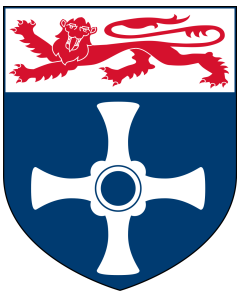
High Performance Concurrency Control and Commit Protocols in OLTP Databases



Jack Waudby

School of Computing

Newcastle University

This dissertation is submitted for the degree of

Doctor of Philosophy

December 2023

For Snowy & Jessie,

who possessed Beauty without Vanity,

Strength without Insolence,

Courage without Ferocity,

and all the virtues of Man without his Vices.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Jack Waudby

December 2023

Acknowledgements

Over the course of my PhD, I was fortunate to have received support from a number of colleagues, collaborators, and friends.

Advisors I would like to start by thanking my supervisor, Dr Paul Ezhilchelvan, for his guidance and patience. Thank you for taking a chance on a student with little Computer Science background, it has been a pleasure to work with you over these past years. I would also think to thank Dr Jim Webber for his advice and continued support across my PhD, I have always taken so much from our discussions. I'll be forever grateful to you both for introducing me to databases and distributed systems.

Colleagues My thanks to the staff of the Newcastle University Centre for Doctoral Training (CDT) in Cloud Computing for Big Data: to Professors Paul Watson and Darren Wilkinson for leading the CDT; thanks especially to Jen Wood and Andrew Turnbull, who literally keep the whole show on the road. Thanks to my fellow CDT PhD students for sharing their knowledge, experience and friendship. In particular thanks go to the members of Cohort 4, Georgia Atkinson, Julian Austin, Matthew Fisher, Rob Geada, Konstantinos (Sam) Georgopoulos, Carlos Vladimiro González Zelaya, Benjamin Lam, Cameron Trotter, and Joshua Barney it has been a pleasure to share this journey with you all, I wish you all the best whatever you do next. Thanks to the other members of the (unofficial) Ez Labs, Thomas Cooper, George Stamatiadis, and Chris Johnson.

Neo4j Clustering Team During my internship at Neo4j I was fortunate to work with many outstanding people. Thanks to Hugo Firth, Aleksey Karasavov, Antony Butterfield, Balázs Lendvai, Tselmeg Baasan, Ragnar Wernersson, Aurélien Arena, Fábio Botelho, and Marie Gaillard.

Research visits and collaborations I would like to acknowledge the support of institutions which hosted my research visits: the Budapest University of Technology and Economics and Centrum Wiskunde & Informatica. I would like to express my gratitude to Gábor Szárnyas and Ben Steer, with whom I worked closely through the Linked Data Benchmark Council's Social Network Benchmark task force. Ben, thank you for sharing your knowledge, experience and friendship with me, I wish you all the best in your future endeavors. Gábor you are a force of nature, I can not express in words how much you have helped me across my PhD, I will be forever grateful. Thanks to Professor Isi Mitrani, with whom I got the opportunity to work on several projects.

Friends and Family Thank you to all my friends for providing an invaluable support network. Also, thank you to the latest addition to the family, Freddie, for bringing a smile back to my face. Finally, thank you to my father, Stuart, to my mother, Heather, and my sister Holly. Not just for the past four years, but for the 24 years before that as well. You have given me endless support, opportunity, and encouragement over the years. Through the highs and lows your support has never wavered. This achievement is yours as much as it is mine. Thank you.

A man with new ideas is a madman, until his ideas triumph.

Marcelo Bielsa

Abstract

Depending on the required scale, modern data-orientated applications are built on top of either many-core or distributed OLTP databases. In both architectures, database concurrency control is a performance critical component. Additionally, in distributed OLTP databases, where a transaction can span across multiple data partitions, distributed atomic commitment is a necessity; unless the latter is designed judiciously, it can significantly degrade performance.

This thesis explores both facets to address the inefficiencies and limitations of existing work. The *wait-hit protocol* is a serializable concurrency control protocol that is designed to offer high performance across all scale points. This thesis also investigates *mixed serialization graph testing* by applying the recently revived graph-based approach to concurrency control, which minimizes unnecessary aborts, to cater for transactions running at weaker isolation levels, a phenomenon common in practice. Protocols are developed for ensuring *reciprocal consistency* and *edge-order consistency* in distributed graph databases, guarantees unique to graph databases which without sufficient concurrency control can be violated causing irreparable data corruption. Finally, we demonstrate that epoch-based distributed databases can amortize atomic commitment costs. In addition to developing an analytical model for choosing the optimal epoch size in such databases, this thesis presents *epoch-based multi-commit* which can reduce wasted work when database nodes fail.

Each protocol has been subjected to extensive performance evaluation either through simulations or implementation. Our experiments show that each protocol offers a marked improvement over the current state-of-the-art.

Table of contents

List of figures

List of tables

List of Algorithms

Nomenclature

Acronyms / Abbreviations

ACID Atomicity, Consistency, Isolation, Durability

AIMD Additive Increase/Multiplicative Decrease

BASE Basically Available Soft State

BCC Balanced Concurrency Control

CG Conflict Graph

DBMS Database Management System

DFS Depth First Search

DSG Direct Serialization Graph

FIFO First in first out

GDBMS Graph Database Management System

HAT Highly Available Transaction

LDBC Linked Data Benchmark Council

LSQB Labelled Subgraph Query Benchmark

$MC-WHP$ Many Core Wait Hit Protocol

$MOCC$ Mostly Optimistic Concurrency Control

MPT Multi-Partition Transaction

$MSGT$ Mixed Serialization Graph Testing

$MTTR$ Mean time to repair

$MVCC$ Multi-Version Concurrency Control

OCC Optimistic Concurrency Control

$TPC-C$ Transaction Processing Performance Council Benchmark C

$OLTP$ Online Transaction Processing

PPS Product-Parts-Supplier

$RDBMS$ Relational Database Management System

SGT Serialization Graph Testing

SNB Social Network Benchmark

BI Business Intelligance

$S2PL$ Strict Two-Phase Locking

$TATP$ Telecommunication Application Transaction Processing Benchmark

TL Terminated List

TO Timestamp Ordering

$2PC$ Two-Phase Commit

$2PL$ Two-Phase Locking

TPS Transactions per second

WAN Wide Area Networks

WHP Wait-Hit Protocol

$YCSB$ Yahoo Cloud Serving Benchmark

Chapter 1

Introduction

Modern data-orientated applications are built on top of online transaction processing (OLTP) database management systems (DBMSs). These DBMSs are either *many-core*, single-node systems with 10s or 100s of cores, or *distributed*, multi-node systems each storing a disjoint portion of the database. In both types of database, concurrency control is a performance critical component. Concurrency control approaches are typically classified as pessimistic or optimistic [12]. The performance of each approach differs based on workload characteristics and database type. Many-core databases opt for optimistic approaches [58, 110, 122], whereas it has been demonstrated that for distributed databases pessimistic approaches generally perform better [54]. Thus, transitioning from a many-core to distributed deployment often requires re-architecting the concurrency control strategy to avoid poor performance, which is not compatible with the modern cloud environment and user expectations. Contemporary cloud providers offer users a wide range of hardware options, enabling them to scale easily from a single-core machine to a many-core machine, to deployments spread across data centers and the globe. Naturally, when users deploy DBMSs in the cloud they expect performance to scale seamlessly as hardware resources are scaled, without needing to consider the performance implications imposed by the underlying concurrency control strategy.

In OLTP DBMSs, another aspect that significantly impacts performance is the isolation guarantee provided. The gold standard isolation level is Serializability, which guarantees that concurrent transactions appear to have been isolated from each other, sequentially executing over the database which, assuming transactions are individually correct, guarantees a correct DBMS state [12]. In practice, efficient implementation of serializable transaction processing is challenging and its performance often unpalatable for applications' requirements [85, 99]. Thus, DBMSs allow transactions to be executed at different, weaker isolation levels, e.g., Read Committed [1, 51]. Weaker isolation levels increase the number of permissible executions compared to Serializability, which allows for more concurrency, thus higher

throughput and lower latency. But, this comes at the cost of potential non-serializable, anomalous behaviour. Despite the ubiquitous usage of weak isolation in practice, the majority of research has focused on making the processing of transactions with Serializable isolation performant [85], leaving the efficient support of processing transactions with weaker isolation guarantees underserved.

The quest for high DBMS performance at scale, combined with the misguided notion that transactions are inherently non-scalable, fueled the development of numerous distributed NoSQL databases, e.g., DynamoDB [26]. These databases forgo cross-partition transactional semantics, at best offering transactions within a single partition. This is suitable for databases with a key-value or document data model, but problematic for one class of database: *graph databases* (GDBMS) [92]. Graph databases use the labelled property graph data model in which data is modeled as nodes and edges, which additionally can be labelled and have properties. Unfortunately, system designers have attempted to build graph databases by adding a graph layer on top of NoSQL databases, e.g., JanusGraph [61]. The lack of cross-partition concurrency control can violate graph integrity and cause irreparable data corruption [41].

In distributed DBMSs that do provide cross-partition, distributed transactions, getting all partitions to agree on the outcome of a transaction requires an atomic commitment protocol (normally, *two-phase commit* (2PC) [12]), which is a well-established bottleneck that can significantly degrade performance. Numerous techniques have tried to avoid or minimize distributed transactions [23, 24], or avoid 2PC all together [71, 104]. One promising approach is *epoch-based commit*, which proposes that 2PC be executed only once for all transactions processed within a time interval called an *epoch*, rather than executed per-transaction [72]. However, determining the right epoch size for a given workload and cluster configuration remains a challenge and is key to achieving the desired throughput and latency. Additionally, a unfortunate drawback with epoch-based commit is if a database node fails then all transactions

within the group must be rolled back and re-executed, potentially resulting in a significant degree of wasted work.

1.1 Research Challenges

In this section, given the outstanding issues introduced above we now describe in more depth the research challenges in OLTP database concurrency control and atomic commitment addressed in this thesis.

A Concurrency Control Protocol for Any Scale

Modern cloud providers enable users to scale easily from a many-core machine, to a deployment spread across data centers and the globe through the provision of a wide array of hardware options. Naturally, when users deploy DBMSs in the cloud they expect the performance of a DBMS to scale as hardware resources are scaled. A critical component in realizing high performance in a DBMS is concurrency control. Unfortunately, many concurrency control protocols are not designed to scale well; a protocol architected for one scale point inevitably needs to be re-adapted for another. This thesis addresses the challenge of designing a high-performance concurrency control protocol that can effectively scale from a single core, to multicore, to the globe whilst offering good performance at each scale point, without having to relax from the gold standard isolation level, Serializability.

High Performance Weak Isolation

Weak isolation is common in practice, with practical DBMSs in fact being *mixed* systems, supporting a range of weaker isolation levels. Yet research has primarily focused on improving serialization transaction processing performance. A recent development in serializable transaction processing has been the revival of graph-based concurrency control, which was

historically deemed nonviable due to concerns over the computational costs of maintaining an acyclic conflict graph. Graph-based concurrency control has the theoretically optimal property of accepting all and only valid schedules and has been demonstrated to offer comparable, and often higher, performance when compared to traditional and contemporary concurrency control protocols in a many-core database. This thesis explores the question, how can graph-based concurrency control be extended to a mixed DBMS supporting transactions executed at weak isolation levels, whilst accepting all and only valid executions? Such an approach would permit higher concurrency and performance.

Preserving Edge Consistency

Several contemporary distributed graph databases (e.g., JanusGraph) use existing NoSQL databases (e.g., Apache Cassandra) as a storage backend, adapting them with an API in order to handle a graph data model. This approach is attractive as the underlying store offers high scalability. However, it only offers weak isolation guarantees across database nodes which has serious ramifications for the integrity of graph database systems and can lead to irreversible database corruption. This thesis will address the challenge of developing a suite of lightweight concurrency control protocols tailored for graph database workloads, which maintain graph integrity and offer suitable transactional throughput and latency.

Optimizing Epoch-Based Commit

Achieving good performance in an epoch-based distributed database requires the database operator to select the right epoch size. This thesis develops two analytical models to estimate throughput and average latency in terms of epoch size taking into account load and failure conditions. Additionally, this thesis addresses the challenge of developing an epoch-based commit protocol that avoids rolling back all transactions within an epoch in the event of a

single node failure. Reducing wasted work decreases system load resulting from retries, and improves performance and user experience.

1.2 Contributions

The work presented in this thesis addresses the research challenges in Section 1.1 and makes the following contributions:

(i) Development and evaluation of the *wait-hit protocol* (WHP), a general purpose concurrency control protocol for any scale. It can effectively scale vertically, as the core count is increased on a given machine and horizontally, as the number of machines is increased in a cluster, whilst providing Serializable isolation.

(ii) Design and evaluation of *mixed serialization graph testing* (MSGT), a concurrency control protocol for mixed DBMSs. MSGT extends graph-based concurrency control to mixed environments using Adya's *mixing-correct theorem* [1]. It preserves the desirable property of minimizing the number of aborted schedules, whilst providing database operators with a selection of isolation levels.

(iii) Three concurrency control protocols that guarantee the preservation of graph integrity in a distributed graph database are introduced and evaluated. Specifically, these protocols ensure *reciprocal consistency* [41] and *edge-order consistency*, guarantees unique to graph databases, are maintained.

(iv) Development of two analytical models of epoch-based commit to aid database administrators in selecting the epoch size that offers the desired trade-off between throughput and latency.

(v) Proposal of *epoch-based multi-commit*, which aims to minimize number of aborted transactions when failures occur, but also performs identically to epoch-based commit when failures do not occur.

1.3 Thesis Structure

This section provides the thesis structure, with an abridged chapter-by-chapter summary. The thesis structure is summarized in Figure 1.1.

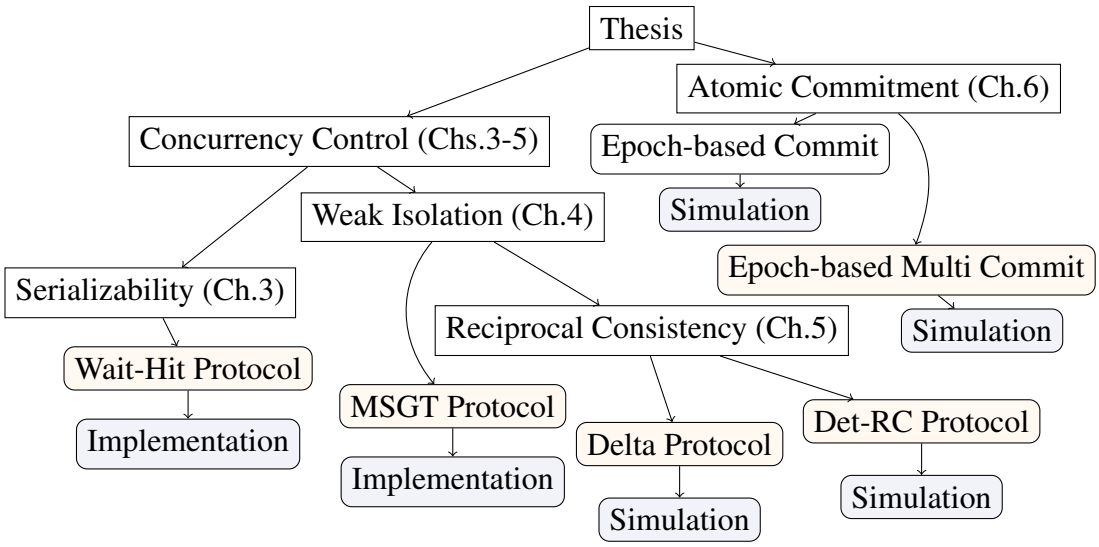


Fig. 1.1 Thesis Structure. Topics are given in boxes with square corners. Algorithmic contributions are illustrated with rounded boxes and shaded in orange; the performance evaluation technique used is shaded in blue.

Chapter 1 presents the motivations behind the work carried out as part of this thesis, highlights the main contributions of the research, and describes the related peer-reviewed publications produced across the course of the PhD.

Chapter 2 describes the necessary technical background material closely linked with the work performed in the chapters of this thesis.

Chapter 3 develops the wait-hit protocol, an optimistic multi-versioned concurrency control protocol that scales vertically, with the core count, and horizontally, with the cluster size. We compare the protocol's performance against classical and state-of-the-art protocols on metrics such as throughput, latency, and proportion of aborted transactions.

Chapter 4 investigates the widespread availability of weak isolation in commercial and open source database systems, and proposes mixed serialization graph testing. A concurrency control protocol that gives high performance on servers with many-cores minimizes the number of unnecessary aborts, and crucially permits concurrent transactions to be executed at a range of isolation levels. We evaluate the protocol's performance using several popular transaction processing benchmarks that have been augmented to generate workloads containing transactions declared with different isolation requirements.

Chapter 5 outlines the design of two protocols that preserve reciprocal consistency and one that preserves edge-order consistency in a distributed graph database. Approximate models are developed for each protocol to allow for a comprehensive performance evaluation. To the best of our knowledge this is the first attempt to develop concurrency control protocols specifically catered for graph databases.

Chapter 6 develops two analytical models of epoch-based commit. These allow for the choosing of an epoch size that maximizes throughput, minimizes average response time, or seeks a trade-off between them. Additionally, epoch-based multi-commit is presented which aims to minimize transaction aborts in the event of node failures. We execute a performance study of the protocols.

Chapter 7 summarizes the conclusions of each chapter in this thesis and outlines the areas for future work.

1.4 Publications

Across the course of my PhD I have contributed to nine peer-reviewed publications, five of which have directly contributed to this thesis. Contributing publications are listed below in

reverse chronological order, with the relevant thesis chapter indicated. A short description is provided for each publication.

Waudby, J., Ezhilchelvan, P., Mitrani, I. and Webber, J., 2022. A Performance Study of Epoch-based Commit Protocols in Distributed OLTP Databases. To appear: In 41st International Symposium on Reliable Distributed Systems, SRDS. **[Chapter 6]**

Description: distributed OLTP systems execute the high-overhead, 2PC protocol at the end of every distributed transaction. *Epoch-based commit* proposes that 2PC be executed only once for all transactions processed within a time interval called an *epoch*. Increasing epoch duration allows more transactions to be processed before the common 2PC. It thus reduces 2PC overhead per transaction, increases throughput but also increases average transaction latency. Therefore, the ability to choose the right epoch size that offers the desired trade-off between throughput and latency is required. To this end, we develop two analytical models to estimate throughput and average latency in terms of epoch size taking into account load and failure conditions. We then present *epoch-based multi-commit* which, unlike epoch-based commit, seeks to avoid all transactions being aborted when failures occur, and also performs identically when failures do not occur. Our performance study identifies workload factors that make it more effective in preventing transaction aborts and concludes that the analytical models can be equally useful in predicting its performance as well.

Waudby, J., Ezhilchelvan, P., and Webber, J., 2022. Pick & Mix Isolation Levels: Mixed Serialization Graph Testing. To appear: In *Proceedings of the 14th TPC Technology Conference on Performance Evaluation & Benchmarking*. **[Chapter 4]**

Description: *Serialization graph testing* (SGT) faithfully implements the *conflict graph theorem* by aborting only those transactions that would actually violate serializability (introduce a cycle), thus maintaining the required acyclic invariant. Historically, SGT was deemed unviable in practice due to the high computational costs of maintaining an acyclic graph.

Research has however overturned this historical view by utilizing the increased computational power available due to modern hardware. Furthermore, a survey of 24 databases suggests that not all transactions demand conflict serializability but different transactions can perfectly settle for different, weaker isolation levels which typically require relatively lower overheads. Thus, in such a mixed environment, providing only the isolation level required of each transaction should, in theory, increase throughput and reduce aborts. This paper extends SGT for mixed environments subject to Adya's mixing-correct theorem and demonstrates the resulting performance improvement. We augment the Yahoo! Cloud Serving Benchmark (YCSB) benchmark to generate transactions with different isolation requirements. Mixed serialization graph testing can achieve up to a 28% increase in throughput and a 19% decrease in aborts over SGT.

[115] Waudby, J., 2022. High Performance Mixed Graph-Based Concurrency Control. In *Proceedings of the VLDB 2022 PhD Workshop co-located with the 48th International Conference on Very Large Databases.* **[Chapter 4]**

Description: this is an abridged version of "Pick & Mix Isolation Levels: Mixed Serialization Graph Testing".

[116] Waudby, J., Ezhilchelvan, P., Webber, J. and Mitrani, I., 2020. Preserving reciprocal consistency in distributed graph databases. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (pp. 1-7). **[Chapter 5]**

Description: reciprocal consistency is an important property that must be preserved in distributed graph databases, failure to do so seriously undermines the integrity of the database itself in the long term. Reciprocal consistency can be maintained as a part of enforcing any known isolation guarantee and such an enforcement is also known to lead to reduction in performance. Therefore, in practice, distributed graph databases are often built atop *basically available soft state* (BASE) databases with no isolation guarantees, benefiting from

good performance but leaving them susceptible to corruption due to violations of reciprocal consistency. This paper designs and presents a lightweight, locking-free protocol and then evaluates the protocol's abilities to preserve reciprocal consistency and also offer good throughput. Our evaluations establish that the protocol can offer both integrity guarantees and sound performance when the value of its parameter is chosen appropriately.

[40] Ezhilchelvan, P., Mitrani, I., **Waudby, J.** and Webber, J., 2019, November. Design and Evaluation of an Edge Concurrency Control Protocol for Distributed Graph Databases. In *European Workshop on Performance Engineering* (pp. 50-64). Springer, Cham. **[Chapter 5]**

Description: a new concurrency control protocol for distributed graph databases is described. It avoids the introduction of certain types of inconsistencies by aborting vulnerable transactions. An approximate model that allows the computation of performance measures, including the fraction of aborted transactions, is developed. The accuracy of the approximations is assessed by comparing them with simulations, for a variety of parameter settings.

Other Publications

The remaining four peer-reviewed publications result from collaboration with the *Linked Data Benchmark Council* (LDBC) *Social Network Benchmark* (SNB) Benchmarking Task Force of which I am a member. These are not included in the thesis, but are demonstrations of my research skills.

Szarnyas, G.,**Waudby, J.**, Steer, B., Szakallas, D., Birler, A., Wu, M., Zhang, Y., and Boncz, P. 2023. The LDBC Social Network Benchmark: Business Intelligence workload. To appear: In *Proceedings of the VLDB Endow. 2023*.

Description: the LDBC SNB's *Business Intelligence* (BI) workload is a comprehensive graph OLAP benchmark targeting analytical data systems capable of supporting graph workloads.

This paper presents SNB BI experiments on both a relational and a native graph database system. SNB BI advances the state-of-the art in synthetic and scalable analytical database benchmarks in many aspects.

[118] Waudby, J., Steer, B., Prat-Perez, A., and Szarnyas, G., I., 2020. Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (pp. 1-8).

Description: this work extends the LDBC SNB data generator by introducing lifespan attributes for the creation and deletion dates of its graph entities to allow for the generation of a temporal dynamic graph. This allows for the definition of complex deletions in LDBC SNB benchmarks which further challenges the performance of graph processing systems.

[117] Waudby, J., Steer, B., Karimov, K., Marton, J., Boncz, B., and Szarnyas, G., I., 2020. Towards Testing ACID Compliance in the LDBC Social Network Benchmark. In *Proceedings of the 12th TPC Technology Conference on Performance Evaluation & Benchmarking*.

Description: verifying ACID compliance is an essential part of database benchmarking, because the integrity of performance results can be undermined as the performance benefits of operating with weaker safety guarantees (at the potential cost of correctness) are well known. Traditionally, benchmarks have specified a number of tests to validate ACID compliance. However, these tests have been formulated in the context of relational database systems and SQL, whereas our context is systems for graph data, many of which are non-relational. This paper presents a set of data model-agnostic ACID compliance tests for the LDBC SNB suite's *Interactive* workload, a transaction processing benchmark for graph databases. We test all

ACID properties with a particular emphasis on isolation, covering 10 transaction anomalies in total. We present results from implementing the test suite on five database systems.

[77] Mhedhbi, A., Lissandrini, M., Kuiper, L., **Waudby, J.**, and Szarnyas, G., I., 2021. LSQB: A Large-Scale Subgraph Query Benchmark. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (pp. 1-11).

Description: this paper introduces Labelled Subgraph Query Benchmark (LSQB), a new large-scale subgraph query benchmark. LSQB tests the performance of database management systems on an important class of subgraph queries overlooked by existing benchmarks. LSQB contains a total of nine queries and leverages the LDBC social network data generator for scalability. The benchmark gained both academic and industrial interest and is used internally by 5+ different vendors.

Chapter 2

Background

Summary

This chapter provides core background material underpinning the work conducted in this thesis; later chapters introduce additional material as and when neccessary. Section 2.1 outlines database concurrency control, specifically, the concepts of Serializability and weak isolation, along with various concurrency control strategies and how these have been applied in many-core and distributed DBMSs. In Section 2.1, we also discuss recent advances in mitigating against the overhead of 2PC. Section 2.2 provides an overview of graph processing, focusing on graph databases. It introduces the notions of Reciprocal Consistency and Edge-Order Consistency outlining how they can be violated given insufficent concurrency control in a common distributed graph database architecture. Lastly, in Section 2.3 we describe the evaluation techniques used in the thesis and the evaluation frameworks used at various points throughout the thesis to analyze protocol performance. Where appropriate we state which chapter the background material is relevant to.

2.1 Database Concurrency Control

Definition 1 (OLTP DBMSs) *Online transaction processing (OLTP) DBMSs respond to and concurrently process a large number of relatively simple database transactions in real-time.*

Concurrency control is an integral component in an OLTP DBMS. Even if transactions are individually correct and there are no system failures, the operations of concurrently executing transactions can interleave in a manner such that the database state is left inconsistent. Concurrency control is responsible for ensuring that the effects of concurrently executing transactions are logically isolated from each other, providing each with the illusion of running alone in the DBMS.

Definition 2 (Concurrency Control) *A concurrency control mechanism is used in order for OLTP DBMSs to maintain data consistency when executing multiple database transactions at the same time.*

2.1.1 Serializability *(Relevant to Chapter 3)*

Definition 3 (Database Isolation) *Database isolation is a database's ability to allow a transaction to execute as if it is alone in the system, even though there may be a large number of concurrently running transactions. The degree of isolation is determined by the concurrency control mechanism. The strongest isolation level is Serializability.*

The DBMS component responsible for providing concurrency control is the *scheduler*, which enforces a correctness criterion called Serializability. An execution of transactions is serializable if it produces a database state equivalent to some serial execution of the same set of transactions; assuming transactions are individually correct, a serial execution trivially ensures a consistent database state [12].

In practice, schedulers enforce a stronger condition called Conflict Serializability, which is sufficient to ensure an execution is serializable. An execution of transactions in a DBMS can be represented by a *schedule*, a time ordered sequence of their operations. For example, consider transactions T_1, T_2, and T_3 shown in schedule s below; a write on item x by transaction T_i is denoted by $w_i[x]$, a read by $r_i[x]$, and a commit operation by c_i.

$$s = w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]\ w_2[z]\ w_3[z]\ r_3[x]\ c_1\ c_3\ c_2$$

As shown in Figure 2.1, the schedule s can be represented by a *conflict graph* $CG(s)$. Each transaction is represented by a node in the graph. A *conflict* exists between two transactions if they both operate on the same data item and at least one operation is a write, thus, changing the order of these operations *could* alter the behaviour of at least one of the transactions.

In a conflict graph, conflicting operations a_i of T_i and b_j of T_j such that $a_i[x] < b_j[x]$, where $T_i \neq T_j$, are represented by an edge $T_i \rightarrow T_j$ in the graph; possible conflict pairs are $(a,b) \in [(r,w),(w,r),(w,w)]$. For example, in s, T_2 reads x after T_1 writes to x, thus there exists an edge from T_1 to T_2 in Figure 2.1. An execution of transactions and its corresponding schedule is conflict serializable if a serial ordering of transactions that satisfies all conflict edges can be found. Such a serial ordering exists iff the conflict graph is acyclic. This is known as the *conflict graph theorem* [12]. Note, s is not conflict serializable because $CG(s)$ in Figure 2.1 contains a cycle.

Theorem 1 (Conflict Graph Theorem) *A schedule s is conflict serializable iff its corresponding conflict graph $CG(s)$ is acyclic.*

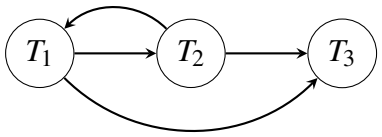


Fig. 2.1 Conflict graph representation of s.

2.1.2 Weak Isolation *(Relevant to Chapter 4)*

Definition 4 (Weak Isolation) *Weak isolation refers to the set of databases isolation levels* weaker *than Serializability. Here weaker means the database increases the number of allowable executions.*

Weak isolation refers to the set of isolation guarantees that increase the number of allowable executions compared to Serializability. Permitting more executions can increase performance, at the cost of possible non-serializable behaviour. Weak isolation levels were first introduced in [51], which describes four *"degrees of consistency"* (Degree 0-3) that provide transactions with increasing levels of protection from concurrent transactions. However, these

definitions are only applicable to single-versioned systems using lock-based concurrency control (see Subsection 2.1.3). The ANSI/ISO SQL-92 [76] specification aimed to define an implementation-independent standard for weak isolation, supporting lock, validation, and timestamp-based concurrency control mechanisms (see Subsection 2.1.3). The isolation levels proposed in the ANSI/ISO SQL-92 specification are written down informally in terms of *anomalies*[1] they prevent: Read Uncommitted prevents no anomalies, Read Committed disallows Dirty Reads, Repeatable Read disallows both Dirty Reads and Fuzzy Reads, while Serializable additionally disallows Phantoms.

