paper[6]. This paper also introduces a novel approach to enforce strict serializability across distributed databases, which reduces unnecessary overhead by guaranteeing one-round latency, lock-free, and non-blocking execution. Understanding the intricacies of a strictly serializable database from this work highlights the ease with which bugs can manifest, this further shows the need for a robust testing mechanism for such systems.

COBRA [1] is a closely related work. This work introduces a mechanism for black-box checking of serializability and strict serializability. The system's architecture includes history collectors that record every request and response, alongside the version of the key being accessed or modified. A verifier then takes this recorded data and constructs a graph, encoding the transaction history to analyze its properties. In this graph, each transaction is represented as a node, and dependencies between them are depicted as edges. Additionally, the graph incorporates constraints representing pairs of potential indirect dependencies, but only one in each pair can hold true. Once the transaction history is available, COBRA proposes reducing the number of constraints to bring down the complexity of the problem.

This is achieved by 3 techniques: a) Combining writes b) Coalescing constraints and c) pruning constraints. MonoSAT SMT solver encodes this pruned transaction history to create a graph. If any of the resulting graphs is compatible with the known dependencies and the constraints and is acyclic then the verifier confirms the transaction history to be serializable.

COBRA uses timestamps associated with each transaction to verify strict serializability. It uses the dependency graph with one additional component - real-time edges. It uses timestamps to determine if one transaction occurred before another and then draws a real-time edge between the two transactions. To accommodate for the clock drift, COBRA uses a clock drift threshold of 100 ms which is added to the original commit timestamps. Doing so will lead to false negatives and as this threshold can trick the verifier into thinking that two consecutive transactions are concurrent. If the added clock drift threshold is smaller than the actual clock skew, it can also lead to false positives where the verifier thinks two concurrent transactions are sequential and are incorrectly inverted. This flaw motivated us to find an alternative method to verify strict serializability. However, other techniques used in this work such as issuing fence transactions, and garbage collection to improve scalability is indeed impressive and could be used in our design too.

Existential consistency [2] analyzes the balance between enhanced consistency and performance in large-scale services like Facebook. This work studies the potential reduction in anomalies when shifting to stronger consistency models. This work assesses millions of requests by collecting operational traces from Facebook's TAO and constructing dependency graphs. They propose adding 35 milliseconds to the commit timestamp and subtracting 35 milliseconds from the start timestamps to account for clock skew while verifying linearizability given the collected transaction history.

While searching for tools to test distributed databases, we came across Jepsen [7]. Jepsen is a prominent Clojure library for testing distributed systems. Using Jepsen we can set up a distributed database, use built-in tests or write our own tests to run operations on them and then verify if these operations were executed correctly. Jepsen also allows us to create a Nemesis process that imparts network partitions, disk errors, etc. to test systems under stress.

Two Jepsen tests [8] are relevant to our work: a) Comments test: checks for a specific type of strict serializability violation, where transactions on disjoint records are visible out of order. Jepsen issues write operations in a specific sequence and at random points does a read to ensure that the transactions are ordered as they were executed. b) Bank test: Designed to verify snapshot isolation. Simulates a set of bank accounts, one per row, and transfers money between them at random, ensuring that no account goes negative at any point. Jepsen also has a separate checker for linearizability called Knossos, however, it is a best-effort tool and hasn't been successful in detecting any real-time ordering violations thus far.

Chapter5

Future work

Future research will explore networking techniques for optimizing the internal channel's performance and implementing and evaluating IRON for diverse database architectures. This work lays a foundational step toward timestamp-independent evaluation of real-time ordering of transactions in distributed systems.

Chapter6

Conclusion

We discussed consistency models, specifically strict serializability, existing methods, and research related to verifying strict serializability, the pitfalls of using timestamps to infer real-time ordering constraints between transactions in dependency graphs and thus the need for this work.

This thesis introduced IRON, a novel approach designed to enhance the verification of strict serializability in distributed databases without relying on timestamps. Our methodology proposes internalizing hypothetical external events between transactions to construct a comprehensive dependency graph, ensuring a robust verification process and eliminating false positives.

Bibliography

[1] Shuai Mu Michael Walfish Cheng Tan Changgeng Zhao. "Cobra: Making Transactional Key-Value Stores Verifiably Serializable". In: *OSDI* 2 (2020).

[2] Philippe Ajoux Jim Hunt Yee Jiun Song Wendy Tobagus Sanjeev Kumar Wyatt Lloyd Haonan Lu Kaushik Veeraraghavan. "Existential Consistency: Measuring and Understanding Consistency at Facebook". In: *SOSP* 3 (2015).

[3] *Consistency models*. URL: `https://jepsen.io/consistency`.

[4] *CockroachDB's consistency model*. URL: `https://www.cockroachlabs.com/blog/consistency-model/`.

[5] Zhong Deng Gaurav Soni Jianxi Ye Jitu Padhye Marina Lipshteyn Chuanxiong Guo Haitao Wu. "RDMA over Commodity Ethernet at Scale". In: *SIGCOMM* 4 (2016).

[6] Siddhartha Sen Wyatt Lloyd Haonan Lu Shuai Mu. "NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall". In: *OSDI* 1 (2023).

[7] *Jepsen*. URL: `https://jepsen.io/`.

[8] *Jepsen tests on cockroachDB*. URL: `https://www.cockroachlabs.com/blog/jepsen-tests-lessons/`.