Flaws with the ANSI/ISO SQL-92 specification were identified in [11]. Their informal definitions offer multiple interpretations, some of which result in inconsistencies. Further, they do not account for known anomalies such as Lost Updates and Dirty Writes, or concurrency control mechanisms used in multi-versioned systems. Thus, Berenson *et al.* in [11] proposed a new set of correct, precise, isolation definitions, but acknowledged these definitions were disguised redefinitions of the earlier lock-based characterization in [51]. To rectify this Adya [1] introduced a framework extending the theory of multi-versioned serialization graphs to weak isolation; the differences between isolation levels being expressed in terms of specific cycles they prohibit in the serialization graph.

Using Adya's formalism the authors of [8] analyzed ACID[2] isolation levels and replicated data consistency guarantees through the lens of high availability. They proposed several new isolation levels: Monotonic Atomic View, Predicate-Cut and Item-Cut and provided a new definition for Snapshot Isolation. Building on [8] a new isolation level called Read Atomic was proposed in [9]. Recently, a new client-centric state-based formalization for isolation levels was proposed by [22].

Despite the aforementioned theoretical advancements, database concurrency control research has primarily focused on improving the performance of Serializable transaction

[1] Anomalies are colored red.
[2] Atomicity, consistency, isolation, durability.

processing [85]. Yet weak isolation is more commonly used in practice than Serializable isolation [85] and Read Committed is in fact the default isolation level for the majority of databases [8]. In Chapter 4, we attempt to address this disconnect by leveraging Adya's formalization of weak isolation to develop a high performance concurrency control protocol for many-core databases that supports transactions executing at a range of isolation levels.

2.1.3 Concurrency Control Approaches *(Relevant to Chapter 3)*

The algorithms used by schedulers vary and have developed in response to changes in hardware (technological advancements and costs reductions) and workload profiles. Broadly speaking, there are four types of concurrency control algorithms: lock-, timestamp-, graph- and validation-based, with some systems using a hybrid approach [113]. Additionally, these algorithms are classified into *pessimistic*, which assumes non-serializable behaviour will happen and pro-actively prevents such behaviour by delaying transactions, and *optimistic* approaches, which assume non-serializable behaviour will not happen and only fixes things when such behaviour is detected, normally by aborting and restarting the offending transaction(s).

We now provide a brief summary of each concurrency control approach.

Lock-based Lock-based concurrency control is a pessimistic approach in which transactions acquire locks to access data items. There is a variety of locking schemes, the simplest being shared/read and exclusive/write locks. Transactions hold all locks they have acquired (growing phase) until they have completed their operations, at which point they release locks (shrinking phase), this approach is known as *two-phase locking* (2PL); if all acquired locks are released together it becomes *strict two-phase locking* (S2PL) [39]. Locking can delay transactions but avoids rollbacks unless deadlocks occur. In 2PL deadlocks can occur when several transactions are waiting for a resource held by one of the others and none can progress. There are several deadlock management strategies: (i) maintain a waits-for

graph (deadlocks manifest as cycles), (ii) use timestamps and employ either *wait-die* or *wound-wait* to determine whether transactions should abort or wait for the lock, and (iii) prevent deadlocks happening; normally achieved by making transactions access data in some fixed order.

Timestamp-based Timestamp-based concurrency control is known as *timestamp ordering* (TO) and is an optimistic approach in which transactions are assigned unique (logical or real-time) timestamps, and additional meta-data is stored with each data item. Specifically, for a data item, the DBMS stores the timestamps of the latest reading transaction, and the latest writing transaction, along with a commit flag, indicating whether the latest writing transaction is active or not. The scheduler uses this information to determine whether a transaction's operations are physically realizable, or the transaction must be aborted and restarted. As a general rule, TO is more suited to workloads consisting mostly of read-only transactions, but will suffer when conflicts are high and will result in frequent aborts and retries. Compared with 2PL, TO does not delay transactions but can cause rollbacks, which leads to a more serious delay and wasted resources. Note, if the DBMS stores multiple versions per data item then *multi-version* TO, often referred to as *multi-version concurrency control* (MVCC) can be used, which decreases the chance a read operation will cause a transaction to abort. The scheduler now creates a new version of a data item for each valid write, assigned with the writing transactions timestamp. When scheduling a read operation it locates the version that was written immediately before the transaction started.

Validation-based Validation-based concurrency is also an optimistic approach (often referred to as *optimistic concurrency control* (OCC)) that allows transactions to proceed without acquiring locks or checking timestamps, but at the appropriate time performs some validation before aborting or committing a transaction [66]. For a transaction, OCC requires the scheduler be informed of its write and read sets prior to execution. Then, a transaction

is executed in three phases: (i) *read*, the transaction executes all reads in its read set and computes all values arising from writes in its write set into a private workspace; (ii) *validate*, the scheduler validates the transaction by comparing its write and read sets with other transactions. If successful the transaction proceeds to the next phase, else it is aborted; (iii) *write*, each write in the transaction's private workspace is written into the database. There are two validation strategies, (i) *backward-oriented*, validate the read set of the validating transaction with the write set of all transactions that were not committed before the start of the validating transaction, or (ii) *forward-oriented*, the write set of the validating transaction needs to be disjoint with all concurrent read-phase transactions.

Graph-based In graph-based concurrency control, often referred to as *serialization graph testing* (SGT), the scheduler maintains an acyclic serialization graph over the execution it controls. When a transaction desires to execute an operation, all resulting *conflicts* are calculated and corresponding edges inserted into the conflict graph. The operation proceeds provided the introduction of new edges did not create a cycle, in such cases the transaction is aborted. SGT has the theoretically optimal property of accepting all and only conflict serializable executions, offering a higher degree of concurrency and minimizing aborted transactions compared to alternative approaches. Several SGT variants exist [12] in addition to *basic-SGT* described above. *Conservative-SGT* never rejects any operations, but requires transactions to predeclare write and read sets so operations can be scheduled and executed with respect to some order that preserves the acyclicity of the conflict graph. In *certifier-SGT*, transactions execute operations optimistically, detecting conflicts as they progress, followed by a certification at commit time. There are several issues unique to SGT that must be addressed, specifically, (i) when nodes can be safely deleted from the conflict graph, (ii) how to detect conflicts, and (iii) how to check for cycles efficiently. SGT was, in fact, considered impractical due to (iii) the costs of cycle checking.

2.1.4 Many-Core Database Concurrency Control *(Relevant to Chapters 3 and 4)*

Definition 5 (Many-Core Databases) *A many-core database is a DBMS that is running on a single machine that has several CPU cores, often up to 100 cores.*

The strategies described in Subsection 2.1.3 do not scale well with recent hardware trends, namely, the advent of machines with many-cores, often in the magnitude of 100 cores. Due to 2PL's poor performance in such a setting, systems employ an optimistic strategy using timestamp allocation. These protocols have three key problems: (i) OCC requires an exclusive verification phase at commit time, which as the number of cores is increased, leads to high contention during this phase, resulting in extremely low throughput, (ii) such protocols require a global counter for timestamp allocation which can also become a bottleneck, and (iii) when there is a significant amount of conflict (high contention) between transactions then optimistic protocols typically exhibit a high number of aborts. These pitfalls have led to numerous protocols each offering new optimizations with performance varying dependent on workload characteristics, e.g., data skew, read ratio, payload size, table cardinality, and transaction size. Several notable examples are: Silo [110], MOCC [113], TicToc [122], BCC [123], and SGT [34] which are now described; for the interested reader see [25, 32, 42, 53, 58, 62, 63, 69, 88, 100, 114, 120, 121] for other contemporary protocols.

To combat problems with timestamp allocation *Silo* introduces a novel way to calculate transaction ids. The bits of transaction ids are divided in two, with higher bits representing a system-wide global epoch counter of the transaction's commit time, and lower bits being used to distinguish transactions within the same epoch. However, the lower bits do not capture the relative order within the same epoch, as a consequence only read-write dependencies can be captured, which restricts concurrency. To avoid assigning global timestamps to transactions *TicToc* lazily computes timestamps from a range of parameters such as read and write sets. TicToc then checks whether these timestamps are valid.

Mostly optimistic concurrency control (MOCC) attempts to mitigate OCC's poor performance under high contention by combining it with 2PL. For highly contended tuples, MOCC uses 2PL and falls back to OCC for less contended ones. This is an appealing approach but introduces the challenge of correctly detecting hot tuples, this behaviour is often transient and can vary largely over time. *Balanced Concurrency Control* (BCC) attempts to reduce the number of unnecessary aborts (false positives) in backward-oriented OCC. It proposes an improved validation rule to determine non-serializable transaction schedules that exploits *essential dependency patterns*. This vastly reduces aborts in high contention workloads, however, this approach uses the same execution model as vanilla OCC and thus suffers from the drawbacks of an exclusive verification phase.

Serialization Graph Testing takes a radically different approach in tackling the aforementioned problems with optimistic protocols. It dusts down the longtime discarded graph-based concurrency control strategy. SGT possesses the theoretically optimal approach to concurrency control, but has never been subjected to exhaustive research, nor has it been deployed in any commercial DBMS. An explanation for this is the general consensus that the approach is merely impractical, due to the costs associated with cycle checking. In [34] this claim is successfully refuted. Their key contribution was a highly concurrent graph data structure used to represent the CG. Access to their graph data structure is governed by a node-level locking structure. There are two types of node locks, *shared locks*, and *exclusive locks*. Shared locks allow concurrent edge insertion and cycle checks. Exclusive locks are taken only for commit critical checks. As stated in [12] a safe commit condition is to wait until a node in the CG has no incoming edges, thus, the exclusive lock can be taken and released quickly. If incoming edges exist, commitment is delayed. It has been demonstrated that this approach scales well on modern, many-core, hardware: (i) cycle checking proceeds in parallel, removing the bottleneck of OCC's single-threaded validation phase, (ii) nodes in the graph double up as transactions ids removing any dependence on global timestamps, and (iii)

it accepts all conflict serializable schedules, removing all unnecessary aborts. SGT is used as the basis for the development of the Wait-Hit Protocol in Chapter 3 and Chapter 4 utilizes the concurrent graph data structure from [34] in the development of MSGT.

2.1.5 Distributed Database Concurrency Control *(Relevant to Chapter 3)*

Definition 6 (Distributed Databases) *A distributed database is a DBMS that is running across multiple machines. In this thesis, a distributed database is assumed to have a shared-nothing architecture, with each machine responsible for managing a disjoint portion of the total database.*

A distributed, shared-nothing, DBMS is comprised of a number of *partitions*[3], each holding a disjoint portion of the complete database. In such a system, transactions can access data at multiple partitions. These transactions are known as *distributed* or *multi-partition transactions* (MPTs). Distributed transactions typically have one partition responsible for coordinating their execution, which issues remote operations to the required remote partitions. The fundamental difference between distributed and centralized transaction processing is that the *logically* single commit/abort operation must now take place in multiple places. The challenge here is two-fold. Firstly, it is important that each partition involved in the processing of a transaction agree on its outcome, that is, if one partition desires to abort then all must. Secondly, the nature of failures takes on a different complexion in a distributed database.

Definition 7 (Atomic Commitment Protocols) *An atomic commitment protocol, such as two-phase commit (2PC), is responsible for ensuring a collection of servers participating in a transaction, in the presence of partial failures, agree on the outcome of said transaction.*

[3] Also referred to as shards or sites.

In a centralized DBMS, failure is binary, either the system is up and transactions can be processed, or it has failed and no transactions can be processed at all. In a distributed system, however, there can be partial failures, some partitions may be functional while others have failed. Thus, the challenge can be reformulated as achieving consistent termination in the presence of partial failures. This challenge is delegated to an *atomic commitment protocol*, such as *two-phase commit* (2PC), which is used by the transaction's coordinator at commit time to bring the set of servers participating in the transaction to agreement on the transaction's termination status (commit or abort) in a manner which is resilient to failures.

2PC consists of two phases: *prepare* and *commit*. During the prepare phase, the coordinator requests a vote from the set of partitions involved in the transaction (participants) on whether they wish to commit or abort the transaction. Once the coordinator has collected the responses the commit phase begins. If any participant, the coordinator included, has voted to abort the transaction, the coordinator issues an abort message to all participants. Else, the coordinator broadcasts a commit message. When participants receive the commit or abort message, they perform any necessary clean up, and send an acknowledgment back to the coordinator. The coordinator also performs clean up during this phase but only responds to the client with the transaction outcome after it receives acknowledgments from all participants.

A simplistic view is that 2PC can be merely glued onto any concurrency control protocol. This is true for 2PL, which differs in a distributed database only in that locks are held until participants have received the commit/abort message during 2PC. However, other concurrency control protocols, such as OCC, require slightly different integration. For example, in OCC, the validation phase is now performed when a participant receives the prepare message from the coordinator. The outcome of the validation phase informs the response to the prepare message.

A comprehensive performance evaluation of distributed concurrency control was performed in [54], comparing four of the classic protocols described in Subsection 2.1.3: 2PL,

TO, MVCC, OCC. The study investigated: (i) the impact of workload factors: degree of contention, update rate, and the percentage of

MPTs, (ii) scalability: increasing the number of partitions/servers, (iii) the performance over wide-area networks (WAN), and (iv) various different application scenarios, namely, TPC-C [109] and the Product-Parts-Supplier (PPS) workload. Overall, the outlook on distributed concurrency control performance was bleak. This is illustrated by the MPT experiment, when transactions involved 2-4 partitions, throughput dropped by 12-40% across all protocols. The study attributed the performance inhibition of MPTs to: (i) overhead of sending remote requests, stalling and resuming execution, and (ii) 2PC. On a brighter note, the scalability experiment did demonstrate that performance gains are possible, albeit limited. Only once scaled to 64 servers did protocols display a 1.7–3.8× increase in throughput over single-server throughput. However, as workload contention increased this dropped to a 0.2–1.5× gain.

Interestingly, whilst research in many-core concurrency control has shunned lock-based approaches for optimistic ones, across the range of experiments performed it was 2PL with `NO-WAIT` deadlock detection that performed best. OCC's poor performance was attributed to the overheads of validation and copying during transaction execution, whereas both MVCC and TO block newer transactions that conflict until the older ones commit, which when data items are highly contended can increase latency, even if the overall abort rate is low.

The differing optimal approaches for many-core and distributed databases is not compatible with cloud environments in which users expect matching performance increases as resources are scaled up and out. For example, users may initially scale up their system and thus opt for an optimistic approach, before then scaling out and then being left with a sub-optimal concurrency control protocol for their new scale point. This raises the following question: how to design a concurrency control protocol that performs equally well in both many-core and distributed databases. Designing such a protocol is tackled in Chapter 3.

2.1.6 2PC: Research Strikes Back *(Relevant to Chapter 6)*

Given the somewhat bleak picture of distributed transaction processing emphasized by the study in [54], it has become a fertile area of research and numerous approaches to improve performance have been proposed.

Several systems attempt to *minimize* or *eliminate* distributed transactions. Schism uses a workload-driven partitioning scheme to minimize distributed transactions [23], while G-Store [24] and LEAP [70] eliminate them through dynamic data repartitioning. *Mandating determinism* also avoids 2PC [90, 103, 104]. The idea behind deterministic databases is to order and replicate transactional inputs prior to execution using a centralized sequencing layer, thus eliminating coordination between servers [103]. There are two key advantages to this approach, atomic commitment can be avoided, and it greatly simplifies replication [91]. The study in [54] included one deterministic approach, Calvin [104]. In most experiments Calvin performed the best, but the dice were unfairly weighted as it avoids 2PC and other message passing. Calvin suffers when transactions involve conditional operations, may internally abort, or involve foreign key lookups. The study found when the workload included these most protocols' throughput was only affected by 2–10%, but Calvin experienced a 36% decrease – determinism is not a silver bullet. An important caveat with deterministic databases is that transactions' read and write sets be known prior to execution. If it cannot be met, a reconnaissance phase is executed to discover these sets which amounts to running a transaction twice. Aria [71] avoids this caveat by using a deterministic reordering mechanism, but its performance suffers under high contention workloads [72]. Prognosticator [60] circumvents this limitation by using symbolic execution to build key-level transaction profiles, which are used to execute transactions with a high degree of parallelism.

Another category of systems *weaken isolation* and *relax consistency guarantees* to achieve better performance, offering Snapshot Isolation [33, 36, 96] or *highly available transactions* (HATs) [8]. Another group of systems opt for a *blended approach* by combining concurrency

control, replication, and commitment into a unified protocol to amortize the costs of 2PC. The principal aim here is to minimize the number of WAN round trips and hence latency is reduced (see MDCC [64], TAPIR [124], Ocean Vista [43], Janus [79], Helios [81], and G-PAC [74]). Another related technique is CockroachDB [98] which uses *Parallel Commits* to halve the latency of distributed transactions by concurrently performing consensus round trips required to commit a transaction.

Several recent OLTP databases utilize epochs to exchange improved throughput for higher latency. Obladi tackles the problem of data access privacy by combining Oblivious RAM and epochs to hide access patterns [21] from cloud providers. STAR [73] runs distributed transactions and single-node transactions in different epochs. COCO [72] leverages epochs to mitigate against the costs of 2PC and synchronous replication. In Chapter 6 we evaluate the performance of epoch-based commit and develop epoch-based multi-commit, which to our knowledge, is the first protocol to combine epochs and data access patterns to minimize aborts in the presence of node failures.

2.2 Graph Processing

(Relevant to Chapter 5)

Graphs have long been a useful mathematical abstraction and commonplace data structure across many facets of computer science. Recent years have seen a proliferation of the development and use of dedicated graph processing technologies [14]. The emergence of these technologies appears to be motivated by organizations recognizing the potential value that can be extracted from rich, highly-connected data sets, and the perceived shortcomings of existing *relational database management systems* (RDBMSs) which are not suitable for serving graph workloads. Thus, organizations often complement their existing data processing pipelines with graph processing systems [93].

It is well established there is no single *"one-size-fits-all"* data management technology that can serve all possible uses of data efficiently [97] which holds true within the realm of

graph data management, owing largely to the variety exhibited across graph workloads. As such graph processing systems broadly fall into one of three categories:

- **Graph databases** focus on transactional online graph persistence and are accessed in real time from some application. Queries are either: (i) *local*, involving a single vertex or edge, (ii) *neighborhood*, retrieve all edges attached to a given vertex, or (iii) *traversals*, exploring a part of the graph beyond a single neighborhood. In each case only a subset of the graph is required to satisfy a query. Examples include Neo4j [92], JanusGraph [61], TigerGraph [106], and Dgraph [29].

- **Graph Analytical Frameworks** focus on offline analytics, supporting global graph algorithms such as clustering and community detection. Most computations involve the whole graph. Examples include GraphX [50], Giraph [49], and GraphChi [67].

- **Graph Streaming Engines** focus on supporting analytics over a continuously changing graph. Typically these systems offer a simple graph-based data model and lack transactional support [13]. Examples include Aspen [30], Kineograph [19], and GraphOne [65].

Given the expressiveness of graphs and the variety of system types, application areas are wide reaching from healthcare, to social networks and fraud detection [35]. The work in this thesis (Chapter 5) is primarily concerned with *graph databases*. In such systems data, the most common data model is the *labeled property graph* [92]. In the property graph model each vertex has a unique ID, a label indicating the type of vertex, a set of incoming and a set of outgoing edges and a collection of key-value properties. Each edge has a unique ID, its start and end vertices, a relationship type label and a collection of properties.

2.2.1 Graph Data Consistency

Definition 8 (Reciprocal Consistency) *Reciprocal consistency is an invariant that ensures the two physical edge pointers that comprise a logical edge in a graph database point to one another.*

Regardless of application-level semantics, the property graph data model imposes a fundamental consistency guarantee that must always remain valid: Reciprocal Consistency. In the property graph model, edges have direction and each edge runs from a *source* vertex to a *destination* vertex. In the storage layer, however, edge directionality does not exist; both the source and the destination vertices store information about each other. This allows edge traversal to be bidirectional and speeds up query performance.

Consider, for example, the statement: "Tolkien *wrote* The Hobbit". It is expressed using vertex *a* for "Tolkien" and vertex *b* for "The Hobbit", and an edge *wrote* running from *a* (source) to *b* (destination). Corresponding openCypher [83] code [4] is given below and Figure 2.2(a) shows the model level view of edge *ab*.

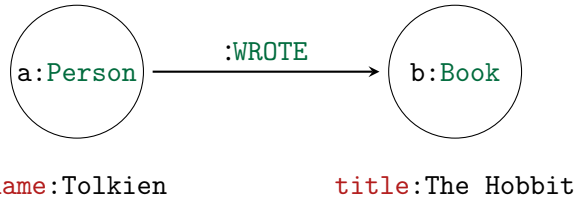
```
MATCH (a:Person), (b:Book)
WHERE a.name = 'Tolkien' AND b.title = 'The Hobbit'
CREATE (a)-[w:WROTE]->(b)
```

Figure 2.2(b) depicts the internal representation of a graph arising from JanusGraph [61] and TitanDB [108]. A vertex, such as *a*, is represented by a record that contains one or more properties of that vertex, followed by a sequence of *edge pointers* pointing to all those vertices to which this vertex is related either as a source or a destination. The sequence of edge pointers is also called the *adjacency list*. It can be seen in Figure 2.2(b) that *a*'s adjacency

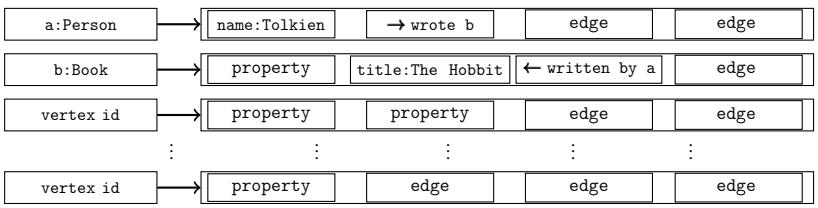
[4]There are many graph query languages, with each vendor often developing their own proprietary language. Many offer a declarative language, e.g., Neo4j's Cypher [45], TigerGraph's GSQL [106] and Oracle's PGQL [84], with others offering the Gremlin API [107]. There have been a few concerted efforts to standardized graph query languages. Efforts include G-Core [5], OpenCypher [83], and more recently, GQL and SQL/PGQ [27].

list has an edge pointer entry that stores "*a wrote b*" while *b*'s list has a corresponding entry storing the reciprocal (or inverse) information "*b written* by *a*". When the adjacency list entries for a given edge refer to each other in a complementary manner like this, that edge is said to exhibit Reciprocal Consistency.

Consider a query: "list all titles by the author who wrote The Hobbit". This query needs to start from *b* which represents the only entity specified explicitly in it. Thanks to the reciprocal information in *b*, it can reach *a* from *b*, even though edge *ab* is "directed" from *a* to *b*, and then compile the necessary list from *a*. Note that Reciprocal Consistency is assumed to prevail when a query reads only the source or destination vertex of an edge.



(a) Logical view.



(b) Storage view.

Fig. 2.2 Logical and storage views of a reciprocally consistent edge *ab*.

2.2.2 Distributed Graph Databases

Definition 9 (Distributed Graph Databases) *A distributed graph database is a distributed DBMS that is designed for managing graph data spread across several machines.*

In practice, graphs can be extremely large, sometimes in the magnitude of 100 billion edges [93], exceeding the storage capacity of a single-node graph database and motivating the need for *distributed graph databases*. A distributed graph database employs a shared-nothing

architecture, partitioning a graph among loosely cooperating servers. Graph partitioning is non-trivial and a common approach is to use a k-balanced edge cut [56]. The objective of such an approach is to minimize the proportion of edges that span partitions and also to balance the distribution of vertices to partitions. Figure 2.3 depicts a graph database partitioned across three servers, $S_i, i = 1,2$ and 3. Typically, each partition would be replicated for fault tolerance and availability. Intra-partition and inter-partition edges are respectively referred to as *local edges* and *distributed edges* (shown using dashed lines in Figure 2.3). The proportion of distributed edges is not negligible and can range from 25-75% [56].

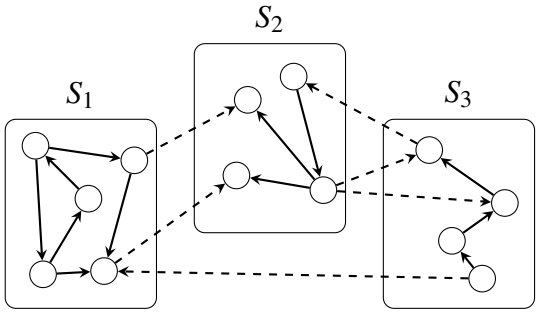


Fig. 2.3 Local and distributed edges

2.2.3 Data Corruption in Distributed Graph Databases

Recent work [41, 119] highlighted that a common distributed graph database architecture permits violations of Reciprocal Consistency in distributed edges, introducing corruption into the database. Such an architecture uses a NoSQL database [89], e.g., Apache Cassandra [6], for storage and is then adapted with a query language expressed in terms of edges and vertices along with some glue-code to bind that interface to the underlying database. Opting for this design appears to be a good choice: it offers the application programmer the modeling convenience of graphs together with the operational characteristics of the underlying highly scalable NoSQL database. However, a major problem with this design option is, in a desire to offer higher performance, a lack of transactional semantics from the underlying database

which seldom provide guarantees for multi-operation, multi-object transactions that span partitions. Without concurrency control across partitions, concurrent updates of distributed edges can interleave and violate Reciprocal Consistency.[5] Moreover, due to the *scale-free* property exhibited by many real world graphs, this corruption can propagate through the database at alarming rates, rendering it irreversibly corrupt. We now describe in detail the mechanism by which this corruption can occur.

Suppose that the edge `(a)-[:WROTE]->(b)` is a distributed edge, with vertices a and b in servers S_i and S_j, $j \neq i$, respectively. When a transaction writes this edge ab: (i) two writes are performed: reciprocal entries in the adjacency lists of nodes a and b are updated, and (ii) write order is unconstrained: a transaction is equally likely to write a then b as it is to write b then a. Concurrent transactions T_x and T_y can interleave in the following three ways and each one is depicted in Figure 2.4:

(a) T_x starts before T_y; it writes a at S_i first and then proceeds to S_j across the network; T_y operates the other way round, beginning with S_j and proceeding to S_i (see Figure 2.4(a)). The net effect is: $T_x \rightarrow T_y$ at S_i and at $T_y \rightarrow T_x$ at S_j, where $T \rightarrow T'$ at S denotes that T *precedes* T' at server S.

(b) Same as the previous case, except that T_y starts earlier than T_x (see Figure 2.4(b)).

(c) Same net effect as in previous two cases, except that both T_x and T_y start their first writes at S_i, $T_x \rightarrow T_y$, but T_y overtakes T_x in reaching S_j where $T_y \rightarrow T_x$ (see Figure 2.4(c)).

We could envisage three more corresponding cases (a') - (c') where the roles of T_x and T_y in cases (a) - (c) are simply interchanged; e.g., in case (a'), T_y starts before T_x, writes a at S_i first and then proceeds to S_j across the network; T_x operates the other way round, beginning with S_j and proceeding to S_i. Thus, there are only six ways concurrent T_x and T_y can interleave. The arguments we make based on cases (a) - (c) of Figure 2.4 equally apply,

[5]The concurrency control primitives provided by NoSQL databases are typically sufficient to ensure Reciprocal Consistency for local edges.

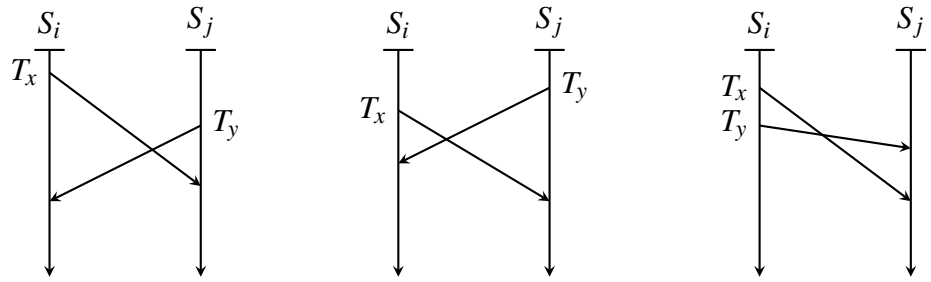


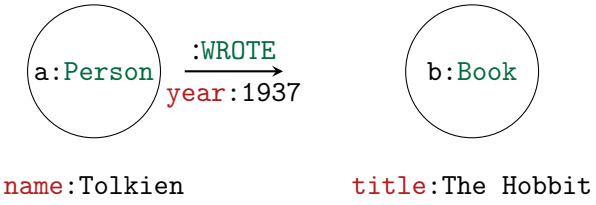
Fig. 2.4 Interleavings of concurrent writes to a distributed edge by transactions T_x and T_y.

by symmetry, to cases (a') - (c') and so, for brevity, we will not consider the latter. At the end of each case in Figure 2.4, the last update on a is by T_y and that on b is by T_x. Unless updates of T_x and T_y are commutative, ab cannot be reciprocally consistent. Note that the case of Figure 2.4(c) can be avoided easily by using sequence numbers and by exploiting the fact that both T_x and T_y modify the edge starting from the same end S_i. Each end of a distributed edge maintains *start sequence number* (ssn) to indicate the number of transactions that modified the edge starting from its end. It also maintains a *finish sequence number* (fsn) to indicate the number of transactions that modified the edge starting from the other end. A transaction (such as T_y) should not modify the edge at the remote end (at S_j) until its ssn is equal to one less than the fsn found at the remote server. After modification(s), the transaction sets $fsn = ssn$. Thus, only cases (a) and (b) of Figure 2.4 are of concern. As an example, suppose that T_x deletes the *wrote* edge while T_y concurrently appends a property *year*:

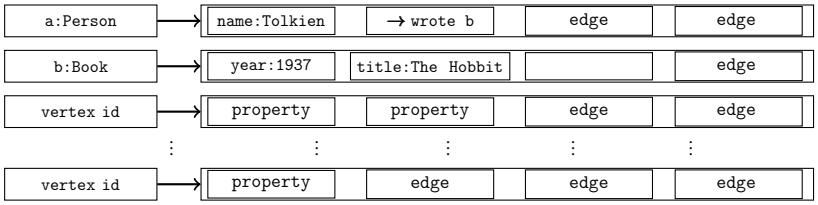
```
// Tx
MATCH (a:Person)-[w:WROTE]->(b:Book)
WHERE a.name = 'Tolkien' AND b.title = 'The Hobbit'
DELETE w


// Ty
MATCH (a:Person)-[w:WROTE]->(b:Book)
```

```
WHERE a.name = 'Tolkien' AND b.title = 'The Hobbit'
SET w.year = 1937
```



(a) Logical view.



(b) Storage view.

Fig. 2.5 Logical and storage views of a reciprocally inconsistent edge *ab*.

The interleaving patterns depicted in Figure 2.4 leave *ab* in violation of Reciprocal Consistency, as shown in Figure 2.5. When T_x and T_y do not interleave, either $T_y \rightarrow T_x$ or $T_x \rightarrow T_y$ holds at both servers S_i and S_j; in that case, the last update on *both* a and b would be by either T_x or T_y, respectively. That is, when transactions update at both ends of *ab* in *some* arbitrarily chosen but identical order, *ab* is left reciprocally consistent after each transaction's update. When interleaving updates leave *ab* reciprocally inconsistent, *ab* can be said to have become *half-corrupted* because if $T_y \rightarrow T_x$ is the chosen order between T_y and T_x, then the edge pointer in b of *ab* is in error; otherwise, the edge pointer in a is erroneous. Thus, a reciprocally inconsistent edge *ab* certainly has a corrupt half but the question of exactly which half is corrupt is decided by what happens subsequently. Suppose that a future transaction T_w *first* reads the edge pointer of reciprocally inconsistent *ab*, say, at vertex a. (Note that when T_w reads an edge, it does not check for reciprocal consistency). At that moment, T_w (implicitly) chooses the order $T_x \rightarrow T_y$ and thereby invalidates the other order

$T_y \rightarrow T_x$ that prevails at vertex b. Thus, from that moment onward, the b end of edge ab becomes the corrupt end and, conversely, the a end becomes the *correct* end.

```
// Tz
MATCH (b:Book),(u:Person)
WHERE NOT (:Person)-[:WROTE]->(b:Book)
AND u.name = 'unknown'
CREATE (u)-[:WROTE]->(b)
```

If no transaction ever reads the edge pointer at vertex b, then the order $T_x \rightarrow T_y$ effectively prevails and the half-corruption of ab remains invisible to the rest of the database. However, if T_z is to subsequently read the edge pointer at b and write another edge based on what it read, i.e., The Hobbit has unknown author, then it is introducing updates not consistent with what T_w read earlier; it thus introduces *semantic corruption* into the database. Further writes based on reading semantically corrupt data also spread corruption. A database is said to be *operationally corrupt* when a significant proportion of its data records are in a semantically corrupt state [41].[6]

To illustrate the severity of this problem, the authors of [41, 119] constructed an analytical model to analyze the time until operational corruption. They found for fair assumptions regards database size, proportion of distributed edges, and network latencies, even with modest transaction arrival rates the database was often found to be operationally corrupt in under 50 hours!

Definition 10 (Edge-Order Consistency) *Edge-order consistency is the guarantee that updates by transactions to multiple edges happen in the same order across all edges.*

[6]Two relevant remarks on past works: a half-corrupted edge is due to a *dirty write* (ANSI *P0* [11], Adya *G0* [2]) in the context of distributed graph databases. If the database provides the ANSI isolation level *Read Uncommitted* across **all** objects, it will identically order the writes of concurrent transactions and this would prevent all interleaving patterns shown in Figure 2.4 and thus avert half-corruption altogether.

Following on from Reciprocal Consistency, we also observe a different type of possible conflict that can arises when transactions update more than one edge during their lifetime. Suppose that transactions T_A and T_B both update edges e and e', and do without interference either among themselves or with other transactions. It may happen that e is updated by T_A before T_B, while e' is updated by B before A, as illustrated in Figure 2.6 (here time flows from left to right and the conflict-free reciprocally consistent updates are collapsed to single instants). Such an occurrence, if allowed, would violate the property of Edge-Order Consistency between transactions.

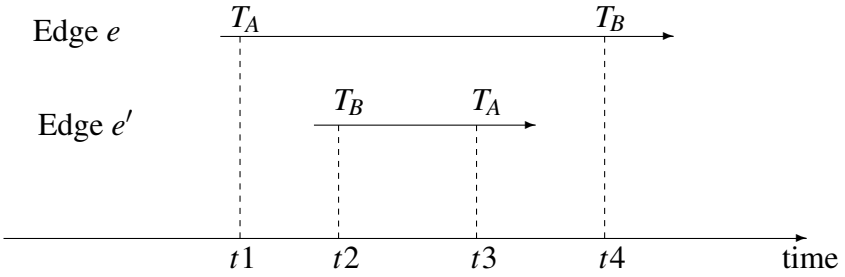


Fig. 2.6 Edge-Order Consistency violation

These issues will be taken up in Chapter 5 where we develop lightweight concurrency control protocols for maintaining Reciprocal Consistency and Edge-Order Consistency, thus avoiding data corruption.

2.3 Performance Evaluation Techniques

The performance of the protocols presented in this thesis are evaluated and benchmarked using a range of techniques: analytic models, simulations, and direct implementations allowing for experimentation with real hardware.

Analytic models allow for the representation of a protocol and its interactions within a system as mathematical expressions. Once derived, such models have a number of advantages [75]. They can be used to quickly garner insights into protocol behaviour under

a variety of different conditions allowing for a quicker feedback loop compared to direct implementations as the critical factors that influence protocol can be easier distilled. Additionally, the process itself of deriving analytic models can be helpful in improving the theoretical understanding of protocols. Lastly, analytic models act as a useful litmus test that can inform the decision of moving to a more labour-intensive implementation within a real system. Analytic models are not without their limitations. They are dependent on their simplifying assumptions, which make them tractable, but can result in a mismatch with reality, as George Box said, "All models are wrong, some are useful". Therefore, validation against real-world data is imperative for ensuring the applicability of analytic models in accurately predicting protocol performance.

Simulations are step up from analytic model in terms of how closely they approximate reality, attempting to mimic an often complex protocol through a virtual model [78]. Simulations share many advantages and limitations with analytic models. They allow for the systematic exploration of a wide range of scenarios through the varying of simulation parameters. Some of these scenarios may be difficult to reproduce in reality, either due to time or financial constraints, hence simulations are often much more cost-effective than direct implementations. However, simulations are also sensitive to the validity of the assumptions made within them. This can have a large impact on the accuracy of the simulation's predictions. Again, as with analytic model it must be emphasized that validation with real-world data is vital for extracting reliable insights from simulations and caution must always be exercised when interpreting simulated results.

Direct implementations are the closest approximation to the reality protocols will experience in practice, but they are by far the most costly in terms of implementation effort and hardware resources. Being an accurate representation of the complexity faced in the real world means that direct implementation can undercover issues not considered during the modeling phases, which can in turn facilitate the development of improved models.

Therefore, direct implementation is important for the validation of analytic models and simulations. With direct implementations there is the challenge of selecting or developing an appropriate benchmark [59]. In Subsection 2.3.1, the evaluation framework used for direct implementations is presented along with the benchmarks used and why they were selected.

2.3.1 Evaluation Framework *(Relevant to Chapters 3 and 4)*

At various points in this thesis protocols are implemented in our evaluation framework[7]. The framework contains a prototype in-memory many-core database, which has a single versioned storage layer and a modular transaction scheduler, and is extendible for multiple workloads; requiring the implementation of a parameter generator, loader, and stored procedures. Within the framework each core acts as an independent client generating transactions, thus the protocols experience a truly concurrent workload.

Unless stated otherwise, experiments were performed using an Azure Standard D48v3 instance with 48 virtualized CPU cores and 192GB of memory. Prior to each experiment, tables are loaded, followed by a warm-up period, before a measurement period; both are of configurable length, we use 60 seconds and 5 minutes respectively in this thesis. We measure the following metrics:

- ***Throughput:*** number of transactions committed per second.
- ***Abort rate:*** rate at which transactions are being aborted.
- ***Average latency:*** the latency time of committed transactions (in *ms*) averaged across the measurement period.

We now describe the workloads implemented in the evaluation framework: YCSB, SmallBank, and TATP. There are several OLTP workloads that could have been selected for implementation; OLTP-bench [31] lists 15. The workloads described below were chosen as

[7]https://github.com/jackwaudby/spaghetti

they offer sufficient coverage to empirically exercise protocols under a breath of scenarios. The transactions in the TATP workload generate few conflict cycles, so there should be few aborts. Whereas, SmallBank was specifically designed to generate non-serializable executions, hence there should be a higher abort rate. It would be expected that optimistic protocols would perform better on TATP, and pessimistic protocols on SmallBank. Lastly, YCSB offers a high degree of configuration, which allows us to explore performance when other workload dimensions change, e.g., proportion of update transactions. A notable exception from the evaluation framework is TPC-C [109], which was excluded a some of its transactions require functionality, such as scans, that are not currently supported in the framework. The workloads are summarized in Table 2.1

Workload	Tables	Columns	Transactions	Read-Only Transactions
TATP	4	51	7	40%
YCSB	1	11	6	50%
SmallBank	3	6	6	15%

Table 2.1 Profiles for the evaluation framework workloads.

2.3.2 YCSB

The YCSB [20] was originally designed to evaluate large-scale Internet applications, it is re-purposed in this thesis as an OLTP microbenchmark. It has one table with a primary key and 10 additional columns each with 100B of random characters. For all our experiments in this thesis, we use a YCSB table of 100K rows. There are two types of transaction: read or update, each contains 10 independent operations accessing 10 distinct items. Update transactions consist of 5 reads and 5 writes that occur in random order. Read transactions consist solely of read operations. The proportion of update transactions is controlled by the parameter, U. Data contention, when multiple transactions try to read or write the same database items, follows a Zipfian distribution, where the frequency of access to sets of hot records is tuned using a skew parameter, θ. When $\theta = 0$, data is accessed with uniform

frequency, and when $\theta = 0.9$ it is extremely skewed (high contention). For Chapter 4, in order to measure the impact of transactions running at weaker isolation we introduce an additional parameter, ω, which controls the proportion of transactions running at Serializable isolation. The remaining transactions are split between Read Committed (90%) and Read Uncommitted (10%).

2.3.3 SmallBank

SmallBank mimics a basic banking application and comprises of six transactions that perform simple operations on customers' accounts. It was designed to generate non-serializable schedules when executed at weak isolation levels (in contrast to TPC-C) [3]. The workload configuration used in this thesis is derived from OLTP-bench [31], a standardized benchmarking tool, thus 25% of all operations are executed on a hotspot area of 100 accounts. The contention levels are varied by adjusting the number of accounts: high contention (100 accounts), mid contention (1000 accounts), and low contention (10000 accounts).

2.3.4 TATP

The *Telecommunication Application Transaction Processing* (TATP) benchmark models a telecommunications application [102]. It consists of three read-only transactions and two update transactions. As regards contention, the primary key distribution uses a non-uniform row access pattern to increases tuple contention. The interesting property of this workload is that TATP transactions can generate few conflict cycles. In fact, only if two `UpdateSubscriberData` transactions interleave by accessing the same keys can a cycle be created in the conflict graph. As a consequence, schedulers that accept all valid executions should seldom abort.

Chapter 3

Wait-Hit Protocol

Summary

This chapter presents the design of the Wait-Hit Protocol, a general purpose concurrency control protocol for any scale. It can effectively scale vertically, as the core count is increased on a given machine and horizontally, as the number of machines is increased in a cluster, without having to relax the gold standard isolation level (Serializability). The protocol takes an optimistic approach, letting transactions collect dependencies efficiently as they execute, and before performing a commit time validation. Based on the type of dependencies collected (write-write, write-read, read-write) a validating transaction will: (i) abort **(hit)** in-flight transactions that if allowed to commit could result in non-serializable behaviour, and/or (ii) delay itself **(wait)** for an in-flight transaction to complete so that dirty reads are avoided.

3.1 Introduction

Cloud providers enable seamless scaling from a few-core machine, to a many-core machine, to deployments spread across data centers and the globe. Users deploying databases in the cloud expect OLTP DBMS performance to scale with hardware resources. Unfortunately, as seen in Subsections 2.1.4 and 2.1.5, many concurrency control protocols do not scale well [54]; a protocol architected for one scale point inevitably needs to be re-adapted for another, or a different strategy altogether used. This raises the challenge of designing high-performance concurrency control protocols that can effectively scale from few-core, to many-core, to the globe whilst offering good performance at each scale point, without having to relax from Serializable isolation. This chapter describes the design and evaluation of the Wait-Hit Protocol which has been developed specifically for this task. To be precise, the Wait-Hit Protocol is a family of optimistic concurrency control protocols each targeting a different deployment.

The rest of this chapter is structured as follows. We begin in Section 3.2 by discussing the design goals of the Wait-Hit Protocol. Section 3.3 presents the principles behind the wait-hit approach. Section 3.4 gives a description of the Basic Wait-Hit Protocol, implementation details, and an evaluation demonstrating the need for the many-core adjustments. Section 3.5 then presents the Many-Core Wait-Hit Protocol, discusses implementation details and optimizations, before evaluating the protocol's performance. Next, Section 3.6 describes the Distributed Wait-Hit Protocol. Finally, Section 3.7 draws conclusions and discusses future work.

3.2 Design Goals

We focus on two database architectures, many-core and distributed. Recall from Subsection 2.1.4, optimistic approaches typically perform better than pessimistic approaches in many-core databases. Moreover, high performance can be achieved by: (i) avoiding global timestamp allocation, (ii) avoiding an exclusive validation phase, and (iii) minimizing unnecessary aborts. As described in Subsection 2.1.5, for distributed databases the converse is true: pessimistic approaches generally perform better than optimistic approaches. For optimistic approaches performance is constrained by the validation phase. For example, in OCC more than 50% of time can be spent in the validation phase, or in a queue waiting to enter the validation phase [54]. An optimistic approach that reduces validation overheads could offer performance comparable to lock-based approaches in distributed databases. Note, it must be stated that 2PC is the dominating performance factor in a distributed database, but the chosen concurrency control protocol still influences performance [54].

A concurrency control protocol that scales well with cores and database nodes should satisfy all the above criteria. Of recently proposed many-core concurrency control protocols, SGT achieves all many-core design goals. However, unfortunately SGT once again appears impractical in a distributed database. Consider the distributed SGT sketch in [34], each

partition stores a local graph representing conflicts between its local transactions, with a special node type indicating a remote partition; this must store information needed to correctly resolve all edges in the global conflict graph. The commit procedure remains unchanged in that transactions delay until they have no incoming edges, but now they must execute 2PC after this. The problem is that SGT effectively validates each operation before execution via a cycle check, which in a distributed database incurs significant remote access overhead. Cycle checking would require traversing a graph distributed across several machines, incurring network hops each in the magnitude 1-10ms. Additionally, due to non-locality of access during traversals, servers not directly involved in the processing of the validating transaction may be contacted. These two factors conspire to increase the time a transaction spends in the database, in turn increasing contention. Contention implies more conflicts and hence increases size of the conflict graph to be traversed, thus more time spent cycle checking and large transaction lifetimes – a vicious circle.

SGT's validation strategy minimizes unnecessary aborts, but maintaining this property appears too costly in a distributed environment. The wait-hit approach borrows aspects from SGT, but crucially allows some unnecessary aborts in order to gain a highly parallelizable lightweight validation strategy which performs well in both a many-core and distributed database.

3.3 Wait-Hit Approach

The wait-hit approach aims to get transactions in and out of the database as fast as possible through optimistic execution and a low-overhead validation phase, minimizing the window in which transactions can contend. Similarly to SGT, it is assumed transactions can cheaply detect conflicts via meta-data existing on data items. An important point is that the wait-hit approach detects all *direct* and *transitive* conflicts collectively referred to as a transaction's *predecessors*. To illustrate this consider T_i reads x, T_j then writes x, then T_k writes x. T_k

directly conflicts with T_j and transitively conflicts with T_i, thus T_k's predecessors are T_j and T_i. For example, in Table 3.1, for the data item "Jack", T_2 would detect T_1 and T_0 as predecessors.

During execution, rather than converting conflicts into edges, inserting them into a conflict graph, and performing a cycle check as in SGT, predecessors are grouped based on whether the conflict was detected by a write operation (ww and rw conflicts), referred to as *predecessors-upon-write*, or by a read operation (wr conflicts), referred to as *predecessors-upon-read*.

At commit time the compressed representation of a transaction's conflict information is used to perform validation. A transaction's predecessors reflect its local view of the expected serialization order. However, local serialization orders across transactions may differ, i.e., there exists a cycle in the (global) conflict graph. Thus, validation must ensure that, using per-transaction predecessor-upon-write and predecessor-upon-read information, the acyclic invariant is preserved. The wait-hit approach achieves this through a combination of forcing a transaction's predecessors to abort, making transactions check if they themselves have been forced to abort, and by waiting for transactions' predecessors to terminate.

To implement these rules, in addition to transactions tracking their predecessors, the wait-hit approach relies on a *hit list* containing the predecessors of previously committed transactions. If these predecessors are permitted to commit it *could* result in non-serializable behaviour (details to follow). When validating, transactions check if they exist in the hit list, aborting if so, this rule is referred to as the *hit rule* and is a key aspect of the wait-hit approach. Additionally, the wait-hit approach requires a *terminated list* containing the outcome (commit/abort) of terminated transactions. Note, all transactions enter their outcome in the terminated list. The validation is divided into two stages: hit phase and wait phase.

The first step in the hit phase for transaction T is checking whether another transaction has forced it to abort by entering it in the hit list. After this, T considers each of its

predecessors-upon-write and attempts to forcibly abort **(hit)** them. There are four cases to consider for each predecessor P in the set of predecessors-upon-write for T: (i) P has committed, but did not have T as a predecessor, (ii) P has committed and detected T as one of its predecessors-upon-write, (iii) P has aborted, and (iv) P is still executing, i.e., it is in-flight.

In case (i), P cannot be involved in a cycle with T, as if it had directly or transitively conflicted with T, then T would have been present among P's predecessors-upon-write and hence entered T into the hit list, thus P can be ignored. In case (ii), T must abort as P and T are involved in a cycle. By traversing conflicts, T is reachable from P and P from T, thus there exists a cycle in the conflict graph. Note by the hit rule, P will have entered T into the hit list, thus the hit check ensures T aborts. For case (iii), an aborted P cannot introduce a cycle with T so can be ignored. Lastly, for (iv) if T completes validation it will then enter P into the hit-list, essentially "hitting" them ensuring that P aborts when it validates by the same logic as case (ii). This avoids non-serializable behaviour occurring, but is pessimistic as it is possible that P does not have T as a predecessor and unnecessarily aborts.

The second step considers predecessors-upon-read. As in SGT, the wait-hit approach allows transactions to optimistically read dirty records. Thus, for each predecessor P in the set of predecessors-upon-read for T there are three possibilities to consider: (i) P has aborted, (ii) P has committed, or (iii) P is in-flight.

For case (i), T must also abort. This is required to ensure T has not acted upon aborted data. For case (ii), T has read from a committed transaction which is permissible and no further action is needed. As regards case (iii), the fate of P is unknown, it may commit or abort. If T is permitted to wait until P has terminated, then correct action could be taken (case (i) or (ii)), unfortunately this could lead to a deadlock. Consider if T had P as a predecessor-upon-read, and vice-a-versa, then both would indefinitely wait for the other to terminate! Note that if T and/or P are read-only transactions, then there is no possibility

of a deadlock – a feature we will exploit later in Subsection 3.5.2. For now, deadlocks are resolved by employing a zero-wait strategy, i.e., in the event of case (iii), T would simply abort itself. This is, of course, a heavy-handed, potentially abort-heavy strategy that may result in unnecessary aborts. However, given that the wait-hit approach essentially compresses conflict information it is necessary, unlike SGT which, by storing a conflict graph, can execute a cycle check to arrive at the correct decision. We present a smarter wait strategy in Subsection 3.5.2.

To summarize, the wait-hit approach validates a transaction by: (i) aborting predecessors-upon-write, that is, *hitting* them and checking if itself has been hit, and (ii) delaying until its predecessors-upon-read completes, that is *waiting*. This is the core essence of the wait-hit approach which simplifies validation greatly. We position this approach as a middle ground between a graph-based approach, e.g., SGT [34], and a single-threaded execution model, e.g., H-Store [97].

3.4 Basic Wait-Hit Protocol

In this section, we describe the Basic Wait-Hit Protocol which implements the wait-hit approach in Section 3.3 using *exclusive* data structures to represent: (i) hit list, (ii) terminated list; an entry with -1 denotes the corresponding transaction aborted and 1 denotes it committed, and (iii) transaction id generator. To ensure exclusive access to data structures the hit and terminated lists are wrapped with a `Mutex` lock and the transaction id generator uses an atomic counter. Each transaction stores two per-transaction data structures to track a transaction's predecessors-upon-read and predecessors-upon-write. In discussions here we assume the database has a set of threads called *workers* that execute transactions.

3.4.1 Protocol Description

Before execution of transactions the database initializes the hit list, terminated list, and transaction id generator. When a worker receives a transaction T it gets an id from the id generator and initializes the predecessors-upon-write and predecessors-upon-read containers for that transaction.

The database optimistically executes T collecting predecessors encountered using the per-item access history (see Table 3.1) and records them as either predecessors-upon-read or predecessors-upon-write depending on the operation performed (see Algorithm 1). When T performs a read operation it reads the latest version of a given item and includes the ids of all transactions that previously wrote to that item amongst its predecessors-upon-read. Note, the predecessor may be still in-flight (at the time of reading). When T performs a write operation on an item it stores the id of all transactions that wrote and read the item before among its predecessors-upon-write. After executing an operation a transaction appends its id and operation type to the access history for the item it accessed.

At commit time, T enters the validation procedure given in Algorithm 2 consisting of two phases, a wait phase and a hit phase. First, T enters the hit phase, acquires the exclusive lock and checks whether it exists in the hit list (Lines 1-2, Algorithm 2). If it does, then another already terminated transaction has directly or transitively conflicted with it and there is a possibility of non-serializable behaviour, so validation fails, the lock is released, and the abort procedure given in Algorithm 3 is executed: T removes itself from the hit list, then acquires the terminated list lock and appends itself with -1. Else, it continues to the next step: T adds its predecessors-upon-write into the hit list after filtering out those which have already terminated for garbage collection reasons explained in Subsection 3.4.2 (Lines 4-7, Algorithm 2). The wait phase (Lines 10-15, Algorithm 2) deals with predecessors detected upon a read. For each predecessor, if it has committed no further action is needed, else (it is

in-flight or aborted) T is aborted. Upon completion of the hit and wait phases, T acquires the lock on the terminated list and appends itself with 1 (Line 18, Algorithm 2).

Algorithm 1: Conflict Pair Detection

```
   Data: Transaction thisTx, Operation op, List accessHistory
   Result: Boolean opSuccess
1  foreach elem ∈ accessHistory do
2      if op.type == read then
3          if elem.type == write then
4              thisTx.readPredecessors(elem.tid)
5      if op.type == write then
6          thisTx.writePredecessors(elem.tid)
7  return true
```

3.4.2 Implementation Details

We implemented the Basic Wait-Hit Protocol in our prototype in-memory database introduced in Subsection 2.3.1. We explain here how conflicts are detected and how the size of the hit and terminated lists are managed.

Conflict Detection In order to efficiently detect conflicts each row in the database stores with it a sequential history of accesses. Each access stores the associated type, read or write, and the associated transaction id. An example of the required meta-data is given in Table 3.1. This information is sufficient for transactions to correctly identify conflicts as they execute, as from it all types of conflict: wr, rw, and ww can be deduced. The corresponding conflict graph is given in Figure 3.1.

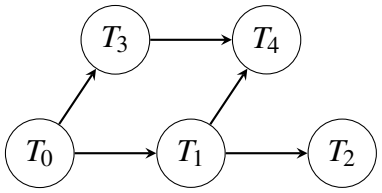


Fig. 3.1 Conflict graph representation of the access histories in Table 3.1.

Algorithm 2: Basic Wait-Hit Protocol commit procedure.

```
   Data: Transaction thisTx, List terminated, List hit
   Result: Boolean successfullyCommitted
   /* hit phase                                                    */
 1 hit.lock()
 2 if thisTx ∈ hit then
 3 |   hit.unlock()
 4 |   return !abort()                                  // hit by another txn
 5 else
 6 |   foreach pred ∈ thisTx.writePredecessors do
 7 |   |   if pred ∉ terminated then
 8 |   |   |   hit.add(pred)                            // hit predecessors
 9 |   hit.unlock()
   /* wait phase                                                   */
10 while thisTx.readPredecessors ≠ ∅ do
11 |   foreach pred ∈ thisTx.readPredecessors do
12 |   |   if pred ∈ terminated then
13 |   |   |   if terminated[pred].state == 1 then
14 |   |   |   |   thisTx.readPredecessors.remove(pred)  // read from committed txn
15 |   |   |   else
16 |   |   |   |   return !abort()                      // read from aborted txn
17 |   |   else
18 |   |   |   |   return !abort()                      // zero-wait policy
19 terminated.lock()
20 terminated.add(thisTx,1)
21 terminated.unlock()
22 return true
```

Epoch-based Garbage Collection The protocol presented in Subsection 3.4.1 requires two data structures that without space management mechanisms would grow monotonically over time. Firstly, two elements of the algorithm in Subsection 3.4.1 ensure the space requirements of the hit list are bounded. When a transaction is adding its predecessors into the hit list (after successfully validating) it uses the terminated list (TL) to merge only its predecessors that are in-flight. The merged transactions are now effectively doomed to eventually abort and when a transaction aborts, it removes itself from the hit list. Thus, the size of the hit list is bounded by the maximum number of in-flight transactions in the database. However, the same is not true of the terminated list which requires a different garbage collection mechanism.

Algorithm 3: Basic Wait-Hit Protocol abort procedure.

```
   Data: Transaction thisTx, List terminated, List hit
   Result: Boolean successfullyAborted
   /* remove from hit list                                          */
1  hit.lock()
2  if thisTx ∈ hit then
3  |  remove thisTx from hit
4  hit.unlock()
   /* add to terminated list                                        */
5  terminated.lock()
6  terminated.add(thisTx,−1)
7  terminated.unlock()
8  return true
```

Name	Age	Access History
Jack	27	$\{0,w\} \rightarrow \{1,w\} \rightarrow \{2,r\}$
Stuart	61	$\{1,w\} \rightarrow \{4,r\}$
Holly	21	$\{0,w\} \rightarrow \{3,r\} \rightarrow \{4,r\}$
Heather	56	$\{2,w\}$

Table 3.1 Example table with access history {transaction id, operation type}.

A transaction T_i can safely be removed from the terminated list when no in-flight transaction can identify T_i as a predecessor. This is true when T_i has terminated and has removed history of its access from records involved in T_i. This is ensured using an epoch-based garbage collector. Time is divided into epochs, $e = 1, 2, ...$, and a global counter E displays the current epoch. Associated with each epoch is a counter denoting the number of in-flight transactions in that epoch, $A(e)$, that is, transactions that started in that epoch but have not yet terminated.

When a transaction T_i begins it records the current value of E as its start epoch $e_{i,s}$ and increments $A(e_{i,s})$ by 1. When a transaction T_i terminates it records the current value of E as its finish epoch $e_{i,f}$ ($e_{i,s} \leq e_{i,f}$) and decrements $A(e_{i,s})$ by 1. Thus, the lifetime of a transaction can be expressed as an epoch tuple $(e_{i,s}, e_{i,f})$. Intuitively, once all transactions started during T_i's lifetime ($e_{i,s}, \leq e \leq e_{i,f}$) have terminated T_i will never be identified as a predecessor and can be removed from the terminated list.

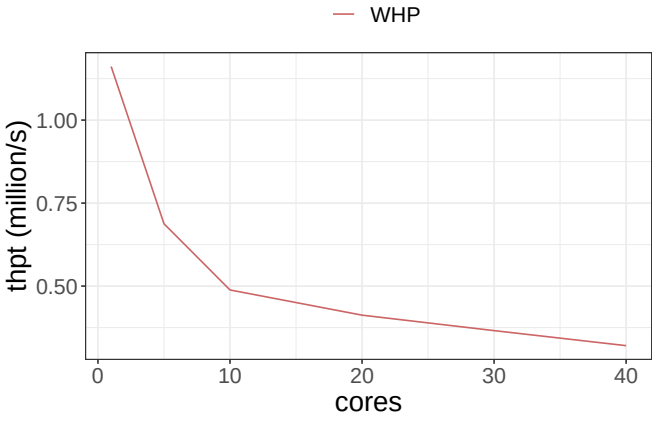
Let α be the smallest epoch number with at least 1 in-flight transaction, hence, all transactions that started with $e_s < \alpha$ have terminated: $(A(\alpha) > 0) \wedge (\forall e < \alpha : A(e) = 0)$. Note, as all transactions eventually terminate (commit or abort) $A(e)$ will eventually become 0 for all epochs. As, $T_i \in TL$ will have no in-flight transaction referring it as a predecessor if $e_{i,f} < \alpha$. Then, all $T \in TL$ with $e_f < \alpha$ can be removed from TL. In practice, periodically, the garbage collector increments the epoch, computes α, and removes all transactions with $e_f < \alpha$ from the terminated list.

3.4.3 Evaluation

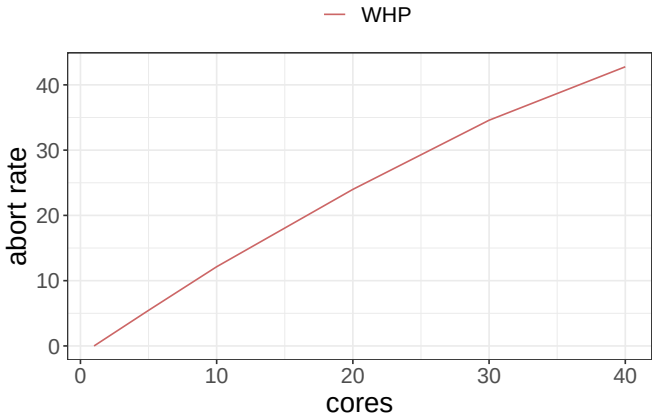
To measure the protocol's scalability the SmallBank workload with high contention described in Subsection 2.3.3 was used. Figure 3.2 displays throughput, abort rate, and average latency. It is evident from Figure 3.2 that the Basic Wait-Hit Protocol does not scale as the core count increases. In Figure 3.2(a) the throughput drops from over 1.25M transactions/s with a single-core to below 100K with 40 cores. This is reflected in the abort rate in Figure 3.2(b), across 1-40 cores it increases linearly, peaking at over 40% when the database has 40 cores. The increase in average latency in Figure 3.2(c) across the range is also near-linear.

3.4.4 Discussion

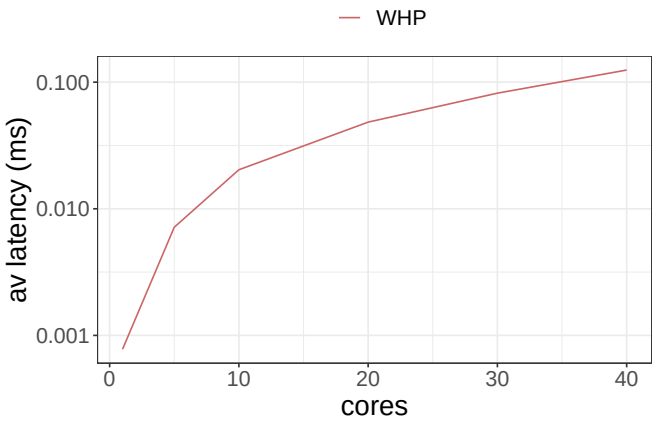
It is evident from Subsection 3.4.3 that the Basic Wait-Hit Protocol does not scale. This can be attributed to its single-threaded validation phase, that is, the hit and terminated lists require exclusive accesses; additionally it uses a centralized transaction id generator. These two elements are known causes of bottlenecks and thus performance degradation in many-core environments (see Section 3.2). However, this does not mean the underlying wait-hit approach is not compatible with scalability as we now demonstrate in Section 3.5.



(a) Throughput *vs*. cores.



(b) Abort rate *vs*. cores



(c) Average latency *vs*. cores.

Fig. 3.2 **SmallBank** – 100 customers (high contention).

3.5 Many-Core Wait-Hit Protocol

In short, the Many-Core Wait-Hit Protocol avoids global timestamp allocation and parallelizes the validation phase. We assume there are n worker threads to process transactions, $\tau = 1, 2, ..., n$. Each worker thread receives a transaction and executes it to completion before receiving additional work, thus can have at most one in-flight transaction at any moment. Each worker thread stores a local *termination list*, a list of entries for each transaction the worker thread has executed. Each entry denotes the transaction's state, in-flight: -1, aborted: 0, or committed: 1. Also, as in Section 3.4, each worker thread maintains two per-transaction data structures to track the predecessors-upon-read and predecessors-upon-write of the transaction it is currently managing.

With this configuration a centralized hit list is no longer required, as assuming that transactions can access other transactions' entries in termination lists stored on other workers, then a transaction T_i can "hit" T_j by setting T_j's state to aborted: -1. Additionally, a centralized termination list is not required as a validating transaction is concerned only with the termination statuses of *its* predecessors, which again, assuming transactions can access other transactions' entries in termination lists stored on other workers, it can directly look up as and when needed.

The key observation in scaling the wait-hit approach here is that the hit list and terminated list can be combined into a single data structure, called the *termination list*, and distributed across worker threads, with each storing only information on the transactions they are responsible for executing.

In summary, each worker thread τ needs to maintain two *thread-local* data structures:

- **Transaction id generator ($\mathbf{ID}_\tau$).** A thread-local counter, or sequence number, used in combination with the thread id to assign transactions with unique ids (`thread-id, seq-num`).

- **Termination list ($\mathbf{TL}_\tau$).** A sequence of entries for each transaction the thread has dealt with, indexed by transaction id. Each entry denotes the corresponding transaction's state (in-flight, aborted, or committed), e.g., Transaction T_i on thread τ has the entry $\mathrm{TL}_\tau(i) \in \{0, -1, 1\}$.

Also, each worker thread maintains two data structures for each transaction to track the predecessors-upon-read and predecessors-upon-write of the transaction is it currently managing. We denote the predecessors-upon-read for transaction T_i on worker thread τ as $\mathrm{PuR}_\tau(i)$ and the predecessors-upon-write as $\mathrm{PuW}_\tau(i)$.

The Many-Core Wait-Hit Protocol has two benefits over the Basic Wait-Hit Protocol. Firstly, the transaction id generation process is distributed across workers mitigating this bottleneck. Secondly, during validation, transactions only need to access the termination lists of the workers who manage its predecessors, avoiding the exclusive validation phase of the Basic Wait-Hit Protocol. Additionally, workers' termination lists do not require exclusive access through a global lock, instead individual entries can have their own §locks, which again increases parallelism.

3.5.1 Protocol Description

Prior to execution of transactions the database initializes the termination list on each worker thread τ, $\mathrm{TL}_\tau = \emptyset$ and the transaction id generator, $\mathrm{ID}_\tau = 0$. When worker thread τ receives a transaction, T, it gets a new id i from ID_τ and initializes the predecessor lists for that transaction T_i, $\mathrm{PuR}_\tau(i) = \emptyset$ and $\mathrm{PuW}_\tau(i) = \emptyset$.

Worker thread τ optimistically executes T_i, collecting predecessor transactions encountered, and recording them in $\mathrm{PuR}_\tau(i)$ or $\mathrm{PuW}_\tau(i)$ as appropriate. Conflict detection is the same as in the Basic Wait-Hit Protocol (see Algorithm 1). For each update of either $\mathrm{PuR}_\tau(i)$ or $\mathrm{PuW}_\tau(i)$ by thread τ, the worker can check the status of $\mathrm{TL}_\tau(i)$, aborting T_i if $\mathrm{TL}_\tau(i) = -1$. This reduces wasted work when transactions are doomed to abort.

Thread τ completes the optimistic execution of transaction T_i and invokes the commit procedure in Algorithm 4. The validation phase again consists of a wait and a hit phase. In the hit phase (Lines 3-9, Algorithm 4), for each predecessor the transaction has detected when performing write operations during its lifetime, it accesses the transaction's state on the corresponding worker's thread and acquires an exclusive lock on the entry. It sets the state to -1 if the predecessor is in-flight (hit). In the wait phase, for each predecessor, it accesses the transaction's state on its worker's thread and aborts itself if the predecessor state is either -1 or 0; assuming a zero-wait policy (Lines 12-19, Algorithm 4). Note, in between each phase the transaction checks if it has been hit. Upon completion of the hit and wait phases the transaction sets its state to 1 in its thread-local termination list (Line 23, Algorithm 4).

3.5.2 Optimizations

Read-only Transactions The zero-wait policy in the Many-Core Wait-Hit Protocol (Lines 16-19, Algorithm 4) can be troublesome for transactions consisting solely of read operations. It is, however, harmless for read-only transactions to wait until all predecessors-upon-read have terminated. This adjustment is permissible as a read-only transaction cannot be a constituent in a wait deadlock.

Deadlocks arise in the wait phase when transactions mutually read what each other has written, in other words, a cycle forms consisting of wr edges. Consider a read-only transaction T_i. A predecessor that exists in T_i's predecessor-upon-read list ($T_j \in \mathrm{PuR}(i)$) must be an update transaction, T_j cannot be a read-only transaction, it must have written at least one record in order for T_i to read it. Consider the case in which T_i waits for T_j which is an update transaction. T_j or no other update transaction can be waiting for T_i to commit because T_i cannot be PuR_τ of T_j, as it has not written anything. Thus, T_j cannot be involved in a wait cycle with T_i. Therefore, no update transaction a read-only transaction may wait for during

Algorithm 4: Many-Core Wait-Hit Protocol commit procedure.

```
   Data: Transaction thisTx, Thread thisThread, Set<Thread> otherThreads
   Result: Boolean successfullyCommitted
   /* hit check                                                          */
 1 if thisThread.terminated[thisTx.id] == -1 then
 2 |   return false                                  // hit by another tx
   /* hit phase                                                          */
 3 while thisTx.writePredecessors ≠ ∅ do
 4 |   foreach pred ∈ thisTx.writePredecessors do
 5 |   |   otherThread = otherThreads.get(pred.thread)
 6 |   |   pred.state = otherThread.terminated[pred.id]
 7 |   |   if pred.state == 0 then
 8 |   |   |   set pred.state = -1                  // hit predecessor
 9 |   |   thisTx.writePredecessors.remove(pred)
   /* hit check                                                          */
10 if thisThread.terminated[thisTx.id] == - 1 then
11 |   return false
   /* wait phase                                                         */
12 while thisTx.readPredecessors ≠ ∅ do
13 |   foreach pred ∈ thisTx.readPredecessors do
14 |   |   otherThread = otherThreads.get(pred.thread)
15 |   |   pred.state = otherThread.terminated[pred.id]
16 |   |   if pred.state == 1 then
17 |   |   |   thisTx.readPredecessors.remove(pred)   // read from committed tx
18 |   |   else
19 |   |   |   return !abort()          // zero-wait policy/read from aborted tx
   /* hit check                                                          */
20 if thisThread.terminated[thisTx.id] == - 1 then
21 |   return false                                  // hit by another tx
22 else
23 |   set thisThread.terminated[thisTx.id] = 1                // commit tx
24 |   return true
```

Algorithm 5: Many-Core Wait-Hit Protocol abort procedure.

```
   Data: Transaction thisTx, Thread thisThread
   Result: Boolean successfullyAborted
 1 set thisThread.terminated[thisTx.id] = -1
 2 return true
```

the wait phase can give rise to a deadlock, read-only transactions may safely wait until all predecessors have terminated.

AIMD Wait Policy This section describes an optimization to the Many-Core Wait-Hit Protocol that improves the wait-policy in the wait-phase; note, it can also be applied to both Basic Wait-Hit Protocol and Distributed Wait-Hit Protocol. Thus far the wait phase has adopted a zero-wait policy, that is, if a predecessor arising from a read operation is in-flight when a transaction is validating then the validation fails. Such a strategy achieves: (i) correctness, alleviates the possibility of reading from a transaction that subsequently aborts, and (ii) avoids deadlocks in which transactions are waiting for each other to terminate. However, this approach could lead to spurious aborts that could be avoided by a limited degree of waiting. We now describe a wait policy based on the additive increase/multiplicative decrease (AIMD) algorithm, of which the zero-wait policy is a special case. It is fashioned after a similar use in the TCP/IP congestion model. The principles of the AIMD algorithm are waiting continues for δ time (one wait cycle). At the end of a wait cycle, if a positive outcome is seen then the wait cycle duration is increased $\delta = \delta + a$; otherwise waiting is decreased $\delta = \delta \times b$. After some upper wait limit, waiting is terminated, else another wait cycle is executed.

The AIMD wait policy is given in Algorithm 6 and has four parameters: δ is the initial wait time which reflects the time period needed to pass before most predecessors have terminated, W denotes a high watermark which imposes a bound on the maximum time spent waiting, a is the additive increase which is applied each time the set of predecessors does not decrease, and b ($0 < b < 1$) is the multiplicative decrease which is applied when the set of predecessors decreases in size. If there is a deadlock, the size of the set of read predecessors will not decrease after a point and δ is guaranteed to fall below the low watermark, and the transaction will abort. Note, in the zero-wait policy: $\delta = 0$ and $a = 0$.

Algorithm 6: AIMD wait-phase

Data: Transaction thisTx, Thread thisThread, Set<Thread> otherThreads, W watermark, δ delta, a additive increase, b multiplicative decrease
Result: Boolean successfulWaitPhase

```
1  while (δ ≥ W) ∨ ( thisTx.readPredecessors ≠ ∅) do
2      S = thisTx.readPredecessors.size()
3      wait δ
4      foreach pred ∈ thisTx.readPredecessors do
5          otherThread = otherThreads.get(pred.thread)
6          pred.state = otherThread.terminated[pred.id]
7          if pred.state == 1 then
8              thisTx.readPredecessors.remove(pred)        // read from committed tx
9          else if pred.state == -1 then
10             return !abort()                              // read from aborted tx
11         else
12             continue
13     if thisTx.readPredecessors.size() < S then
14         δ = δ + a
15     else
16         δ = δ * b
17 if (thisThread.terminated[thisTx.id] == −1) ∨ (δ < W) then
18     return false
19 else
20     return true
```

3.5.3 Implementation Details

The Many-Core Wait-Hit Protocol requires the database to use a thread-per-core model, with each worker thread pinned to its respective core. As each thread can access the terminated list on other threads it is necessary such accesses are thread-safe, for example, each entry in the list is guarded by a `Mutex`. For space reasons, the terminated list must be pruned, this next section describes the epoch-based garbage collection mechanism used.

Epoch-based Garbage Collection The Many-Core Wait-Hit Protocol adapts the epoch-based garbage collector mechanism in Subsection 3.4.2. Again, time is divided into epochs numbered, $e = 1, 2, \ldots$. Let E denote the current epoch number, which the garbage collector periodically increments and asynchronously informs the threads of the new E. Thus, each

thread τ maintains its own local view of E, which is denoted as E_τ. $E_\tau \leq E$ at any time. In addition, the garbage collector maintains a list A of the number of active transactions that started in epoch e: $A(e)$ indicates the number of transactions that started in epoch e and are still known to be active. When τ starts a new transaction T_i, it records E_τ as start time $e_{i,s}$, for $T_i : e_{i,s} = E_\tau$ and increments $A(E_\tau)$ by 1. When T_i terminates, τ records the current value of E_τ as its finish epoch $e_{i,f} \rightarrow (e_{i,s} \leq e_{i,f})$ and decrements $A(e_{i,s})$ by 1.

Let α be the smallest epoch number **across all threads** with at least 1 in-flight transaction, hence, all transactions T_i that started with $e_{i,f} < \alpha$ have terminated. Therefore, such transactions will never be predecessor and can be garbage collected. The garbage collection thread periodically views the local views of α_τ and computes the global view of α by taking the minimum value of α_τ across all threads. α is then broadcast back to threads, who use it to safely remove all entries in TL_τ where $e_{\tau,f} < \alpha_\tau$. This approach is appealing as it does not require threads' views of E and α to be synchronously updated.

3.5.4 Evaluation

In this section, we experimentally compare the Many-Core Wait-Hit Protocol, referred to as MC-WHP, with the many-core SGT implementation described in Subsection 4.2.2 and with classical strict 2PL (Subsection 2.1.3) in which locks are held until commit point; the evaluation framework does not support predicate locks, thus only read/write locks are used. To be precise, we focus on ascertaining the performance implications and scalability of the Many-Core Wait-Hit Protocol's lightweight validation strategy and determining how the wait-hit approach compares to a graph-based approach. We include 2PL as it is still used in several commercial systems, e.g., [87, 92].

3.5.5 SmallBank

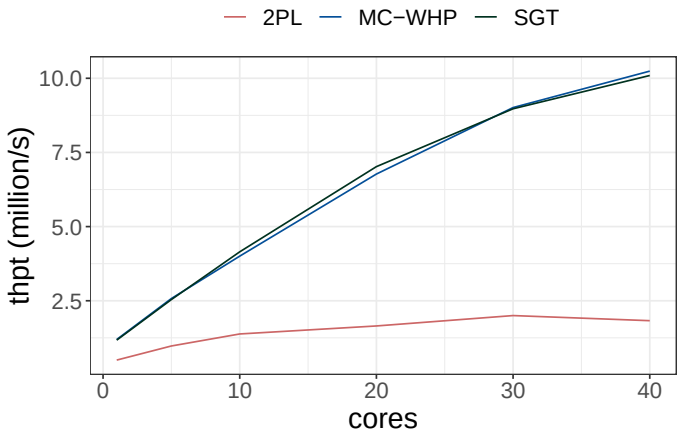
Firstly, we test the protocols using the SmallBank workload described in Subsection 2.3.3. We consider the case of high contention by setting the database to contain only 100 customers; transaction requests are uniformly distributed spread across the 100 customers. As can be seen from Figure 3.3(a), as the core count is increased MC-WHP offers comparable throughput to SGT (within 2% difference) and exceeds that of 2PL by over 4x. However, in Figure 3.3(b), MC-WHP does have around a 2x increase in aborts over SGT, but half as many as 2PL. This is anticipated as the Many-Core Wait-Hit Protocol's validation strategy is designed to sacrifice some executions in order to be lightweight and highly parallelizable. For average latency displayed in Figure 3.3(c), SGT and MC-WHP's performance is indistinguishable (within a 1% difference).

3.5.6 YCSB

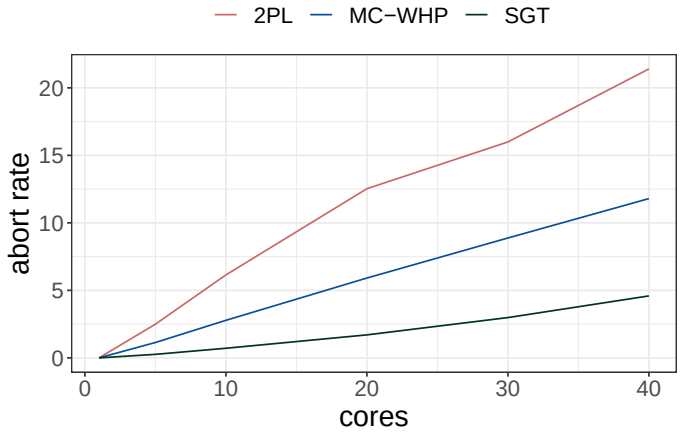
Then, we further investigate protocol performance using YCSB with the following workload factors: we opt for a medium contention level, $\theta = 0.8$, and a balanced read/update transaction mix, $U = 0.5$. The observed pattern is similar to Subsection 3.5.5. However, interestingly in Figure 3.4(a) the Many-Core Wait-Hit Protocol is able to outperform SGT at 30 cores, showing a 1.3% increase in throughput and 36% increase at 40 cores. Again, the abort rate is around 3x higher in MC-WHP compared to SGT in Figure 3.4(b), even though after 30 cores the gap shrinks to 2x. Lastly, in Figure 3.4(c), SGT has marginally lower latency up to 30 cores after which MC-WHP has lower average latency. Across all metrics, SGT and the Many-Core Wait-Hit Protocol outperform 2PL.

3.5.7 TATP

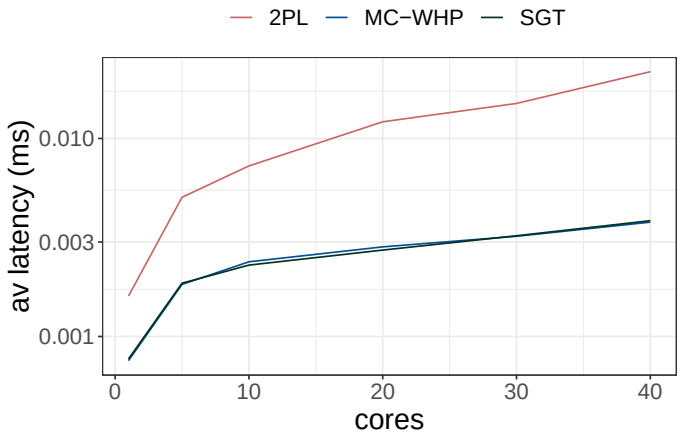
Finally, we examine the scalability of the protocols using the TATP workload. TATP is interesting as its transactions generate few conflict cycles, thus the abort rate is naturally



(a) Throughput *vs*. cores.

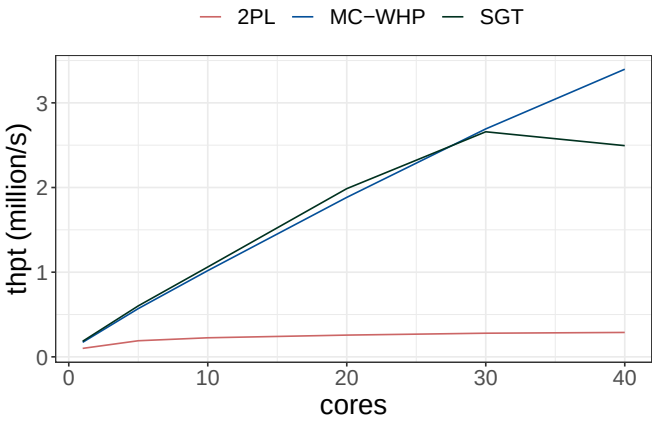


(b) Abort rate *vs*. cores

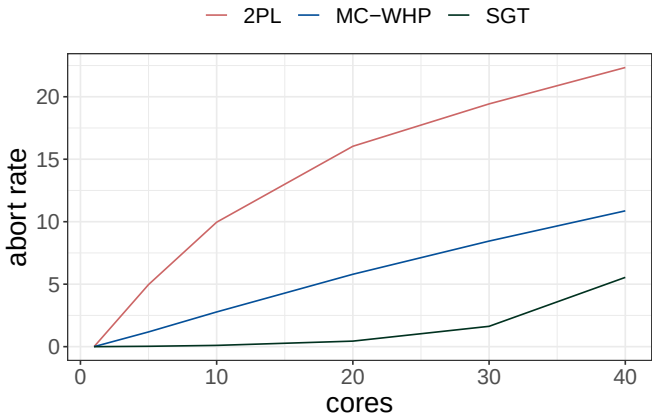


(c) Average latency *vs*. cores.

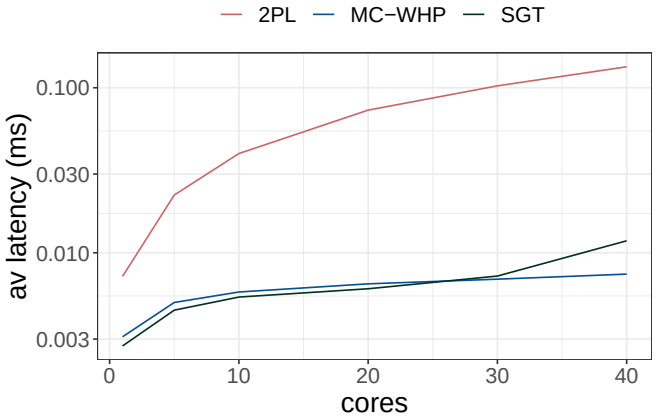
Fig. 3.3 **SmallBank** – 100 customers (high contention).



(a) Throughput *vs*. cores.



(b) Abort rate *vs*. cores



(c) Average latency *vs*. cores.

Fig. 3.4 **YCSB** – Performance measurements for the protocols as the core count is increased from 1 to 40 cores with medium contention, $\theta = 0.8$, and a balanced update rate $U = 0.5$.

lower. We anticipate that due to this the performance difference between MC-WHP and SGT will be narrow. This is what we observe in Figure 3.5, whilst SGT does outperform the Many-Core Wait-Hit Protocol across all metrics, the magnitude of difference is lower than in previous experiments.

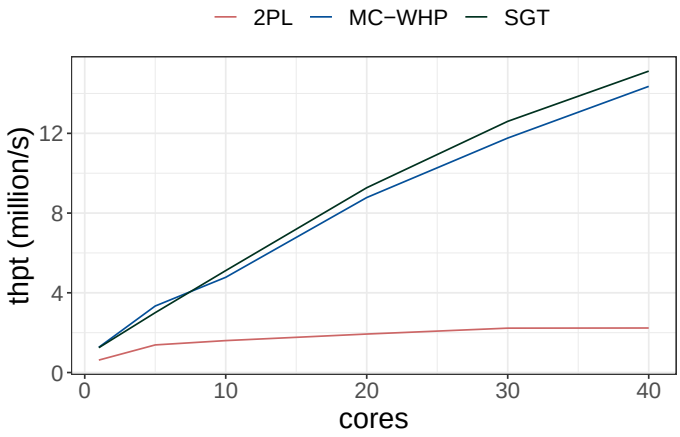
3.5.8 Discussion

All three workloads considered so far have featured a non-negligible proportion of aborts, however, in reality abortion may rarely take place in some workloads. Under these conditions the Many-Core Wait-Hit Protocol protocol with a zero-wait policy again has an advantage over its competitors.

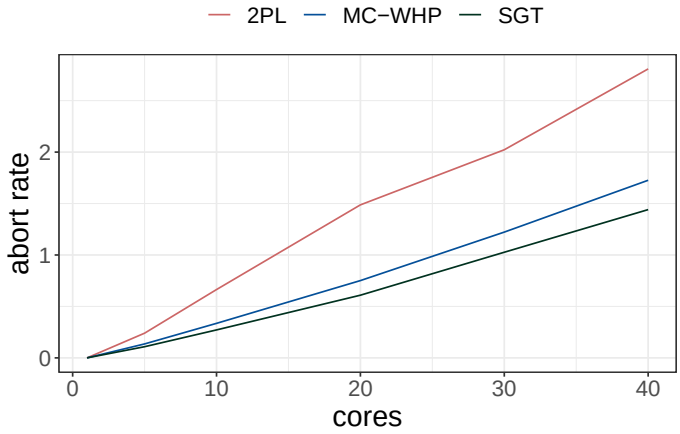
Firstly, consider the case when infrequent aborts arise from low contention in the workload. A lack of contention implies transactions are not accessing the same data concurrently, which for the Many-Core Wait-Hit Protocol translates to few predecessors being collected and therefore a quicker validation phase. Secondly, there is the case when there is high contention in the workload, but this does not translate to cycles and thus the abort rate still remains low. In this case, as stated in Subsection 3.5.2, the zero-wait policy could lead to spurious aborts, hence the AIMD wait policy could be used. This however is not a panacea, as in the case when a transaction has incorrectly waited and aborted, we have in effect increased the transaction's time spent in the system. An increased transaction lifetime means an increased contention window and therefore the overall abort rate in the system may rise as a sequence of transactions that are 'doomed' to abort needlessly wait for one another.

3.6 Distributed Wait-Hit Protocol

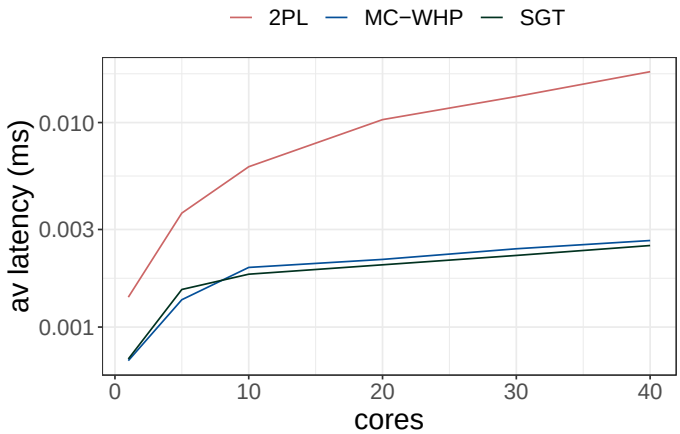
In this section, we show how the wait-hit approach can be adapted to a distributed database. The Distributed Wait-Hit Protocol assumes a shared-nothing database consisting of several



(a) Throughput *vs*. cores.



(b) Abort rate *vs*. cores



(c) Average latency *vs*. cores.

Fig. 3.5 **TATP** – 100 entries.

machines, or *shards*. Each shard is responsible for managing a disjoint partition of the database. Each multi-partition transaction, which involves data stored in multiple partitions has a *home shard*, which is the shard responsible for coordinating the transaction on behalf of the client and a set of *remote shards*. From a shard's perspective, transactions for which it is the coordinator are referred to as *local transactions*, whereas, transactions that are managed by another shard, but execute on the shard, are referred to as *foreign transactions*.

A multi-partition transaction begins at its home shard and is assigned to a *coordinator thread*. The multi-partition transaction then visits at least one remote shard, where it is managed by a *surrogate thread* on the remote shard. In summary, within each shard there is a set of threads designated for transaction management, which are further divided into two sets: (i) **coordinator threads:** responsible for handling all local transactions and coordinating with remote shards if transactions are multi-partition. (ii) **surrogate threads:** responsible for handling foreign transactions. Each thread, τ, has a number of thread-local data structures:

- **Transaction ID generator:** used to assign a transaction with a database-wide unique id. It is a combination of the shard id, thread id, and a thread-local sequence number. (Coordinator thread only).
- **Termination list ($\mathbf{TL}_\tau$).** A list of entries for each transaction the thread has dealt with, indexed by transaction id. Each entry denotes the corresponding transaction's state (in-flight: 0, aborted: -1, or committed:1). Note, if the thread is the coordinator thread then all entries will be local transactions, else it is a surrogate thread and all transactions will be foreign transactions.

Additionally, as in Sections 3.4 and 3.5 each worker maintains two per-transaction data structures to track the predecessors of the transaction (local or foreign) it is currently managing: predecessors-upon-read and predecessors-upon-write.

3.6.1 Protocol Description

The Distributed Wait-Hit Protocol consists of three stages: initialization, execution, and commitment. The commitment stage is broken into two phases: wait and agreement. This section describes each stage from the perspective of the coordinator thread and the surrogate threads. The normal execution flow of the protocol is provided in Subsection 3.6.3, orange circles link to key points in the execution flow. Messages are introduced as and when needed; a summary of messages sent in the protocol is given in Subsection 3.6.3.

Coordinator Thread

Initialization A shard in the database receives a transaction request from a client. The shard then allocates the transaction to a thread. This thread is the coordinator thread for the transaction and it performs three actions: (i) assigns the transaction with an id using its ID generator, (ii) appends the transaction to its termination list, and (iii) creates transaction-local predecessor upon read/writes lists, (1).

Execution The coordinator thread optimistically executes the operations within the transaction. If read/write predecessors are encountered while accessing local data, they are recorded into the appropriate predecessor list. An important point to note is that if the operation is a write operation and predecessors are detected then they are *proactively hit* and their state set to aborted once the operation has completed. Whilst this may induce more aborts it simplifies atomic commitment (details to follow in Subsection 3.6.2). If the transaction requires remote data on another shard, the coordinator forwards the operation to the relevant shard using a **REMOTE-OP** message, (2). Note, the remote shard will return the result of the **REMOTE-OP** request; if the remote operation is a write operation then any predecessors-upon-write detected on the remote shard will also be proactively hit on the remote shard. This process repeats until the completion of all operations within a transaction and it is ready to commit at

which point the coordinator sends a **VALIDATE** message to participant shards (including itself) contacted during the transaction's execution in order to begin the commit procedure, (3).

Commit Due to proactive hitting when the commit procedure is called all of a transaction's predecessors-upon-write will have been aborted globally. Thus, the commit procedure contains a single validation phase: wait. In the wait phase the coordinator iterates through its predecessor-upon-read list, if the predecessor is committed it is removed from the list, else the predecessor has aborted or is in-flight, thus the coordinator must abort the transaction. Additionally, concurrent with this procedure the coordinator is listening for **FAILED** messages from remote shards; such a message is sent by surrogate threads on remote shards to indicate the transaction has failed validation locally. If the coordinator fails the wait phase locally or receives such a message then it terminates the transaction, informs the client, and sends **ABORT** to all surrogate threads. Else, the wait phase is completed and the coordinator delays until it has received a **VERIFIED** message from all remote shards at which point all surrogates have completed the wait phase and the transaction has globally been validated. The coordinator now enters the agreement phase and sets the transaction state to committed in its termination list. *This is the point of no return, once here the transaction must and will always commit.*

In the agreement phase the coordinator simply sends **COMMIT**, (5), to all remote shards and waits until it has received **COMMITTED** from all participants before responding to the client, (7).

Surrogate Thread

Initialization When a shard receives a **REMOTE-OP** it allocates the request to a surrogate thread. The request will include the transaction id assigned by the coordinator thread. It adds the **REMOTE-OP**'s corresponding transaction to its termination list; indexed by the

corresponding transaction ID (assigned by the coordinator thread). The surrogate thread then creates transaction-local predecessor upon read/writes lists, (2). Note, these lists only store predecessors detected from operations executed for this transaction on this remote shard.

Execution The surrogate thread optimistically executes the operation and any subsequent operations for this transaction, detecting predecessors and installing them into the appropriate predecessor list. If the remote operation is a write operation then any predecessors-upon-write detected on the remote shard are proactively hit. The surrogate thread then delays until it receives **VALIDATE** from the coordinator thread and then enters the commit procedure, (4). Note, at any time during execution if an **ABORT** message is received the surrogate thread aborts the transaction.

Commit In the commit procedure, the surrogate performs the wait procedure, iterating through its predecessor-upon-read list: if the predecessor is committed it is removed from the list, else the predecessor has aborted or is in-flight, thus the surrogate thread must abort the transaction and sends a **FAILED** message to the home shard. If the wait phase is successful then a **VERIFIED** message is sent to the coordinator and the surrogate waits for a **COMMIT** message before proceeding to the agreement phase. In the agreement phase, the surrogate simply sets the transaction state to committed and replies with **COMMITTED** to the coordinator, (6).

3.6.2 Discussion

To summarize, the Distributed Wait-Hit Protocol follows a similar pattern to other optimistic concurrency control protocols in that it integrates the validation phase with the first round of 2PC: the validation phase maps to the prepare phase of 2PC. In the Distributed Wait-Hit Protocol this is enabled by proactively hitting any predecessors-upon-write detected at the end of each write operation performed. Without proactive hitting an additional phase (round

trip) in the commit procedure would be needed to ensure that all predecessors-upon-write are globally hit across all participant shards. The additional latency incurred per-transaction for another round trip would potentially be unpalatable for most OLTP applications.

The benefit of Distributed Wait-Hit Protocol is that validation at each shard (remote and home) can proceed independently in parallel, a clear advantage over distributed SGT which requires distributed cycle check and thus a high-level of inter-shard communication. The validation phase also possesses the same benefits as that in the Many-Core Wait-Hit Protocol in that it is extremely lightweight, compared to OCC, the Distributed Wait-Hit Protocol does not have the overheads of copy items during transaction execution. Compared to MVCC and TO which block newer transactions that conflict until the older ones commit, the Distributed Wait-Hit Protocol effectively allows transactions to race, thus should increase throughput in situations when data items are highly contended, but the overall abort rate is low.

In Subsection 3.5.8 the performance of the Many-Core Wait-Hit Protocol under a workload with infrequent aborts was discussed. We now discuss the possible implications of such a workload for the Distributed Wait-Hit Protocol. When the abort rate is low due to low contention the performance of the protocol is bound by the network speed for performing validation. In the case there is contention (thus predecessors lists are non-empty) and a non-zero wait policy is employed, the drawbacks of incorrectly waiting are more pronounced than in the Many-Core Wait-Hit Protocol as the lifetime of a transaction is already higher due to the network calls required for validation, so the contention window of transactions would increase. This could lead to cascading aborts, degrading system performance.

3.6.3 Messages

This section summarizes the messages exchanged between servers in the execution of a distributed transaction.

- **REMOTE-OP:** issued by a coordinator thread to a remote shard in order to access data on the remote server. The acknowledgement to a **REMOTE-OP** can include the results to read operations if required.
- **VALIDATE:** issued by a coordinator thread to indicate its transaction has completed execution and wishes to be committed. Sent to all surrogate threads to initiate their validation phase.
- **VERIFIED:** sent by surrogate threads to the coordinator thread to indicate they have completed the wait phase and locally successfully validated.
- **FAILED:** sent by surrogate threads to the coordinator thread to indicate it has failed validation locally.
- **COMMIT:** issued by the coordinator thread to surrogate threads if they have all responded with **VERIFIED** (all completed validation).
- **ABORT:** issued by a coordinator thread to surrogate threads if at least one surrogate thread responded with **FAILED**.
- **COMMITTED:** sent by surrogate threads to the coordinator thread to indicate it has completed the agreement phase and successfully committed.

3.7 Conclusion

In this chapter we presented Wait-Hit Protocol, an optimistic concurrency control protocol that is designed to scale vertically, as the core count is increased, and horizontally, as database nodes are added. This is achieved through a lightweight validation phase that is highly parallelizable. We have experimentally demonstrated that the wait-hit approach can offer comparable performance with contemporary high performance protocols in a many-core

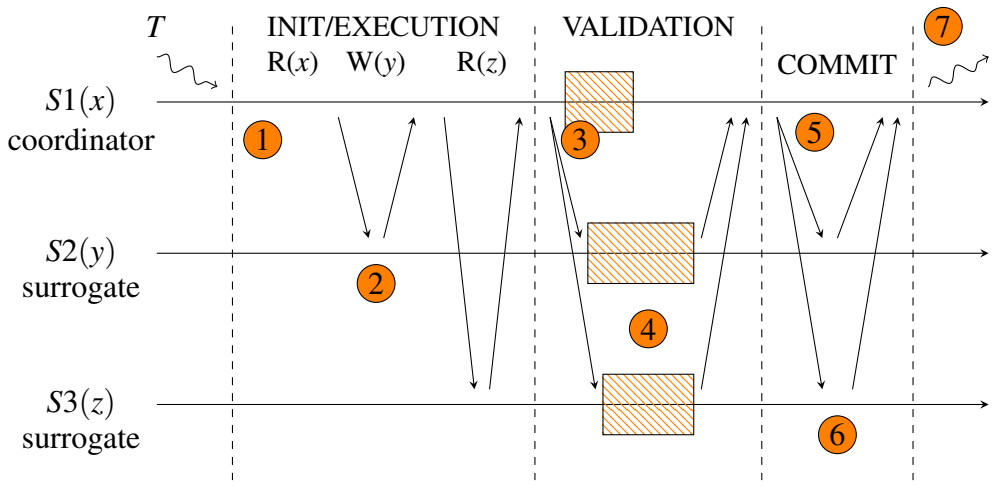


Fig. 3.6 Distributed wait-hit protocol messages exchanged during happy path execution. Orange circles link to the textual description of the protocol. Note, the wait phase is indicated by the orange hashed area in the validation phase.

environment at the cost of some additional aborts. The validation phase is designed in a manner that addresses previous shortcomings of optimistic protocols in distributed databases.

3.7.1 Further Work

Additional Experiments As part of future work we wish to extend our evaluation framework to allow for the implementation of several classical and state-of-the-art distributed concurrency control protocols. Doing so will allow us empirically to compare the performance of Distributed Wait-Hit Protocol with other protocols such as distributed OCC, MVCC, and SGT, enabling us to validate our design and gain empirical support for the claims made in Subsection 3.6.2.

Prior work on evaluating distributed concurrency control protocols focused on performance in a single data center [54]. Future work could expand to large-scale distributed database systems that span multiple data centers and geographic regions. In such a setting the communication overhead imposed by optimistic methods (such as the Distributed Wait-Hit Protocol) for validation may increase latency and reduce system throughput to unpalatable levels. This challenge presents an opportunity to explore incorporating several mitigation

strategies into the protocol. One such strategy would be aggregating multiple transactions into batches to amortize the network overheads over many transactions, this could increase the latency of individual, but maintain a stable average latency making for an improved user experience. Another approach would be to exploit advances in modern hardware and use technologies such as programmable switches [16] to design optimistic protocols that interact better with the underlying hardware to maximize performance.

Integration with Machine Learning Another fertile area for exploration is the impact machine learning (ML) can have on improving the performance of concurrency control protocols in database systems. Some initial work has focused on making improved scheduling decisions in many-core main memory database systems [95], it would be interesting to investigate whether similar techniques could be applied in a distributed context in combination with the wait-hit approach. Additionally, other recent work [112] has leveraged ML to develop a learning-based framework that adapts the concurrency control protocol to the workload, future work could determine if a similar approach could be used to optimize the variants of the Wait-Hit Protocol to the workload *and* the scale point it is targeting.

Fault Tolerance and Availability For fault tolerance and availability, shards within a distributed database are normally replicated. Balancing fault tolerance and availability with transactional performance and system consistency is a delicate act. One approach future work could explore is how the wait-hit approach can be integrated with strong replication strategies such as Raft [82] and Paxos [68]. However, using these protocols can lead to increased latency due to the extra communication between replicas to guarantee consensus, also reducing throughput. On the other hand, weaker replication strategies offer improved performance at the cost of weak consistency guarantees in the face of component failures and network partitions, which may not match up with user expectations.

Switching The motivation for developing the family of optimistic concurrency controls in this Chapter was to have a single protocol that can scale from a many-core environment to a large-scale deployment within a data center or across multiple data centers. Protocols have been presented for each of these scale points, but missing from discussions was how to facilitate the transition between them. Therefore, an open and interesting avenue for future work is to investigate how to dynamically switch from the Many-Core Wait-Hit Protocol to the Distributed Wait-Hit Protocol as resources are added and removed. This will help satisfy users' expectations of seamless elastic database scaling as hardware resources are altered.

The goal to strive for when switching from the Many-Core Wait-Hit Protocol and the Distributed Wait-Hit Protocol from the client perspective is for there to be no performance drop, for there to be no read or write unavailability to the system, and for the transition to be tolerant to faults. One key issue to be addressed is whether the single-node running the Many-Core Wait-Hit Protocol would become part of the distributed system running the Distributed Wait-Hit Protocol or not. Such an in-place transition would be resource efficient, but the bootstrap time to form the distributed cluster may impact performance. This could be particularly pronounced if the cluster is running at near peak load as the underlying thread pool would be reduced due to division between coordinator threads and surrogate threads for the Distributed Wait-Hit Protocol.

An alternative approach would be to start a shadow cluster, avoiding the need for a bootstrap phase, and use a database proxy to mirror traffic to the system running the Distributed Wait-Hit Protocol- keeping them in sync. This approach would be less resource efficient compared to an in-place transition, but this may be tolerable in a cloud environment as the server running the Many-Core Wait-Hit Protocol can be decommissioned as soon as the switch has been made. Running a hot shadow cluster would also possibly provide more flexibility around data sharding. Before the transition the shadow cluster can move data between shards without the concern of impacting throughput or latency, whereas, with the

in-place switch the initial cluster running the Many-Core Wait-Hit Protocol would have a imbalanced data partitioning which could yield unstable performance.

Chapter 4

Mixed Serialization Graph Testing

Summary

Serialization Graph Testing faithfully implements the conflict graph theorem described in Subsection 2.1.1 by aborting only those transactions that would actually violate serializability (introduce a cycle), thus maintaining the required acyclic invariant. All other known approaches, such as 2PL and the Wait-Hit Protocol, disallow certain valid schedules to increase throughput. Thereby, Serialization Graph Testing has the theoretically optimal property of accepting all and only conflict serializable schedules. However, as discussed in Subsection 2.1.2, not all applications require conflict serializability, but can perfectly settle for various, known weaker isolation levels which typically require relatively lower overheads; we corroborate this with a survey of the isolation levels supported by 24 databases in Section 4.3. In such a mixed environment, providing only the isolation level required of each transaction should, in theory, increase throughput and reduce aborts. This chapter extends Serialization Graph Testing for mixed environments subject to Adya's mixing-correct theorem and confirms improvement in performance. **Mixed** Serialization Graph Testing can achieve up to a 28% increase in throughput and a 19% decrease in aborts over Serialization Graph Testing.

4.1 Introduction

Compared to the other approaches described in Subsection 2.1.3, the graph-based approach[1] to serializable transaction processing possesses the desirable theoretical property of accepting all conflict serializable schedules. Other approaches approximate the complete space of conflict serializable schedules, but in doing so sacrifice a degree of concurrency in an attempt to achieve higher throughput.

This can be illustrated by the following example. Consider two transactions, T_1 and T_2, execute in a database running 2PL. Both transactions concurrently attempt to access

[1] Also referred to as *Serialization Graph Testing (SGT)*

a single data item, x; assume T_1 and T_2 access no other items and that there are no other transactions executing in the system. T_1 wishes to write x and T_2 wishes to read x. Let us say T_1 accesses x first and thus acquires a write lock on x. When T_2 attempts to acquire a read lock on x, depending on the deadlock detection strategy used, either T_1 or T_2 will be aborted. This discounts a valid conflict serializable schedule, as no interleaving of T_1 and T_2 could introduce a cycle in the conflict graph (either $T_1 \rightarrow T_2$, or $T_2 \rightarrow T_1$).

Despite Serialization Graph Testing's theoretical upper-hand, it was historically discounted as a viable strategy because the associated computational costs of maintaining an acyclic graph were deemed impractical. However, recent research has refuted this perceived wisdom. In [34] it was demonstrated how SGT can be implemented efficiently in a many-core database, offering comparable, and often higher, performance when compared to traditional and contemporary concurrency control protocols.

In spite of recent advances, serializable transaction processing performance often remains unsatisfactory for application demands. As introduced in Subsection 2.1.2, another tool at databases' disposal to increase performance is to execute transactions at *weak isolation* levels, e.g., Read Committed. Here, the number of permissible schedules is increased at the expense of potentially allowing non-serializable behaviour, e.g., Fuzzy Reads. Weak isolation is pervasive in real world systems, with most systems offering a range of isolation levels; a comprehensive survey of the isolation levels supported by commercial and open source databases is given in Table 4.1. A database that allows concurrent transactions to be executed at different isolation levels is said to be *mixed* [1]. For example, transaction T_{RC} can run at Read Committed and transaction T_S at Serializable.

The classical method to implement a mixed DBMS is to opt for a 2PL-variant in which transactions vary the duration they hold locks [51]. For example, T_{RC} would release read locks on data items immediately after performing the read operation, whereas T_S holds all locks until commit time. This mechanism and others used in mixed DBMSs suffer from

the same problem as their serializable equivalents: some valid executions are prevented leading to unnecessary aborts. This begs the question: how can SGT be extended to allow some transactions to be executed at weak isolation levels, whilst accepting all and only valid executions? Such an approach would permit higher concurrency and performance.

This chapter presents *Mixed Serialization Graph Testing* (MSGT), which accepts all valid schedules under Adya's *mixing-correct theorem* [1], thus maintaining SGT's property of minimizing aborts. We evaluate MSGT's performance using a range of popular OLTP benchmarks.

The remainder of this chapter is structured as follows: Section 4.2 provides a detailed overview of SGT and the many-core implementation presented in [34]. Section 4.3 provides a survey of the isolation levels provided by ACID and NewSQL [86] databases highlighting the prevalence of mixed DBMSs and demonstrating the utility of MSGT. Section 4.4 describes Adya's formalism of weak and mixed isolation levels. Section 4.5 presents Mixed Serialization Graph Testing. Section 4.6 gives results, before Section 4.7 concludes.

4.2 Serialization Graph Testing

This section presents Serialization Graph Testing. Subsection 4.2.1 provides a description of the protocol. Subsection 4.2.2 describes the many-core optimized concurrent graph data structure used by the SGT implementation in [34]. Where appropriate we draw comparisons with the Wait-Hit Protocol in Chapter 3.

4.2.1 Protocol Description

In this chapter we focus on basic-SGT [12], which operates as follows:

1. When the database scheduler receives transaction T_i it creates a node in its conflict graph for T_i.

2. Then, for each operation on item x, $op_i(x)$, within T_i,

   (a) Conflicts are determined. $op_i(x)$ conflicts with $op_j(x)$ of T_j if $op_j(x) < op_i(x)$, and if $op_i(x)$ is a read and $op_j(x)$ is a write (wr conflict), or if $op_i(x)$ is a write and $op_j(x)$ is a read or write (rw/ww conflicts).

   (b) For each conflict, an edge is inserted into the graph from the conflicting transaction, $T_j \rightarrow T_i$; provided an edge $T_j \rightarrow T_i$ does not already exist. Note, transactions only insert *incoming* edges to themselves.

   (c) Then, a cycle check is performed *before* executing the operation. If executing the operation would introduce a cycle T_i is aborted and its *outgoing* edges removed.

3. At commit time, a T_i delays until it has no incoming edges and then removes its *outgoing* edges. The removal of outgoing edges ensures conflicting transactions will commit. Note, if T_i has no incoming edges then it cannot be involved in a cycle.

Delaying the commitment of transactions until they have no incoming edges achieves: (i) safe node deletion, (ii) *recoverability*, and (iii) *order preservation*. To illustrate this consider schedule s, which contains transactions T_1, T_2, T_3, and T_4. Recall, a write on item x by transaction T_i is denoted by $w_i[x]$, a read by $r_i[x]$, and a commit operation by c_i. The corresponding conflict graph $CG(s)$ is shown in Figure 4.1.

$$s = w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]\ w_2[z]\ w_3[z]\ r_3[x]\ r_3[a]\ w_4[a]\ c_1\ c_3\ c_2\ c_4$$

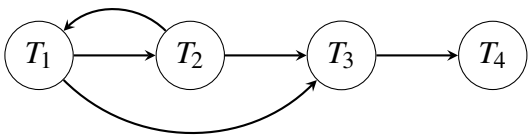


Fig. 4.1 Conflict graph representation of s.

Due to space constraints nodes must be pruned from the conflict graph. Allowing transactions to commit with incoming edges can lead to subtle serialization violations.

For example, assume in Figure 4.1, T_3 has committed and T_2 is still executing. If T_3 is removed upon commitment, T_2 may subsequently perform an operation that introduces a cycle with T_3, but which goes undetected due to T_3's removal, introducing a serialization error. For a node to be involved in a cycle it must have at least one incoming and one outgoing edge. As transactions only insert edges incoming to themselves during execution, after issuing a commit request, no more incoming edges are added. Thus, once all incoming edges are removed from the committing node, via the parent node terminating, it will not be in a cycle.

Recoverability ensures failures do not leave the database in an inconsistent state. In s, T_1 writes x, then T_2 reads x. If T_2 commits before T_1, T_1 may subsequently abort, at which point T_2 has read from a value that never existed. Delaying T_2's commit until it has no incoming edges prevents this issue; if T_1 aborts then T_2 is also aborted.

Order preservation ensures the real-time commit order matches the serialization order. In s, T_4 overwrites the value a read by T_3, thus in the serial order: $T_3 \rightarrow T_4$. If T_4 commits before T_3 no recoverability issues are introduced, but the real-time commit order does not match the serialization order. Delaying T_4's commit until it has no incoming edges avoids this, providing users with an improved, more intuitive, experience.

Note, the Wait-Hit Protocol in Chapter 3 bears resemblance to SGT in that for each operation within a transaction conflicts are detected, but rather than converting this information into edges in a conflict graph, the Wait-Hit Protocol compresses conflict information into predecessor-upon-read and predecessor-upon-write sets. Thus, exact cycle detection in Wait-Hit Protocol is not possible and some schedules are unnecessarily aborted. However, the Wait-Hit Protocol's compressed conflict information allows the protocol's validation to scale horizontally in a partitioned database, whereas in SGT's cycle checking across

partitions would incur significant network overheads for each conflict detected. In summary, Wait-Hit Protocol is trading off some unnecessary aborts for scalability.

4.2.2 Many-Core Implementation

SGT was efficiently implemented in [34] using a graph data structure designed to facilitate concurrent cycle checking and avoid the pitfalls of other many-core concurrency control protocols.

As stated in Subsection 2.1.4, many-core DBMSs use optimistic approaches as their pessimistic counterparts suffer from poor performance as the core count increases. A common bottleneck in optimistic approaches is an exclusive single-threaded verification phase. Regarding SGT, in [34], to avoid a global lock for graph operations, the graph data structure utilizes a node-local locking protocol. Nodes in the graph each store a transaction status (committed, active, aborted) and two sets of pointers representing incoming and outgoing edges. Nodes can then be locked in two modes:

- ***Shared mode:*** transactions can concurrently access the node for edge insertions, edge deletions, and cycle checking. Edge sets guarantee thread-safe concurrent access for scans, insertions, and deletes under the shared lock.

- ***Exclusive mode:*** used for the commit-critical check for incoming edges.

The node-level locking protocol works as follows: when a transaction T_x identifies a conflict with transaction T_y it first checks if an edge already exists from T_y to T_x, if so, no additional work is needed. Else, T_x acquires a shared lock on T_y's node. If T_y is active, then an edge pointer to T_x is inserted into T_y's outgoing edge set and an edge pointer from T_y is inserted into T_x's incoming edge set. T_x then checks for a cycle using a *reduced depth-first search* (DFS) algorithm. Reduced DFS begins at the validating node (T_x) and traverses only the portion of the graph that is needed; each step holds nodes in shared lock mode. At commit

time, T_x acquires an exclusive lock on its node and checks for incoming edges. If there are none, T_x commits and shared locks are acquired on each node in T_x's outgoing edge set and the edge from T_x removed. Else, there exists at least one incoming edge and the exclusive lock is released and the check is repeated until either the transaction commits or is aborted. Note, the exclusive lock is needed to prevent other transactions traversing T_x during their cycle checking or from adding in outgoing edges from T_x.

Another common bottleneck is the reliance on a global timestamp allocator, this is avoided in [34] by letting conflict graph nodes double up as transaction ids.

SGT requires a mechanism to derive conflicts. In [34] database rows store a sequential history of accesses. Each access stores the operation type, read or write, and the transaction id. Decoupling the access information from the graph data structure decreases contention. Note, sequentially ordered access is ensured by per-row spin-locks which are released immediately after the operation completes, i.e., the lock is not held until commit time, an improvement over lock-based approaches.

In summary, the SGT implementation in [34] decouples conflict detection from the management of the conflict graph, and employs graph structure that allows concurrent cycle checking. In particular, the commit critical check only shortly blocks other threads from accessing a node and only the part of the serialization graph needed for validation is traversed. Lastly, nodes in the graph double up as transaction ids, avoiding a well established bottleneck in other protocols. It achieves this whilst also minimizing the number of unnecessary aborts, accepting all conflict serializable schedules and provides an ideal baseline for the development of MSGT.

4.3 Mixing in the Wild

This section motivates the development of a mixed graph-based scheduler that minimizes unnecessary aborts by surveying the isolation levels supported by commercial and open source DBMSs.

It is rare that practical DBMSs offer applications only a singular isolation, instead permitting transactions to be run at different isolation levels. In order to assess this claim we surveyed the isolation levels offered by 24 ACID and NewSQL database systems in Table 4.1. Classification was performed based on each database's public documentation. We found seven isolation levels represented: Read Uncommitted (RU), Read Committed (RC), Cursor Stability (CS), Snapshot Isolation (SI), Consistent Read (CR), Repeatable Read (RR), and Serializable (S). Note, the exact behaviour of each isolation level is highly system-dependent. Interestingly, we found 18 databases supported multiple isolation levels. Of systems offering a single isolation level Serializable was the most common; these systems were typically NewSQL [86] systems, e.g., CockroachDB [98]. This may suggest a trend away from mixed databases, however, TiDB [55, 105] recently added support for Consistent Read isolation indicating the utility of weaker isolation remains.

Our survey's findings are corroborated by a 2017 survey of database administrators on how applications use databases [85], the survey found the majority of transactions execute at Read Committed. In short, this evidence illustrates the ubiquity of mixed databases and motivates the development of MSGT.

4.4 Mixing Theory

This section presents the correctness criteria utilized by MSGT. Subsection 4.4.1 reproduces the system model from [1], which is used to define weak isolation levels in Subsection 4.4.2, before the mixing-correct theorem is defined.

Table 4.1 Isolation Levels Supported by ACID and NewSQL Databases

Database System	**Isolation Level**						
	RU	***RC***	***CS***	***SI***	***CR***	***RR***	***S***
Actian Ingres 11.0	✓	✓	✓	✗	✗	✓	✓*
Clustrix 5.2	✗	✓[e]	✗	✗	✗	✓*[c]	✓
CockroachDB 20.1.5	✗	✗	✗	✗	✗	✗	✓*
Google Spanner	✗	✗	✗	✗	✗	✗	✓*
Greenplum 6.8	✓[b]	✓*	✗	✗	✗	✓	✗
Dgraph 20.07	✗	✗	✗	✓*	✗	✗	✗
FaunaDB 2.12	✗	✗	✗	✓	✗	✗	✓*
Hyper	✗	✗	✗	✗	✗	✗	✓
IBM Db2 for z/OS 12.0	✓	✓[a]	✓*	✗	✗	✓	✗
MySQL 8.0	✓	✓	✗	✗	✗	✓*	✓
MemGraph 1.0	✗	✗	✗	✓*	✗	✗	✗
MemSQL 7.1	✗	✓*[e]	✗	✗	✗	✗	✗
MS SQL Server 2019	✓	✓*	✗	✓	✗	✓	✓
Neo4j 4.1	✗	✓*	✗	✗	✗	✗	✓
NuoDB 4.1	✗	✓	✗	✗	✓*	✗	✗
Oracle 11g 11.2	✗	✓*	✗	✓	✗	✗	✗
Oracle BerkeleyDB	✓	✓	✓	✓	✗	✗	✓
Oracle BerkeleyDB JE	✓	✓	✗	✗	✗	✓*	✓
Postgres 12.4	✓[b]	✓*	✗	✗	✗	✓[c]	✓
SAP HANA	✗	✓*	✗	✓	✗	✗	✗
SQLite 3.33	✓	✗	✗	✗	✗	✗	✓*
TiDB 4.0	✗	✗	✗	✓*	✓	✗	✗
VoltDB 10.0	✗	✗	✗	✗	✗	✗	✓*
YugaByteDB 2.2.2	✗	✗	✗	✓*	✗	✗	✓

* Indicates the default setting.
[a] Referred to as ***Read Stability***.
[b] Behaves like ***Read Committed*** due to MVCC implementation.
[c] Implemented as ***Snapshot Isolation***.
[d] Requires manual lock management.
[e] Behaves like ***Consistent Read***.

4.4.1 System Model

In Adya's system model, transactions consist of an ordered sequence of read and write operations to an arbitrary set of data items, book-ended by a `BEGIN` operation and a `COMMIT` or an `ABORT` operation. The set of items a transaction reads from and writes to is termed its *item read set* and *item write set*. Each write creates a *version* of an item, which is assigned a unique timestamp taken from a totally ordered set (e.g., natural numbers) version i of item x is denoted x_i; hence, a multiversioned system is assumed. All data items have an initial *unborn* version $\perp$ produced by an initial transaction $T_\perp$. The unborn version is located at the start of each item's version order. An execution of transactions on a database is represented by a *history*, H. This consists of a *partial order of events*, which reflects (i) each transaction's read and write operations, (ii) data item versions read and written and (iii) commit or abort operations, and a *version order*, which imposes a total order on committed data item versions.

There are three types of dependencies between transactions, which capture the ways in which transactions can *directly* conflict. *Read dependencies* capture the scenario where a transaction reads another transaction's write. *Antidependencies* capture the scenario where a transaction overwrites the version another transaction reads, they can be seen as the opposite of read dependencies. *Write dependencies* capture the scenario where a transaction overwrites the version another transaction writes. Their definitions are as follows:

Read-Depends Transaction T_j *directly read-depends* (wr) on T_i if T_i writes some version x_k and T_j reads x_k.

Anti-Depends Transaction T_j *directly anti-depends* (rw) on T_i if T_i reads some version x_k and T_j writes x's next version after x_k in the version order.

Write-Depends Transaction T_j *directly write-depends* (ww) on T_i if T_i writes some version x_k and T_j writes x's next version after x_k in the version order.

Using these definitions, from a history H a *direct serialization graph DSG*(H) is constructed. Each node in the *DSG* corresponds to a committed transaction and edges correspond to the types of direct conflicts between transactions. Anomalies can then be defined by stating properties about the *DSG*. To illustrate the difference between an Adya history and a classic schedule, s is given with versions accessed by each operation (version order: $[x_0 \ll x_1, y_0 \ll y_1, z_2 \ll z_3]$). This is visualized in Figure 4.2.

$$H = w_1[x_1]\ r_2[x_1]\ r_2[y_0]\ w_1[y_1]\ w_2[z_2]\ w_3[z_3]\ r_3[x_1]\ c_1\ c_3\ c_2$$

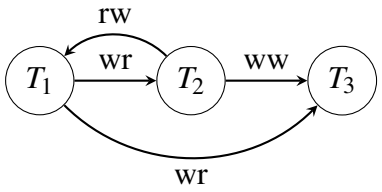


Fig. 4.2 Direct serialization graph (DSG) representation of H.

The above *item-based* model can be extended to handle *predicate-based* operations [1]. Database operations are frequently performed on sets of items provided a certain condition called the *predicate*, P, holds. When a transaction executes a read or write based on a predicate P, the database selects a version for each item to which P applies, this is called the version set of the predicate-based denoted as $Vset(P)$. A transaction T_j changes the matches of a predicate-based read $r_i(P_i)$ if T_i overwrites a version in $Vset(P_i)$.

4.4.2 Weak Isolation Levels

Using the system model in Subsection 4.4.1, definitions of isolation levels are given via a combination of constraints and the prevention of types of cycles in the *DSG*. In total 11 isolation levels are presented in [1]. In this chapter we consider a subset: Read Uncommitted, Read Committed, Repeatable Read, and Serializable.

Read Uncommitted Read Uncommitted isolation prevents anomaly Dirty Write (G0), that is, the *DSG* cannot contain cycles consisting entirely of write-depends edges. H_{G0} below contains a Dirty Write (G0) anomaly and the corresponding *DSG* is shown in Figure 4.3. Informally, this means writes to different data items must be ordered consistently across transactions.

$$H_{G0} = w_1[x_1]\ w_2[y_2]\ w_1[y_1]\ w_1[x_1]\ c_1\ c_2$$

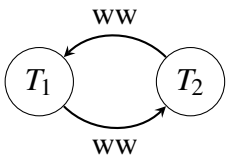


Fig. 4.3 DSG(H_{G0}) displays a Dirty Write (G0) anomaly.

Read Committed Read Committed isolation prevents G0 and anomalies, (i) Aborted Read (G1a), transactions cannot read data item versions created by aborted transactions (H_{G1a}), (ii) Intermediate Reads (G1b), transactions cannot read intermediate data item versions (H_{G1b}), and (iii) Circular Information Flow (G1c), the *DSG* cannot contain cycles consisting of write-depends and read-depends edges (H_{G1c}, DSG in Figure 4.4). Informally, this means transactions only read committed information.

$$H_{G1a} = w_1[x_1]\ r_2[x_1]\ a_1\ c_2$$

$$H_{G1b} = w_1[x_1]\ w_1[x_2]\ r_2[x_1]\ c_1\ c_2$$

$$H_{G1c} = w_1[x_1]\ r_2[x_1]\ w_2[y_2]\ r_1[y_2]\ c_1\ c_2$$

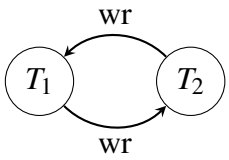


Fig. 4.4 DSG(H_{G1c}) displays a Circular Information Flow (G1c) anomaly.

Serializable Serializable isolation prevents anomalies G0, G1, and G2, the *DSG* cannot contain cycles containing one or more anti-depends edges. In short, the *DSG* cannot contain any cycles. H_{G2} below contains a G2 anomaly and the corresponding *DSG* is shown in Figure 4.5.

$$H_{G2} = r_1[x_0]\ w_2[x_1]\ r_2[y_0]\ w_1[y_1]\ c_1\ c_2$$

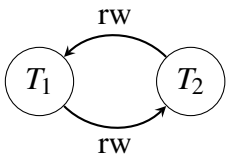


Fig. 4.5 DSG(H_{G2}) displays a G2 anomaly.

4.4.3 Mixing of Isolation Levels

To define a correctness criteria for a mixed DBMS, a *DSG* variant is used to represent a mixed history, referred to as a *mixed serialization graph*, $MSG(H)$. A *MSG* only includes *relevant* and *obligatory* conflicts. A relevant conflict is a conflict that is pertinent to a given isolation level, e.g., read-depends edges are relevant to Read Committed transactions but not Read Uncommitted transactions. An obligatory conflict is a conflict that is relevant to one transaction but not the other, e.g., an item-anti-depends edge between a Read Committed transaction and a Serializable transaction is relevant to the Serializable transaction and not the Read Committed transaction but still must be included in the *MSG*. Adya defines the edge inclusion rules for an *MSG* as follows:

1. Write-depends edges are relevant to all transactions regardless of isolation level thus always included.

2. Read-depends edges are relevant for edges incoming to Read Committed, Repeatable Read, or Serializable transactions.

3. Item-anti-depends edges are included for outgoing edges from Repeatable Read and Serializable transactions.

4. Predicate-anti-depends edges are included for outgoing edges from Serializable transactions.

Now in a mixed DBMS, a history is correct if each transaction is provided the isolation guarantees that pertain to its level leading to the *mixing-correct theorem* [1]. Figure 4.6 illustrates the differences between DSG (Figure 4.2) and MSG representations of a history with the non-relevant and non-obligatory edges removed.

Theorem 2 (Mixing-Correct Theorem) *A history H is mixing-correct if $MSG(H)$ is acyclic and phenomena G1a and G1b do not occur for Read Committed, Repeatable Read, and Serializable transactions.*

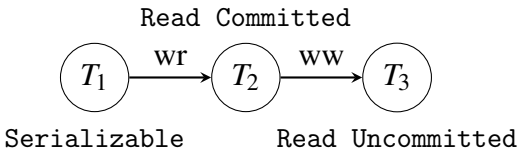


Fig. 4.6 Mixed serialization graph representation of H.

4.5 Mixed Serialization Graph Testing

In this section, we describe mixed serialization graph testing, focusing on the adjustments to the SGT algorithm (sketched in Subsection 4.2.1) and the concurrent graph data structure

described in Subsection 4.2.2. Subsection 4.5.2 discusses several potential performance optimizations.

4.5.1 Protocol Description

For MSGT we use the graph data structure and node-level locking protocol in Subsection 4.2.2 to represent an *MSG*, which requires one alteration: nodes include transactions' required isolation levels. MSGT proceeds in the same manner as SGT, with one key exception: for each operation, edge insertion of a detected conflict is subject to MSG's edge inclusion rules enumerated in Subsection 4.4.3. MSGT's edge insertion algorithm is given in Algorithm 7. First, if an edge already exists for this conflict type no further action is needed, else a cascading abort check is performed. If the parent node has aborted and the inserting node is Serializable or Read Committed it must also abort to avoid G1a anomalies. Then, the edge is inserted iff it satisfies MSG's inclusion rules, before a cycle check is executed. If a cycle is found the transaction must abort. At commit time the transaction delays until it has no incoming edges.

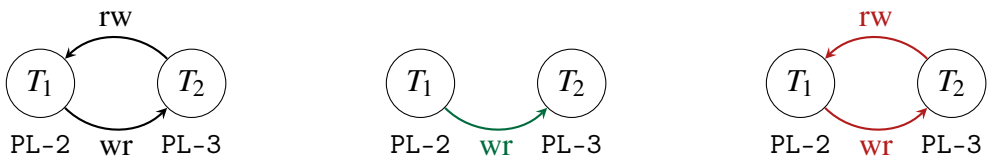
4.5.2 Optimizations

Restricted DFS Our first optimization arises from the following observation: the reduced DFS used in the SGT implementation described in Subsection 4.2.2 can result in a transaction aborting due to detection of a cycle it is not involved in. For example, assume T_i has detected a conflict with T_j, thus inserted $T_j \rightarrow T_i$. Now, assume T_j and T_k are involved in a cycle, $T_j \rightarrow T_k \rightarrow T_j$. It is possible for the cycle check initiated from T_i to detect this cycle and unnecessarily abort. This can be solved by *restricting* the DFS to only detect cycles involving the edge which has just been inserted. Informally, this is correct as if adding an edge introduces a cycle into the graph, then it and the two nodes it connects will be members of the cycle. *Restricted DFS* can benefit both SGT and MSGT.

Algorithm 7: MSGT Edge Insertion

```
1  Input: Node& this, Node& from, Conflict cType
2  if (from.id,cType) ∉ this.inSet then
3      if cType != RW ∧ this.iso == (RC ∨ S) ∧ from.state == aborted then
4          return false // cascading abort
5      if cType == ww then
6          this.inSet.add(from.id)
7          from.outSet.add(this.id)
8      else if cType == wr ∧ this.iso != RU then
9          this.inSet.add(from.id)
10         from.outSet.add(this.id)
11     else if cType == rw ∧ from.iso == S then
12         this.inSet.add(from.id)
13         from.outSet.add(this.id)
14     else
15         return true // not relevant/obligatory
16     cycle = cycleCheck(thisNode)
17     return !cycle
18 else
19     return true // edge already exists abort
```

Relevant DFS The second optimization can be attributed to the observation that reduced DFS is blind to the type of edge being traversed and thus the type of cycle found, which can also contribute to unnecessary aborts. In other words, transactions can unnecessarily abort due to a detecting cycle that is not applicable to its isolation level. For example, the cycle involving T_1 and T_2 in Figure 4.7(a) is a G2 cycle, which is relevant to T_2 the Serializable transaction (Figure 4.7(c)), but not to T_1 the Read Committed transaction (Figure 4.7(b)). Without explicit consideration, T_1 could detect the cycle, which results in T_2 also (cascadingly) aborting due to the wr edge between T_1 and T_2. To address this we propose the following heuristic: assuming restricted DFS, if T_i has inserted a rw edge between itself and T_j, if T_i is a Read Committed or Read Uncommitted transaction, then if a G2 cycle is detected then T_j is aborted, else T_i aborts. We term this *relevant DFS*. Informally, this works as if a cycle is detected then it can be broken by aborting either transaction connected by the edge. In



(a) MSGT (b) T_1's view (c) T_2's view

Fig. 4.7 MSGT and individual transaction's relevant views.

the case of a G2 cycle, we leverage this to selectively abort the transaction with a higher isolation level where possible to minimize the impact of cascading aborts.

Early Commit Rule The third optimization comes from the observation that a non-relevant incoming edge can delay a transaction from committing. For example, consider T_i (Read Committed) is attempting to commit, but has a single incoming anti-depends edge from T_j (Serializable), $T_j \xrightarrow{\text{rw}} T_i$. Recall from Section 4.2, delaying commitment until a node has no incoming edges simplifies node deletion, ensures recoverability, and provides order-preservation. However, in the example, T_i can never lie on a cycle relevant to its isolation level (G0 or G1c cycle.), thus under the *early commit rule* a transaction can commit without delay *with* incoming edges, provided these are not relevant to the transaction's isolation level, i.e., T_i could commit immediately. Such a scheme should provide lower latency for transactions that require weaker isolation levels via this more flexible commit rule. In the example, T_i would not unnecessarily waste CPU cycles checking its incoming edge set.

Care, however, must be taken regarding node deletion if the early commit rule is used. To illustrate this, T_i may still be a member in a cycle for a transaction committing at a higher isolation level, e.g., T_j may insert the edge $T_i \xrightarrow{\text{rw}} T_j$. Similarly to that described in Subsection 4.2.1, removing the node upon commit would allow the possibility of missed cycles and hence executions not permitted under the mixing correct theorem. Thus, regardless of isolation level a node can only be scheduled for removal once *all* incoming edges have been removed. A possible implementation is to offload the management of early committed nodes to a background thread that periodically checks if all edges have been removed, at

which point the nodes can be deleted and recycled; this would allow nodes to be accessed, if needed, during cycle checking initiated by other nodes.

4.5.3 Discussion

MSGT has several benefits. Firstly, most DBMSs offer weak isolation levels and thus a considerable portion of real world applications are built atop such guarantees. Normally, weak isolation is an afterthought as DBMSs chase better performance, with MSGT weak isolation catered for as a first-class citizen. Such an approach provides a higher degree of concurrency and hence performance, whilst also providing the optimal property of no unnecessary aborts. The relevant DFS optimization further minimizes aborts and the early commit rule promises lower latency for transactions that require weaker isolation levels.

There are multiple drawbacks that must also be considered. It is worth noting the utility of weak isolation is limited to applications that can tolerate potentially non-serializable behaviour. Additionally, if a workload exhibits low contention or is designed in a manner such that anomalies provably do not occur [44], then the additional overhead of managing the MSG has little benefit. Unfortunately, if the early commit rule is deployed, the real-time commit order may no longer match the serialization order, i.e., order-preservation is violated. However, in the search for high performance this may be a trade-off the database operator is willing to make.

4.5.4 Implementation Details

MSGT was implemented in our prototype in-memory database (Subsection 2.3.1). The DBMS has a pool of worker threads and each transaction is pinned to a specific worker thread for its duration. Each worker thread has an independent workload generator. Thus when a transaction is committing we repeatedly execute the commit routine (check for incoming edges).

To ensure high operation throughput under a concurrent workload, the necessary data structures use atomic operations where appropriate. The use of totally ordered transaction ids was avoided using the address of the transaction's node in the graph as the transaction id to avoid contention updating a global atomic counter. For safe memory reclamation in a concurrent environment epoch-based garbage collection is used [46] and nodes' edge sets are recycled after a transaction is committed and deleted.

Adya's system model is defined in terms of a multiversion model, but as the MSGT scheduler allows transactions to optimistically read dirty records, the possibility of cascading aborts is introduced. Unwinding writes due to cascading aborts lead to unnecessary system load and are not useful for either the user of the system. Therefore, only one uncommitted transaction is allowed to modify a data item. Also, the prototype DBMS does not currently support predicate-based operations, thus a simple read/write model is assumed. Hence, some isolation levels, e.g., Repeatable Read, can not be captured.

4.6 Evaluation

In this section, we experimentally compare MSGT with SGT focusing on two implementations: a graph-based scheduler (SGT) and a mixed graph-based scheduler (MSGT). To be precise, we focus on answering the following questions:

- What are the performance benefits of MSGT vs. SGT under a workload with transactions executed at different isolation levels?
- Under what workload conditions does MSGT exhibit better performance than SGT?
- What is the overhead of MSGT vs. SGT when all transactions are executed at Serializable isolation?

Our first set of experiments in Subsections 4.6.1 to 4.6.4 use YCSB (described in Subsection 2.3.2) as a microbenchmark. This benchmark allows various aspects of an OLTP

workload to be altered to measure how the protocols perform under a variety of workload conditions. Specifically, we tweak the proportion of serializable transactions, vary the proportion of update transactions, differ the contention level, and increase the core count to measure scalability. YCSB's workload factors are summarized in Table 4.2. The next set of experiments in Subsections 4.6.5 and 4.6.6 model different application scenarios, namely, SmallBank and TATP described in Subsection 2.3.3 and Subsection 2.3.2 respectively. SmallBank is helpful in addressing our research question of ascertaining the overhead of MSGT compared to SGT, as all SmallBank transactions are executed at Serializable isolation. TATP also provides an ideal candidate for measuring the overhead of MSGT in comparison with SGT. However, TATP permits all transactions to be executed at Read Committed isolation. Thus, it is useful to help determine whether MSGT can still provide performance gains in workloads with a low abort rate.

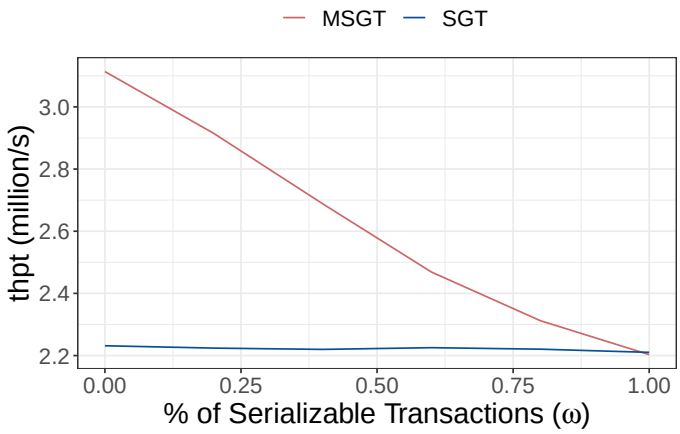
Table 4.2 YCSB Workload Factors

Symbol	Meaning	Values
ω	Contention Level	0.6-0.9
U	Proportion of update transactions	0.0-1.0
θ	Proportion of Serializable transactions	0.0-1.0
cores	DBMS core count	1-40

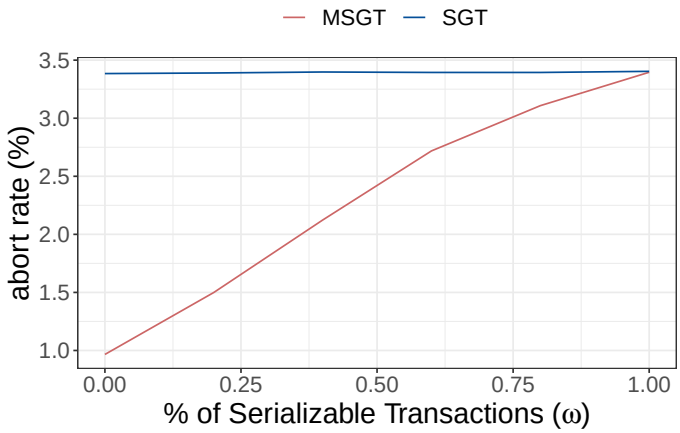
4.6.1 Isolation

We begin with measuring the impact of increasing the proportion of transactions executing at Serializable isolation (ω) from 0% to 100%. This aims to test MSGT's ability to leverage its theoretical properties to offer increased performance when transactions are run at weaker isolation levels. For this experiment, we opt for a medium contention level by setting $\theta = 0.8$, and the framework is configured to run with 40 cores.

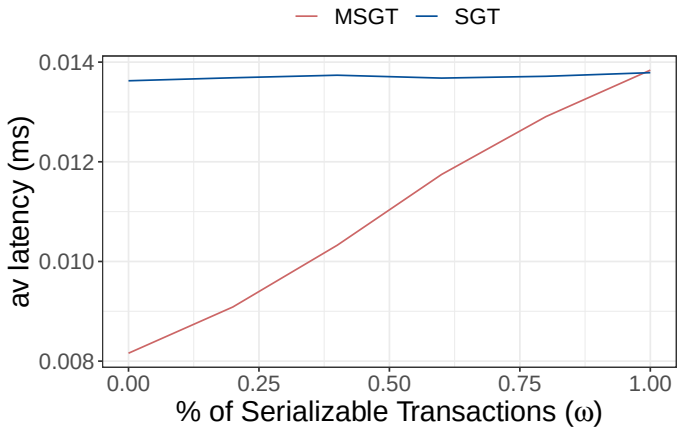
In Figure 4.8(a), SGT's throughput is invariant to the proportion of Serializable transactions, this is anticipated as it is unable to take advantage of transactions' declared isolation



(a) Throughput *vs.* ω.



(b) Abort rate *vs.* ω



(c) Average latency *vs.* ω.

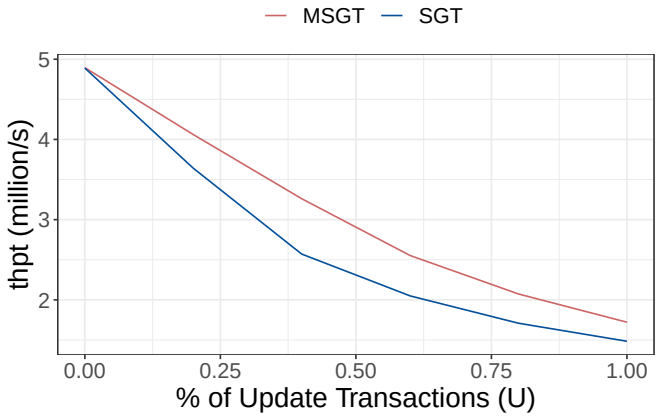
Fig. 4.8 **Isolation** – SGT and MSGT with 40 cores when varying the proportion of Serializable transactions from 0% to 100% with medium contention $\theta = 0.8$ and 50% update rate.

levels, in effect, executing all transactions at Serializable. Meanwhile, the throughput of MSGT decreases as ω is increased, converging towards SGT's throughput. When there are no Serializable transactions ($\omega = 0.0$), MSGT achieves a 39% increase in throughput. At $\omega = 0.4$, this drops to a 21% increase and at $\omega = 0.8$ a 4% gain is exhibited. When $\omega = 1.0$, SGT marginally outperforms MSGT, this can be attributed to the additional overhead of managing the MSG. This relationship is reflected in the abort rate displayed in Figure 4.8(b), across the range of ω, SGT's abort rate varies from a 3x increase over MSGT's abort rate to an equivalent abort rate when all transactions are executed at Serializable. A higher abort rate degrades the user experience, reduces throughput and, as can be seen in Figure 4.8(c), harms latency.

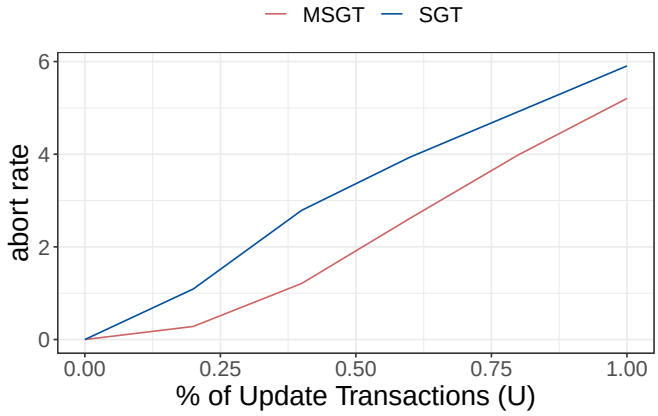
4.6.2 Update Rate

For the next experiment, we explore the effect of varying the proportion of update transactions (U). For this experiment, we opt for a medium contention level, $\theta = 0.8$, set the proportion of Serializable transactions to $\omega = 0.2$, with the framework configured to run with 40 cores.

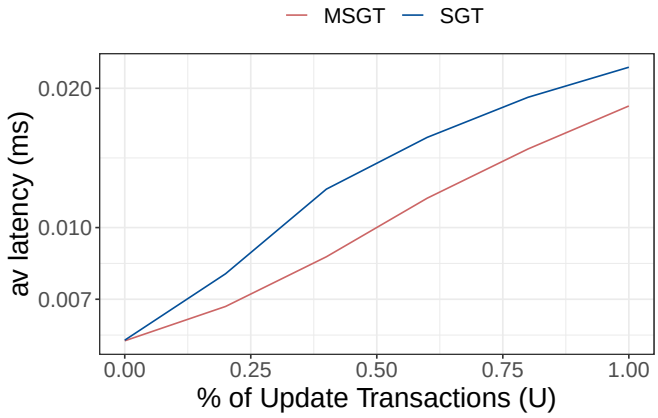
When $U = 0.0$, the workload is read-only thus no (ww, wr, rw) conflicts are generated. Thus, no edges are inserted and neither SGT or MSGT are required to perform any work, which explains why performance is indistinguishable across all metrics in Figure 4.9. As U is increased more updates are executed and more conflicts occur, thus both schedulers have work (edge insertion/cycle checks) to perform, which explains the downward trend in throughput in Figure 4.9(a). MSGT's throughput decreases at a slower rate as it is able to leverage its selective conflict detection rules, achieving between 11% and 27% higher throughput compared to SGT.



(a) Throughput *vs*. U.



(b) Abort rate *vs*. U



(c) Average latency *vs*. U.

Fig. 4.9 **Update Rate** – Performance measurements at 40 cores for the protocols as the proportion of update transaction (U) is varied from 0% to 100% with medium contention, $\theta = 0.8$, and a low proportion of Serializable transactions, $\omega = 0.2$.

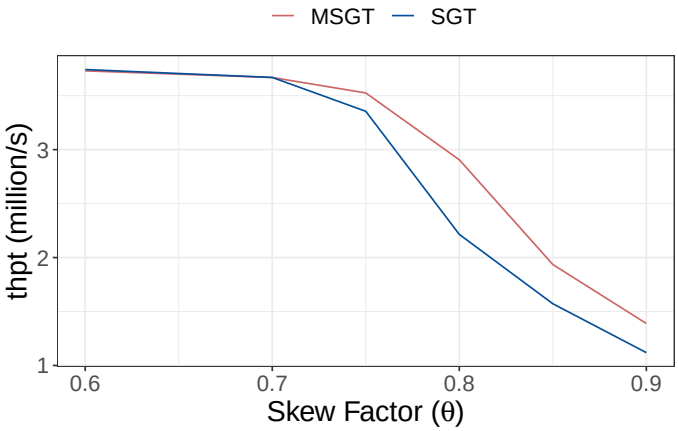
4.6.3 Contention

In the next experiment we measure the effect of increasing contention in the system by varying θ from 0.6 to 0.9. In theory, contention increases the chance of conflicts between transactions. This should translate into an increase in the number of edges inserted into the conflict graph. Under SGT all edges are inserted, whereas MSGT utilizes isolation levels to be more selective over edge insertions (only adding relevant or obligatory edges) hence it inserts fewer edges into the conflict graph, and should find fewer cycles (aborts) compared to SGT. We set the proportion of Serializable transactions to $\omega = 0.2$. Again the experiment was run with 40 cores.

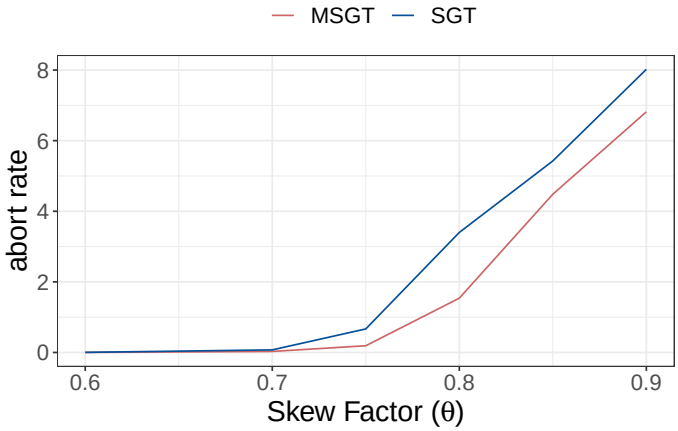
Figure 4.10(a) displays the throughput of SGT and MSGT as the contention is increased. As θ increases the throughput decreases for both protocols. For low levels of contention SGT performs marginally better than MSGT ($<$1% difference), but under high contention this reverses and MSGT offers a 24% increase in throughput. Figure 4.10(b) shows that after $\theta = 0.7$, the abort rate begins increasing for both protocols. At the highest level of contention ($\theta = 0.9$), 0.1% of the data is accessed by 35% of the queries, and SGT aborts 17% more transactions than MSGT. Lastly, in Figure 4.10(c), above $\theta = 0.7$, MSGT achieves between a 9% and 28% reduction in the average latency.

4.6.4 Scalability

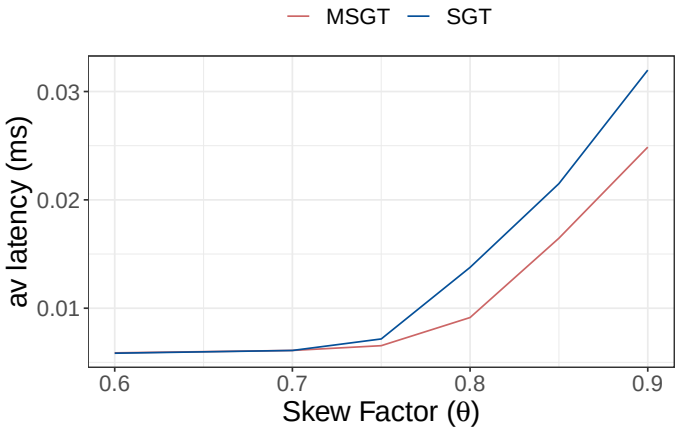
The previous three experiments have investigated the impact of several workload factors in a database with 40 cores. In this experiment we fix the workload factors and vary the core count (1 to 40) to evaluate MSGT's scalability compared to SGT. We anticipate that MSGT scales better than SGT as its scheduler generally performs less work (edge insertions and cycle checking). From Figure 4.11(a) it can be seen that until 20 cores the throughput of both protocols is indistinguishable; in fact, up to 10 cores SGT exhibits between a 1.2% and 3.1% increase over MSGT. After this point, a gap appears, at 30 cores MSGT has 13.1% higher



(a) Throughput *vs.* θ.



(b) Abort rate *vs.* θ



(c) Average latency *vs.* θ.

Fig. 4.10 **Contention** – SGT and MSGT with 40 cores when varying the contention (skew factor) in the YCSB workload with $\omega = 0.2$, and $U = 0.5$.

throughput and at 40 cores this difference increases to 27.9%. In Figure 4.11(b) it can be seen the abort rate of the protocols starts to diverge after 10 cores: SGT has an abort rate of 1.65% and 3.39% at 30 and 40 cores respectively, whereas MSGT's is 0.50% and 1.54%.

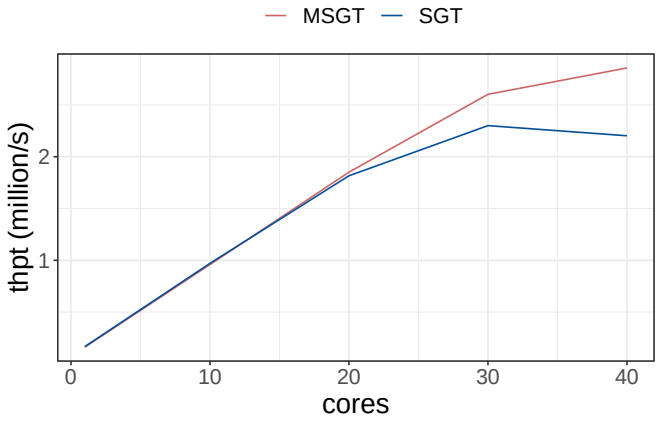
4.6.5 SmallBank

Next we test the protocols using the SmallBank workload. In this workload all transactions are executed at Serializable isolation, we consider the case of unusual high contention by setting the database to contain only 100 customers; transaction requests are uniformly distributed across the 100 customers. This creates a worst case scenario for MSGT, the high contention results in a significant number of edge insertions, and it cannot leverage its selective edge insertion rules due to all transactions requiring the highest isolation level. Therefore, this allows us to quantify the overheads of MSGT compared to SGT.

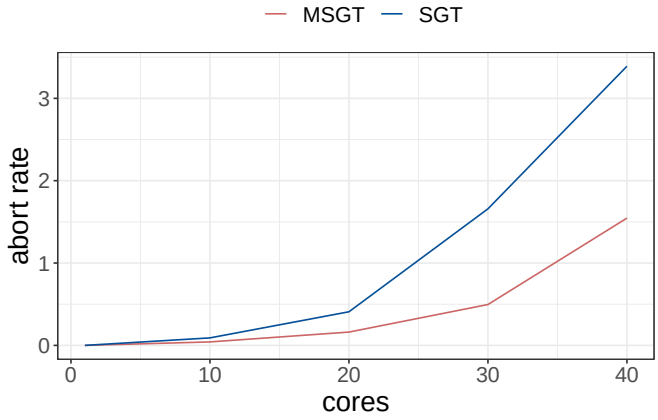
It is clear from Figure 4.12 that MSGT has a non-negligible overhead, with SGT displaying improved performance across all metrics. Thus, if a workload requires all transactions to execute at Serializable isolation then SGT is the better choice. In Figure 4.12(a), at 30 cores, SGT has a 6% increase in throughput over MSGT. From Figure 4.12(b), it can be seen that SGT surprisingly has a lower abort rate than MSGT, this can be explained by Figure 4.12(c), which shows the average latency is higher under MSGT. The longer transactions are in the database, the more their chances for conflict increases, which results in more aborted transactions.

4.6.6 TATP

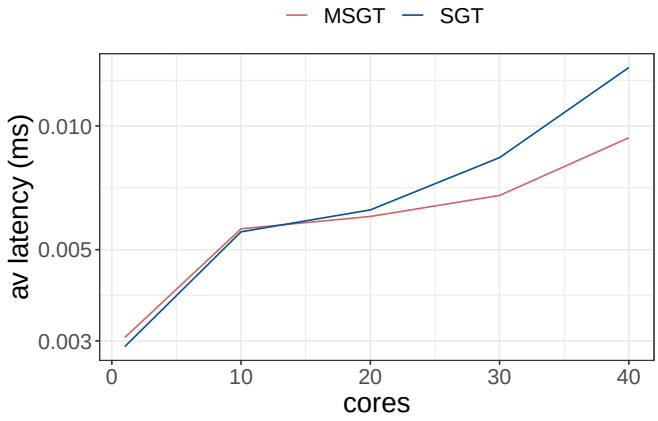
Lastly, we examine the scalability of the protocols using the TATP workload. TATP is interesting as its transactions generate few conflict cycles. Thus, aborts are naturally minimized especially for a protocol that accepts all conflict serializable schedules (SGT). However, it also only requires transactions to be executed with Read Committed isolation. This provides



(a) Throughput *vs*. U.

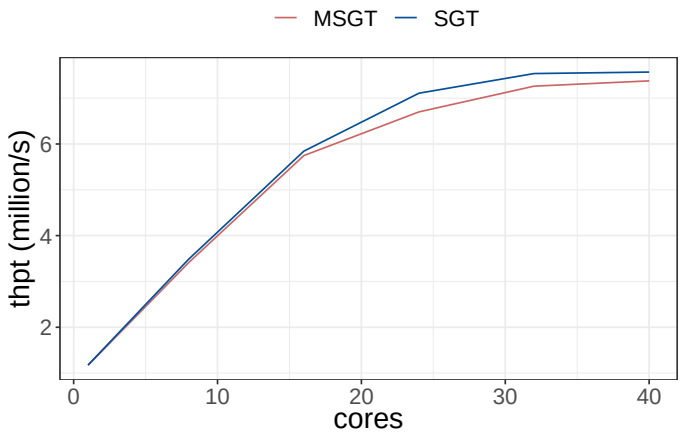


(b) Abort rate *vs*. U

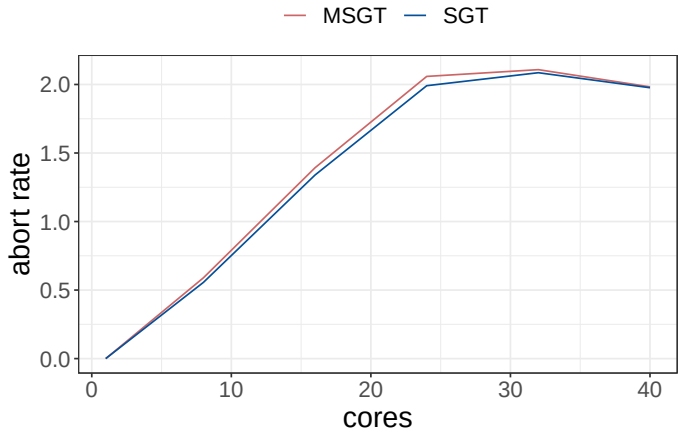


(c) Average latency *vs*. U.

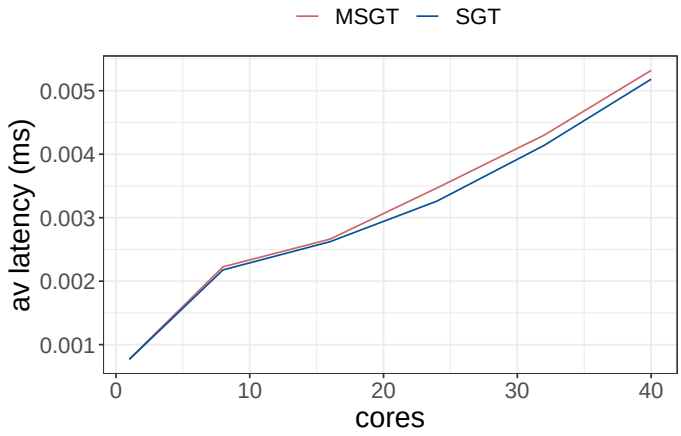
Fig. 4.11 **Scalability** – Performance measurements for the protocols as the core count is increased from 1 to 40 cores with medium contention, $\theta = 0.8$, low proportion of Serializable transactions, $\omega = 0.2$, and a medium update rate $U = 0.5$.



(a) Throughput *vs.* ω.



(b) Abort rate *vs.* ω



(c) Average latency *vs.* ω.

Fig. 4.12 **SmallBank** – 100 customers (high contention) with all transactions executed at Serializable isolation.

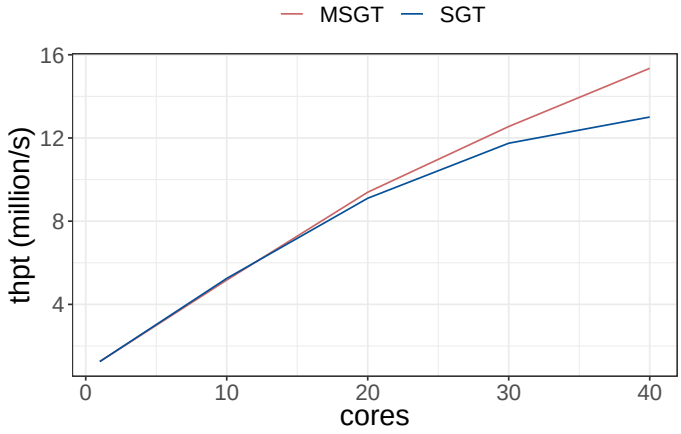
a situation to evaluate whether MSGT can still provide a performance gain in workload with low abort rates, i.e., when cycles are not common.

Figure 4.13 demonstrates in realistic workloads with a low conflict rate, MSGT can outperform SGT across all metrics, provided transactions are run at weaker isolation levels. In Figure 4.13(a), at 40 cores MSGT achieves at 10% increase in throughput, this is matched with a significant decrease in the abort rate and latency (Figures 4.13(b) and 4.13(c)).

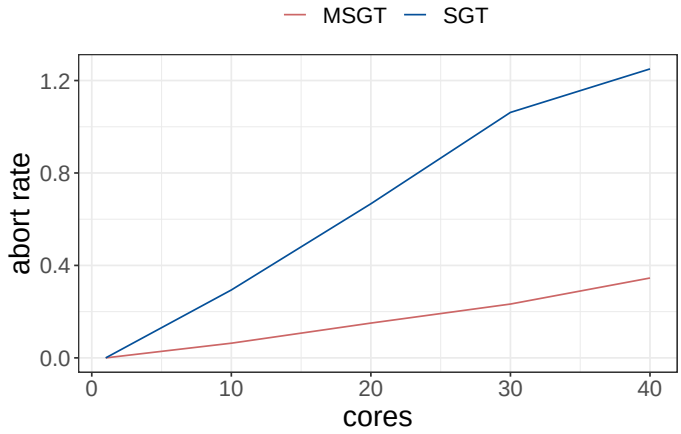
4.6.7 Concurrency Control Overhead

To separate the concurrency control overhead from the actual transaction workload, we turned off the tuple access history, conflict detection, cycle checking, and restarted handling for both protocols. We re-ran the TATP and SmallBank experiments, and ran YCSB with high contention ($\theta = 0.9$), balanced read/update transaction mix ($U = 0.5$), and half of the transactions running at Serializable isolation ($\omega = 0.5$). All workloads were run with 40 cores. Note, the only concurrency control requirement for "No CC" is that tuple accesses are serialized.

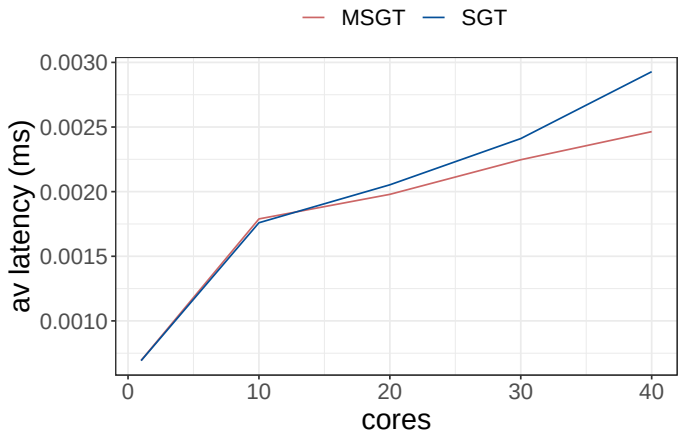
Table 4.3 displays the results. In all experiments, both protocols spend the majority of time for concurrency control: tuple access history, conflict detection, cycle checking, and restart handling. SGT has marginally lower overhead in the SmallBank experiment, which contains all Serializable thus MSGT wastes cycles unnecessarily checking if it can avoid inserting an edge. In TATP and YCSB, MSGT exhibits a 1.3% and 4.9% less overhead compared to SGT. In these workloads, MSGT is able to insert fewer edges, which results in few cycle checking invocations and a smaller graph to explore when cycle checking is required.



(a) Throughput *vs.* ω.



(b) Abort rate *vs.* ω



(c) Average latency *vs.* ω.

Fig. 4.13 **TATP** – 100 entries all transactions executed at Read Committed isolation.

Table 4.3 Overhead of maintaining accesses, conflict detection, cycle tests, aborts, and live-lock handling. Reported metric is throughput at 40 cores.

Benchmark	SmallBank	TATP	YCSB
No CC	26.7M	36.9M	3.88M
SGT	7.57M	13.4M	0.67M
MSGT	7.41M	15.2M	0.72M
SGT Overhead	71.7%	63.7%	82.7%
MSGT Overhead	72.3%	58.8%	81.4%

4.6.8 Optimizations

In this section, we explore the performance implications of the optimizations described in Subsection 4.5.2.

Cycle Checking Strategies In this experiment, we compare the performance of reduced, restricted, and relevant DFS cycle checking strategies for MSGT. We repeat the isolation experiment from Subsection 4.6.1, varying the proportion of transactions executing at Serializable isolation from 0% to 100%; medium contention, $\theta = 0.8$, balanced read/update transaction ratio, $U = 0.5$, and 40 cores.

Interestingly, as seen in Figure 4.14(b) reduced DFS has a lower abort rate than restricted DFS. One explanation is under restricted DFS cycle checking takes longer, as any discovered cycles must include the inserted edge, increasing the transaction lifetime and hence resulting in more conflicts and cycles. This hypothesis is supported by Figure 4.14(c) in which MSGT with restricted DFS has the highest average latency. In Figure 4.14(b), relevant DFS displays the lowest abort rate, between a 6% and 16% decrease compared to reduced DFS. Surprisingly, this does not translate into higher throughput or lower latency in Figures 4.14(a) and 4.14(c). One explanation for this it that transactions with lower isolation are still delayed from committing as they have incoming edges from higher isolation transactions that will

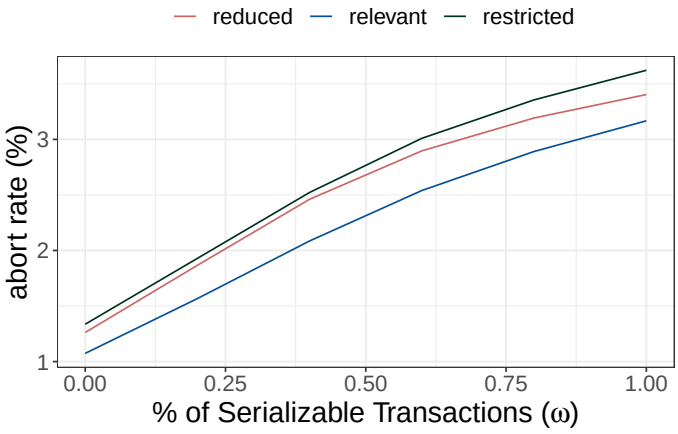
subsequently abort. Potentially, this can be solved with the early commit rule, which is now given preliminary investigation.

Early Commit Rule To investigate the potential usefulness of the early commit rule, the above experiment was repeated, however now when a Read Uncommitted or Read Committed transaction was committing we checked whether it *could* have committed early, computing the proportion of transactions that would have benefitted from the rule, *ce*. When $\omega = 0.0$, we found $ce = 0\%$, as there were no Serializable transactions to introduce rw edges into the MSG. At $\omega = 1.0$, we also found $ce = 0\%$, as there are no weaker isolation transactions to leverage the rule. In between the extremes, we found that up to 2.5% of weaker isolation transactions were prevented from committing early due to non-relevant edges, when $\omega = 0.8$. Such a non-negligible proportion surely would impact performance, thus these provisional findings motivate a full implementation and evaluation of the early commit rule.

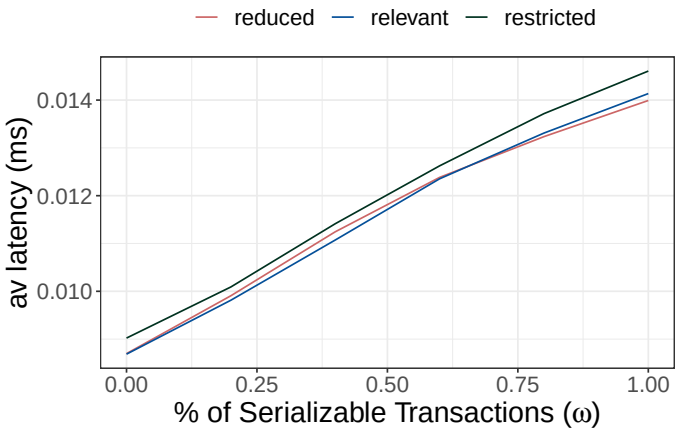
4.7 Conclusion

In this chapter we presented mixed serialization graph testing, a graph-based scheduler that leverages Adya's mixing-correct theorem to permit transactions to execute at different isolation levels. When workloads contain transactions running at weaker isolation levels, MSGT is able to outperform serializable graph-based concurrency control by up to 28%. Additionally, MSGT scales as the number of cores is increased, an important property given modern hardware. Like SGT, MSGT minimizes the number of aborted transactions, accepting all useful schedules under the mixing-correct theorem, which greatly improves user experience. Additionally, we presented several optimizations: restricted DFS, relevant DFS, and early commit that aim to further reduce aborts and decreases latency respectively. In summary, this work in this chapter strengthens recent work refuting the assumption that graph-based concurrency control is impractical.

(a) Throughput *vs.* ω.



(b) Abort rate *vs.* ω



(c) Average latency *vs.* ω.

Fig. 4.14 MSGT with various cycle checking strategies when varying the proportion of Serializable transactions from 0% to 100% with medium contention $\theta = 0.8$, 50% update rate, and 40 cores.

4.7.1 Further Work

Additional Isolation Levels As demonstrated in Section 4.3, databases offer a wide range of isolation levels and Adya provides a definition of several (11 in total [1]) which could possibly be expressed by the MSGT protocol. Of notable interest would be the popular Snapshot Isolation [11].

The simplest extension would be to incorporate Adya SI [1] . To model this isolation level, Adya extends the system model to include a transaction's logical start and commit timestamps. This graph is referred to as a *start-ordered serialization graph*, $SSG(H)$, and adds *start-dependency edges* between the nodes, i.e., two transactions T_i and T_j are start-ordered if the commit timestamp of one precedes the start timestamp of the other. The simplest implementation would use a centralized timestamp allocator to add the start and end timestamp nodes. This however reintroduces a global bottleneck which could negatively impact performance.

Additional Experiments As part of future work we wish to extend our performance evaluation to include industry standard benchmarks, e.g., TPC-C [109], and analyze their isolation requirements to help understand where various transaction types occur in practice. Another avenue for further work would be to evaluate MSGT's performance against other mixed concurrency control protocols such as mixed 2PL. This will allow for validation of the claim that MSGT accepts all valid schedules under the mixing-correct theorem and hence reduces aborts.

Distributed MSGT Another avenue for future work is exploring how MSGT can be integrated into a distributed shared-nothing database. Specifically, how isolation levels, e.g., Read Committed, can be highly available [8] in the presence of concurrent transactions executing at isolation levels that are provably unavailable, e.g., Serializable.

Mixed Wait-Hit Protocol Also as part of future work we will explore how the Wait-Hit Protocol can be extended to mixed environments. As in MSGT the inclusion of a given edge is determined when the associated conflict is detected this should also be feasible in the Wait-Hit Protocol with certain edges now not included in either the predecessor-upon-read or predecessor-upon-write sets.

Chapter 5

Edge Consistency in Distributed Graph Databases

Summary

This chapter discusses the design and evaluation of protocols for ensuring the preservation of edge consistency in distributed graph databases. Three protocols are presented: Delta protocol, Deterministic Reciprocal Consistency Protocol, and Deterministic Edge Consistency Protocol. The first two focus on preserving Reciprocal Consistency and leverage the fact that the updating of end pointers of a distributed edge must immediately follow each other and the small interval between them is the sole raison d'être for Reciprocal Consistency violations.

The Delta protocol is a lightweight, locking-free protocol, which depending on the choice of the protocol's parameter, Δ, still permits inconsistencies. Evaluations establish that the protocol can offer both integrity guarantees and sound performance when the value of its parameter is chosen appropriately.

The Deterministic Reciprocal Consistency Protocol ensures no possibility of Reciprocal Consistency violations. The protocol associates metadata with each update of an edge pointer, which is tracked and used to identify violations, aborting the relevant transaction if so. Corruption is thus prevented, at the expense of some aborted transactions.

The Deterministic Edge Consistency Protocol extends the Deterministic Reciprocal Consistency Protocol to provide an additional guarantee: Edge-Order Consistency. This protocol is a simplified version of the Wait-Hit Protocol. Performance evaluations investigate the magnitude of aborts under various system configurations.

5.1 Introduction

Section 2.2 introduces graph databases, specifically two important edge consistency guarantees in these systems: Reciprocal Consistency and Edge-Order Consistency. Further in Subsection 2.2.3, we illustrated how Reciprocal Consistency can be violated in a distributed graph database built atop a NoSQL database. Importantly, the resulting data corruption can swiftly

propagate throughout the database rendering it in a state with little operational value. This motivates the development of protocols that prevent said violations, and as a consequence, slow or avert this corruption from spreading. To this end, two protocols have been developed.

The Delta protocol is presented in Section 5.2 and is exclusively designed for one purpose only: Reciprocal Consistency in distributed edges. The protocol assumes that the interval that elapses between a transaction updating both pointers of a distributed edge can be estimated. Thus, by selecting the parameter Δ to be larger than this value, it can be assured when a transaction is updating an edge pointer, that both updates of previous updates to that distributed edge have completed. If a preceding update within Δ is encountered the updating transaction aborts (although in some cases this may be unnecessary). Therefore, none of the conflicts arising from concurrent writes outlined in Figure 2.4 will occur and semantic corruption will not spread; a correctness reasoning is provided in Subsection 5.2.2. However, there is a chance that the selected value for Δ is an underestimate and violations of Reciprocal Consistency still occur. To measure the efficacy of the Delta protocol the model developed in [41, 119] for measuring the degradation of distributed graph databases built atop NoSQL databases is augmented. This requires an adjustment of the conflict probability, q_i, which is derived in Subsection 5.2.3. The evaluation uses two metrics that measure the protocol's ability to avoid inconsistencies (ensuring *safety*) and unnecessary aborts (enhancing *throughput*); Subsection 5.2.3 explains the strategies used for evaluating these metrics. The evaluations in Subsection 5.2.4 establish that the protocol can offer both integrity guarantees and sound performance when the value of its parameter is chosen appropriately.

The Deterministic Reciprocal Consistency Protocol is then presented in Section 5.3, the key distinction between itself and the Delta protocol is that Reciprocal Consistency is preserved under all eventualities. This is achieved using a *collision detection* mechanism

which leverages metadata associated with each update of an edge pointer to identify violations. When violations are detected the offending transactions are aborted.

We then extend Deterministic Reciprocal Consistency Protocol in Section 5.4 to prevent violations of Edge-Order Consistency, a different type of conflict that can arise when transactions update multiple edges during their lifetime. The Deterministic Edge Consistency Protocol combines *collision detection* with a mechanism called *order arbitration*, to enforce Reciprocal Consistency for distributed edge and Edge-Order Consistency between transactions respectively. Collision detection is applied at every update and may trigger an immediate abort. Transactions that survive collision detection may, if necessary, go to order arbitration. The latter may also result in an abort. To investigate the performance of the Deterministic Edge Consistency Protocol protocol, we develop an approximate model that allows for the computation of performance measures, including the fraction of aborted transactions and average transaction response time. The accuracy of the approximations is assessed by comparing them with simulations, for a variety of parameter settings.

5.2 Delta Protocol

As stated in Subsection 2.2.3, a violation of Reciprocal Consistency is a manifestation of a Dirty Write in the context of a distributed graph database. A straightforward solution to preventing dirty writes is for transactions to take long duration write locks [11], releasing them only after the acquiring transaction has committed or aborted. To prevent deadlock, a policy such as `NO-WAIT` deadlock avoidance is used, which was shown to be the optimal policy in a distributed, partitioned database [54].

The Delta protocol employs principles behind all these well-tested strategies but has two crucial differences: (i) no locks are used, and (ii) a write operation need not wait until the preceding write commits, but can proceed if at least Δ duration (measured by local clock) has elapsed. These differences lead to several advantages in the context of graph databases

which are characterized by the following two aspects. First, there is usually a subset of edges that are traversed and modified with a very high frequency, e.g., critical sections of motorway in a road network, leading to high contention. Secondly, graph transactions tend to be longer-lived than those in other databases. So, having to wait for earlier writes to terminate while transactions are long lived will severely limit the scope for concurrent processing and reduce throughput in a highly contentious environment.

With these concerns in mind the Delta protocol was developed as a protocol that neither uses locks nor requires transactions to wait until earlier transactions terminate.[1] The Delta protocol aims at preventing edges from becoming half-corrupted and hence quashing the seed of corruption whilst keeping performance at an acceptable level.

5.2.1 Protocol Description

Recall from Subsection 2.2.3, each end of a distributed edge maintains *start sequence number* (ssn) to indicate the number of transactions that modified the edge starting from its end. It also maintains a *finish sequence number* (fsn) to indicate the number of transactions that modified the edge starting from the other end.

The Delta protocol has five rules:

1. A transaction T_i's write on an end pointer of an edge is initially tentative which would become permanent only if that transaction is permitted to commit.

2. A tentative write at the final end is possible only if the end pointer is either in a committed state or the immediately preceding tentative write was done at least Δ time before (where time is measured as per local clock) and $fsn = ssn_i - 1$ at the server, where ssn_i is the sequence number T_i obtained when it started modifying the distributed edge.

[1]The Delta protocol is a concurrency control mechanism only for distributed edge updates; it does not concern itself with updates on vertices.

3. If T_i performs all its tentative writes, then it is permitted to commit; otherwise, it must abort.

4. A transaction commits when all its tentative writes are made permanent, e.g., by using an atomic commit protocol.

5. Tentative writes of an aborting transaction are ignored. An ignored tentative write can make a new transaction abort unnecessarily for up to Δ time after it was created; it is harmless thereafter and can be garbage collected at any time.

5.2.2 Correctness Reasoning

Let us define δ as the bound *estimate* on the interval that may elapse between a transaction completing its update at one end of a distributed edge at one server and starting its update at the other end of the same edge at another server. To aid readability Figures 2.4(a) and 2.4(b) are reproduced in Figure 5.1.

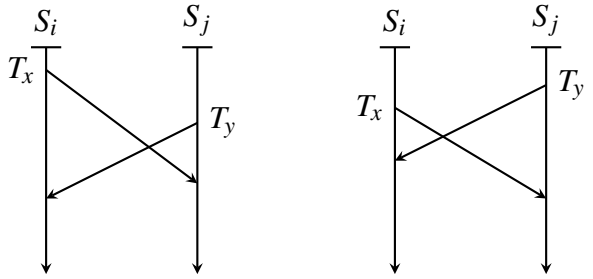


Fig. 5.1 Interleavings of concurrent writes to a distributed edge by transactions T_x and T_y.

Let Δ be chosen such that $\Delta > \delta$. Now consider the interleaving in Figure 2.4(b) and let t_x be the (global) time when T_x starts at server S_i; similarly t_y be the time T_y starts at server S_j. Since T_x starts after T_y in Figure 2.4(b), $t_x > t_y$, i.e. $(t_x - t_y) > 0$.

Say T_y reaches S_i at time $t_y + d_y$, where d_y is the *actual* time elapsed between T_y completing a tentative write at one end and starting at another end, let us assume that $d_y \leq \delta$.

When T_y arrives at S_i it will find a tentative write already done at time t_x. In this case, $t_y + d_y - t_x = d_y - (t_x - t_y) < d_y \leq \delta < \Delta$; so, T_y will abort, preventing writes from interleaving and half-corrupting the edge. Similar arguments can be made for the scenario in Figure 2.4(a), if $d_x \leq \delta$ where d_x is the *actual* time elapsed between T_x completing a tentative write at one end and starting at another end. Note that $t_y > t_x$ and T_x will abort because it will find out, on reaching server S_j for its next tentative write, that $t_x + d_x - t_y = d_x - (t_y - t_x) < d_x \leq \delta < \Delta$.

Assume that $d_x > \delta$ or $d_y > \delta$ is possible, i.e., the estimate δ can fail to hold. In Figure 2.4(b), interleaving writes are avoided only if $t_y + d_y - t_x = d_y - (t_x - t_y) < d_y < \Delta$; otherwise, Reciprocal Consistency can be violated. Similarly, interleaving writes of Figure 2.4(a) are avoided only if $t_x + d_x - t_y = d_x - (t_y - t_x) < d_x < \Delta$; otherwise, Reciprocal Consistency can be violated. Thus, the probability of reciprocal consistency not being guaranteed is the probability that $\Delta \leq \max\{d_x, d_y\}$.

In summary, the Delta protocol eliminates interleaving of transactions during edge writes, so long as Δ remains larger than the interval d that elapses between a transaction completing its write at one end of a distributed edge and starting at the other end. Since the exact value of d taken by a transaction cannot be known in advance, its bound δ is estimated with the best effort and Δ is chosen to be $\Delta > \delta$.

The larger the value of Δ used, the more likely it is that $\Delta > d$ holds and half-corruption and thereby operational corruption are averted; also, on the downside, the more likely it is that non-interleaving transactions will find their tentative writes within Δ time of each other and the later ones choose to abort *unnecessarily*. In the extreme case, choosing a very large Δ ($\Delta \approx \infty$) totally eliminates any risk of Reciprocal Consistency violation but does not allow any tentative write until the preceding one is made permanent. It is equivalent to enforcing `NO-WAIT` policy, wherein a requesting transaction that finds the requested record being locked, must abort.

Our performance evaluation in Subsection 5.2.3 will therefore measure the following two metrics for various values of Δ:

- **Time to operational corruption:** time taken for $\gamma\%$ of a large database to be corrupted,
- **Abort rate:** number of transactions aborted per second.

If the actual time between completing at one end and starting at the other, d, is exponentially distributed with mean $1/\mu$, the probability of d exceeding Δ is $e^{(-\mu\Delta)}$. The values of Δ chosen will explore a range of probabilities of Δ being exceeded.

5.2.3 Performance Evaluation Strategies

To assess the time to operational corruption, the model developed in [41] is adapted for the Delta protocol. For ease of the reader, the model is explained here before we discuss the adaptation. We note that an empirical evaluation of the time to operational corruption using a real system would be impractical due to the sheer length of time and the cost it would take to run the experiment. (For certain values of Δ, the model predicts anywhere between 1-75 years for operational corruption!)

Degradation Model As stated in Subsection 2.2.3, when an edge is updated, both reciprocal entries in the adjacency lists of the source and the target nodes must be updated. From a modeling perspective, an update to a local edge is assumed to be instantaneous. For a distributed edge, a write operation is carried out first on one of its servers and then, after a small but non-zero delay, d, on the other. This implementation of distributed writes makes it possible for edge records to become half-corrupted. The interleavings illustrated in Figure 2.4 will be referred to as *conflicts*.

One end of a half-corrupted edge may be considered correct, but an external observer cannot tell which is which. The question of exactly which half is corrupt is decided by what

happens subsequently. Suppose that a future transaction first reads one edge pointer of a half-corrupted edge say, at vertex a. At that moment, the transaction implicitly chooses the order and thereby invalidates the other order that prevails at vertex b. Thus, from that moment onward, the b end of an edge ab becomes the corrupt end and, conversely, the a end becomes the *correct* end.

A subsequent transaction which happens to read the correct entry of a half-corrupted edge, and completes a write operation for it without a conflict, will repair the fault and make the edge record clean again. Else, if it reads the incorrect entry and writes any edge, it causes the target to become semantically corrupted, or simply *corrupted*. The correct and incorrect reads are equally likely, so each occurs with probability 1/2.

Any edge can become corrupted by being written on the basis of reading incorrect information. Corrupted edges cannot be repaired, since there is no post-facto solution to the graph repair problem in the general case. Transactions that update edges arrive in a Poisson stream with rate λ per second (TPS). We assume that each transaction contains a random number of read operations, K, followed by one write operation. This is a conservative assumption, since more than one write per transaction would increase the rate of corruption. The variable K can have an arbitrary distribution, $P(K = k) = rk$, $k \geq 0$. In practice K tends to be at least 2, i.e., $r0 = r1 = 0$. The K edges read, and the one written by the transaction are assumed to be independent of each other (but note below that they are not equally likely).

The edges in the database are divided into T types, numbered $1, 2, ..., T$. The probability that a read or a write operation accesses an edge of type i is p_i, with $p_1 > p_2 > ... > p_T$. That is, type 1 is most popular, type 2 is the second most popular and so on. The number of edges of type i is N_i, and typically $N_1 < N_2 < ... < N_T$. The total number of edges is N. In every type, a fraction f of the edges are distributed and the rest are local. The probability of accessing a particular edge of type i, for either reading or writing, is $\frac{p_i}{N_i}$. At time 0, all edges are clean (free from corruption). When a certain fraction, γ (e.g., $\gamma = 0.1$), of all

edges become corrupted, the database itself is said to be corrupted for practical purposes, i.e., operationally corrupt. The object of the model is to provide an accurate estimate of the length of time that it takes for this to happen. In summary, at any moment in time, an edge belongs to one of the following four categories: **category 0:** local and clean, **category 1:** distributed and clean, **category 2:** half-corrupted, and **category 3:** semantically corrupted. Figure 5.2 visualizes the various edge states and possible transitions between them.

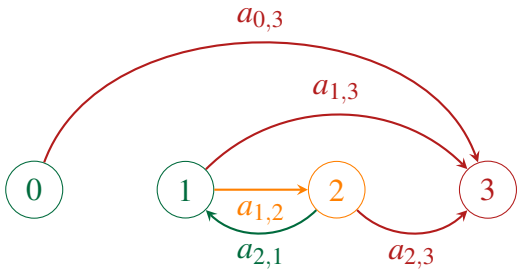


Fig. 5.2 Edge transitions between clean, half-corrupted and semantically corrupt states.

Only distributed edges can be in category 2, but any edge, including local ones, can be in category 3. Denote by $n_{i,j}(t)$ the number of type i edges that are in category j at time t. The set of vectors $n_i = [n_{i,0}(t), n_{i,1}(t), n_{i,2}(t), n_{i,3}(t)]$, for $i = 1, 2, ..., T$, defines the state of the database at time t. At all times, the elements of vector n_i add up to N_i. Any state such that

$$\sum_{i=1}^{T} n_{i,3}(t) \geq \gamma N \tag{5.1}$$

will be referred to as an *absorbing state*. The absorbing states correspond to an operationally corrupt database. The value of interest is U, the average first passage time from the initial state where $\mathbf{n_i} = [(1-f)N_i, fN_i, 0, 0]$ (i.e., a clean database), to an absorbing state. The above assumptions and definitions imply that a read operation performed at time t would return a correct answer with probability α, given by

$$\alpha = \sum_{i=1}^{T} \frac{p_i}{N_i} [n_{i,0}(t) + n_{i,1}(t) + \frac{1}{2} n_{i,2}(t)] \tag{5.2}$$

Note that the last term accounts for the read operation choosing the correct end of a half-corrupted, category 2 edge which happens with probability 1/2. The probability, β, that all the read operations in a transaction arriving at time t return correct answers, is equal to

$$\beta = \sum_{k=1}^{\infty} r_k \alpha^k \tag{5.3}$$

Suppose that the distribution of K is geometric with parameter r, starting at $k = 2$; in other words, $r_k = (1-r)^{k-2} r$. Then the above expression becomes

$$\beta = \frac{\alpha^2 r}{1 - \alpha(1-r)} \tag{5.4}$$

Consider now the probability, q_i, that a transaction accessing an edge of type i arriving at time t and taking a time d to complete a write operation, will be involved in a conflict. That is the probability that another transaction accessing an edge of type i arrives between t and $t+d$ and writes the same edge, but starting at its other end. This can be expressed as

$$q_i = 1 - e^{-\frac{1}{2}\lambda p_i d / N_i}; i = 1, 2, ..., T \tag{5.5}$$

If the time to complete a distributed write is not constant, but is distributed exponentially with mean d, then the conflict probability would be

$$q_i = -\frac{\lambda p_i d}{2N_i + \lambda p_i d}; i = 1, 2, ..., T \tag{5.6}$$

When d is small, there is very little difference between these two expressions. An incoming transaction that is involved in a conflict would change the category of a distributed edge from 1 to 2, provided that all read operations of both queries return correct results. Hence, the instantaneous transition rate, $a_{i,1,2}$, from state $[n_{i,0}, n_{i,1}, n_{i,2}, n_{i,3}]$ to state $[n_{i,0}, n_{i,1} - 1, n_{i,2} + 1, n_{i,3}]$ can be written as

$$a_{i,1,2} = \frac{\lambda p_i n_{i,1}}{N_i} q_i \beta^2 \tag{5.7}$$

Conversely, an incoming transaction writing a category 2 edge can change it to a category 1 edge, provided that all its read operations return correct results and it is not involved in a conflict. Hence, the instantaneous transition rate, $a_{i,2,1}$, from state $[n_{i,0}, n_{i,1}, n_{i,2}, n_{i,3}]$ to state $[n_{i,0}, n_{i,1}+1, n_{i,2}-1, n_{i,3}(t)]$, is given by

$$a_{i,2,1} = \frac{\lambda p_i n_{i,2}}{N_i} (1-q_i)\beta \tag{5.8}$$

The other possible transitions convert an edge of category 0, 1 or 2 into an edge of category 3. This happens when a transaction writes after receiving an incorrect answer to at least one of its reads. Denoting the corresponding instantaneous transition rates by $a_{i,j,3}$, for $j = 0, 1, 2$, we have

$$a_{i,j,3} = \frac{\lambda p_i n_{i,j}}{N_i} (1-\beta) \tag{5.9}$$

Using these transition rates, one can simulate the process of corrupting the database and obtain both point estimates and confidence intervals for the average time to operational corruption, U_γ.

Delta Protocol Adaptation. The Delta protocol reduces the conflict probabilities $q_i, 1 \leq i \leq T$ and thus we now compute q_i^{new}. The new probabilities q_i^{new} are used in the model and simulations of [41] to estimate U_γ. The arrival time, a, of a conflicting transaction, T, is assumed exponentially distributed with rate ρ. Where $\rho = \frac{\lambda p_i}{2n_i}$ is the probability a given operation accesses the incorrect record of a half-corrupted edge of type i. Recall, the time interval, d, that elapses between transaction T completing its tentative write at one end of a distributed edge and starting at another end, is assumed exponentially distributed with rate μ.

Consider the interleaving in Figure 2.4(a) and assume T_x arrives at S_i at time 0. Then, T_x arrives at S_j after d_x. Assume T_y arrives at S_j at some time a. Then, T_y arrives at S_i at time $a+d_y$. Half-corruption occurs under the following conditions:

(i) At S_j, $d_x > a+\Delta$

(ii) At S_i, $a+d_y > \Delta$

The conflict probability, q_i^{new}, for edge type i is given by,

$$\begin{aligned} q_i^{new} &= P\left[(d_x > a+\Delta) \cap (d_y > \Delta - a)\right] \\ &= \int_0^{\Delta} \frac{\lambda p_i}{2n_i} e^{-\frac{\lambda p_i}{2n_i}a} e^{-\mu(\Delta+a)} e^{-\mu(\Delta-a)} da + \int_{\Delta}^{\infty} \frac{\lambda p_i}{2n_i} e^{-\frac{\lambda p_i}{2n_i}a} e^{-\mu(\Delta+a)} da \\ &= e^{-2a\mu} - \left(\frac{\mu}{\frac{\lambda p_i}{2n_i}+\mu}\right) e^{-(\frac{\lambda p_i}{2n_i}+2\mu)a} \end{aligned} \tag{5.10}$$

From the perspective of computing q_i^{new}, Figure 2.4(b) is equivalent to Figure 2.4(a) and the above expression holds. Then, q_i^{new} for each edge type can be used with the transition rates to simulate the process of corrupting the database under the Delta protocol, obtaining estimates of the average time to operational corruption, U_γ.

Number of aborts per second. To evaluate this metric for various values of Δ, a second simulation that focuses specifically on the subset of most frequently accessed distributed edges was performed.

Note that both metrics that we set out to evaluate will be influenced by several parameters that characterize the database and other aspects:

- *Database Size*. Size is expressed by the total number of edges N, and the fraction f of distributed edges.

- *Workload*. Measured as transactions per second (TPS). Significant for measuring U_γ are: the fraction of this load that writes after reads and the number of reads that precede a write.

- *Distributed Write Delays and Choosing* Δ. The smaller the delays the less likely the bound Δ is violated. Conversely, smaller Δ is the more likely the bound Δ is violated. We describe how to choose Δ below.

Choosing Δ. To choose to bound Δ consider the probability of the message delay exceeding Δ:

$$\begin{aligned} P[M > \Delta] &= 1 - e^{-\delta\Delta} \\ 1 - e^{-\delta\Delta} &= 1 - \varepsilon \\ e^{-\delta\Delta} &= \varepsilon \\ \Delta &= -\frac{\ln(\varepsilon)}{\delta} \end{aligned} \tag{5.11}$$

Substituting Δ into the above equation yields the conflict probability for a given ε. For example, $\varepsilon = 0.001$ gives a 0.1% probability the message delay exceeds Δ, for this $\Delta = 0.035s$,

5.2.4 Evaluation

The following parameter choices are inline with industry experiences. The graph analyzed consisted of seven edge types, $n_1 = 10^4, n_2 = 10^5, n_3 = 10^6, n_4 = 10^7, n_5 = 10^8, n_6 = 10^9, n_7 = 10^{10}$, totaling 11 billion edges, with access probabilities $p_1 = 0.5, p_2 = 0.25, p_3 = 0.13, p_4 = 0.06, p_5 = 0.03, p_6 = 0.02$ and $p_7 = 0.01$; a graph of this size would have approximately 1 billion vertices. The number of read operations before a write per transaction is

geometrically distributed starting at 2, with an average of 15. In all edge types, $f = 0.3$ are distributed, the remainder are local, in proportion with a good graph partitioning algorithm.

The delay d between a transaction completing a tentative write at one end and starting at another end is exponentially distributed with a mean of $5ms$. The database is initially clean and considered to be operationally corrupted when 10% ($\gamma = 0.1$) of all edges are semantically corrupted. The time taken until operational corruption, U, is measured in days. We consider a range of transaction arrival rates, $\lambda = (1000, ..., 10000)$; a typical graph workload comprises of 90% read-only transactions and 10% read-write transactions [4], hence the chosen range reflects a total workload $(10000, ..., 100000)$. The following Δ values were considered $\Delta = 50, 75, 100ms$. For each Δ the probability that d exceeds Δ is $P(d > \Delta) = 4.5^{-5}, 3.1 \times 10^{-7}, 2.1 \times 10^{-9}$ respectively.

The results for measuring the impact of Δ on the time until operational corruption are given in Figure 5.3 (where the y-axis is in log scale). With no concurrency control (U =NA), U ranges between 50-500 days. For $\Delta = 50ms$, U increases to 1-75 years. For $\Delta = 75, 100ms$ the time to corruption is significantly large as shown in Figure 5.3.

To evaluate the number of aborts that occurred per second (α), simulations were run for 10 seconds for each arrival rate, $\lambda = (1000, ..., 10000)$. Figure 5.4 reports the fraction $\frac{\alpha}{\lambda}$ for various values of λ. This fraction is also the probability that an incoming transaction is aborted due to the Delta protocol. For $\Delta = 50$ ms, the abort probability is between $1 - 5\%$, this increases to between $1 - 7\%$ for $\Delta = 75$ ms and $1 - 9\%$ for $\Delta = 100$ ms.

5.3 Deterministic Reciprocal Consistency Protocol

The Deterministic Reciprocal Consistency Protocol preserves Reciprocal Consistency in all eventualities, in other words, there is zero probability for the violation of Reciprocal Consistency. This is achieved through a mechanism, referred to as *collision detection*, that enforces Reciprocal Consistency for distributed updates.

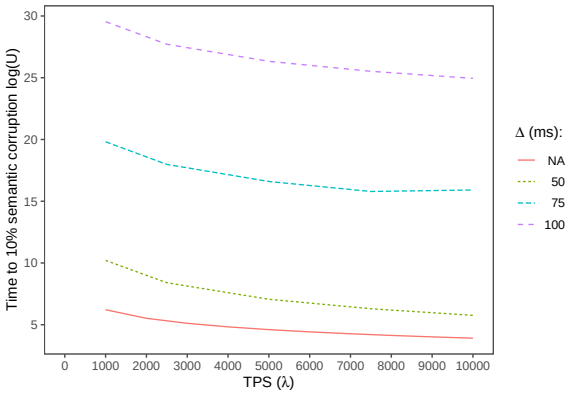


Fig. 5.3 Time until operational corruption ($\log U$ measured in days).

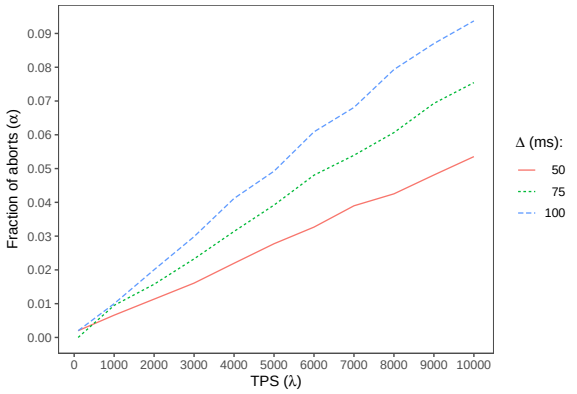


Fig. 5.4 Fraction of aborts.

Like many concurrency control protocols in the literature (e.g., [38, 66]), the Deterministic Reciprocal Consistency Protocol treats updates as being provisional initially. They become permanent only if, and when, the transaction that contains them is allowed to commit.

Provisional updates on a given record can occur only one at a time and each is time-stamped using the local server's clock. Thus, when a transaction attempts to update a given record, it can identify all other transactions, called *predecessors* (if any), that have earlier updated that record provisionally (similarly to the Wait-Hit Protocol in Chapter 3). Read operations receive the latest committed version of a record and ignore any provisionally updated values.

5.3.1 Protocol Description

Collision detection. Recall from Subsection 2.2.1 that an update operation by a transaction T_x for a distributed edge has a part 1, carried out at the first server visited (e.g., S_i in Figure 5.1), and part 2, performed at the second server (e.g., S_j in Figure 5.1) after a network delay. The corresponding provisionally updated records are now given *label 1* and *label 2*, respectively, and are associated with the id of that transaction.

These labels act as *history* meta-data indicating the server where a transaction started and completed its update. After performing part 1 at say S_i, T_x remembers the labels of all predecessors that have operated on the edge at that server before it. When arriving at S_j to perform part 2, T_x retrieves the labels of transactions (predecessors) that have operated on that edge at S_j before it. The following rule is applied at S_j:

Cancellation rule: If, by the time part 2 is performed, a previous provisional update labeled 1 has been observed for a predecessor, but the corresponding label 2 has not been observed, then this update is canceled. In other words, an update is canceled if it has observed the start of a previous attempt by a transaction, but not its completion. When an update is canceled, the transaction containing it is aborted and all its provisional records are erased.

According to the this rule, T_x in Figure 5.1(a) is canceled because it observes a predecessor T_y with label 1 in S_j, but has not observed T_y's label 2 in S_i. Whether T_y is canceled or not depends on whether T_x's provisional updates remain or have been erased by the time T_y performs part 2.

We now describe an extension to the Deterministic Reciprocal Consistency Protocol that additionally ensures Edge-Order Consistency (see Subsection 2.2.3) before developing an analytical model and evaluating the protocol's performance.

5.4 Deterministic Edge Consistency Protocol

The Deterministic Edge Consistency Protocol extends the Deterministic Reciprocal Consistency Protocol to ensure there is also zero probability for the violation of Edge-Order Consistency. Deterministic Edge Consistency Protocol employs two mechanisms: (i) Deterministic Reciprocal Consistency Protocol's *collision detection* to enforce Reciprocal Consistency for distributed updates, and (ii) *order arbitration* to enforce Edge-Order Consistency between transactions.

As before, the Deterministic Edge Consistency Protocol treats updates as being provisional initially and they become permanent only if, and when, the transaction that contains them is allowed to commit. Also, again provisional updates on a given record can occur only one at a time and each is time-stamped using the local server's clock. Hence, when updating a record, a transaction can identify all predecessors (if any), that have earlier updated that record provisionally. Read operations receive the latest committed version of a record and ignore any provisionally updated values.

5.4.1 Protocol Description

For a transaction, collision detection is applied at every update it executes and may trigger an immediate abort. Transactions that survive collision detection may, if necessary, go to order arbitration. The latter may also result in an abort.

Order arbitration. The purpose of this mechanism is to detect and prevent edge-order inconsistencies between transactions. It only applies to transactions that contain more than one distributed update. Those with a single update that have not been aborted by collision detection are allowed to commit and depart.

Using the records relating to provisional updates, each multi-update transaction maintains a *predecessor list* containing all predecessor transactions it encountered during its provisional updates. If that list is empty when the transaction successfully completes all its provisional updates, then it commits and departs. Otherwise it goes to arbitration.

The arbiter is a special service, assumed here to have been implemented in a dedicated server. A list called the *hit-list* is maintained. It contains transactions which, if allowed to commit, risk violating edge-order consistency. Transactions arriving for arbitration join a queue and are served in order of arrival. If the transaction at the head of the queue is not present in the hit-list, it commits. All transactions in its predecessor list are added to the hit list if not already there. If it is in the hit list, it aborts and all its provisional updates are

erased. What has happened in this case is an overtaking: the current transaction was named as a predecessor by a transaction that committed earlier.

Note that order arbitration is a simple version of the Wait-Hit Protocol in Chapter 3. Additionally, it is also possible here to adopt a pattern similar to the Distributed Wait-Hit Protocol from Section 3.6 and execute a validation and commit phase, instead here we opt for a centralized arbiter.

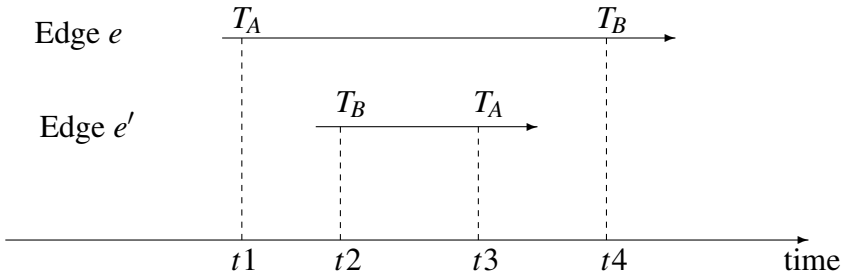


Fig. 5.5 Edge-Order Consistency violation

We can informally argue that our approach is correct by considering the edge-order inconsistency depicted in Figure 2.6 (reproduced from Figure 2.6 for convenience). It can be seen that T_A will have T_B in its predecessor list while updating e', and T_B will observe T_A as a predecessor while updating e. Both T_A and T_B must approach the arbiter because they update more than one edge and have a non-empty predecessor list. If the first transaction to be processed by the arbiter is allowed to commit, the second one will be entered in the hit list and will abort. Thus only one of T_A and T_B, but not both, can commit and edge order inconsistency is always avoided.

Note that this approach to arbitration is pessimistic. It aborts a transaction as soon as it detects a risk of edge-order violation, even though the actual violation may not occur. Consequently, some transactions are aborted unnecessarily, just because they are overtaken by their successors. To eliminate unwarranted aborts, the arbiter would have to keep much more detailed information about the updates performed by all transactions, and would have to do considerably more processing.

We now proceed to the task of evaluating certain performance measures, such as the average number of transactions that are aborted per unit time, the offered load at the arbiter, and the average time a transaction remains in the system. Since the processes involved are rather complex, such an evaluation will inevitably entail approximations. That, in turn, will necessitate an assessment of the accuracy of those approximations.

5.4.2 Approximate Model

We are concerned with updates performed on distributed edges in a distributed graph database (i.e., edges whose source and destination nodes are stored on different servers). These edges are divided into T types, numbered 1, 2, ..., T. The number of edges of type i is N_i, and the probability that an update operation is aimed at an edge of type i is p_i. All edges of a given type are equally likely to be addressed, so that the probability of accessing a particular edge of type i is p_i/N_i.

Transactions arrive into the system in a Poisson stream, at the rate of λ per second. Each transaction performs a random number, K, of updates for different distributed edges. The distribution of K is arbitrary: $P(K=k)=r_k$ ($k=1,2,\ldots$). The average number of updates per transaction is κ. Thus, the arrival rate of updates at a *particular* distributed edge of type i, ξ_i, is equal to

$$\xi_i = \frac{\kappa\lambda p_i}{N_i} \;\; ; \;\; i=1,2,\ldots,T \; . \tag{5.12}$$

The first approximation is to assume that the arrival process of updates for a particular distributed edge of type i is Poisson with rate ξ_i.

We wish to estimate the probability, u_i, that a transaction, T_x, containing an update for an edge of type i, is cancelled due to a collision with another transaction, T_y, containing an update for the same edge. That is, either T_y arrives in the opposite server during the network delay of T_x (Figure 2.4 case (a)), or T_x arrives in the opposite server during the network

delay of T_y (case (b)), or T_y arrives in the same server during the network delay of T_x *and* its network delay completes before that of T_x (case (c)).

Assume that the network delays are i.i.d random variables distributed exponentially with parameter δ (mean $1/\delta$). This may or may not be an approximation.

Updates for a particular distributed edge of type i arrive in one of the two servers involved in storing the edge at rate $\xi_i/2$. Moreover, a given network delay completes before another with probability 1/2. Hence, we can estimate the probabilities of cases (a), (b), and (c), $u_i^{(a)}$, $u_i^{(b)}$ and $u_i^{(c)}$, as

$$u_i^{(a)} = u_i^{(b)} = \frac{\xi_i}{\xi_i + 2\delta} \;\; ; \;\; u_i^{(c)} = \frac{1}{2}\frac{\xi_i}{\xi_i + 2\delta} \;\; ; \;\; i = 1, 2, \ldots, T \; , \tag{5.13}$$

where ξ_i is given by (Equation (5.12)) and δ is the parameter of the network delay. The overall probability, u_i, that at least one of those events will happen, is

$$u_i = 1 - (1 - u_i^{(a)})(1 - u_i^{(b)})(1 - u_i^{(c)}) \approx \frac{2.5\xi_i}{\xi_i + 2\delta} \;\; ; \;\; i = 1, 2, \ldots, T \; . \tag{5.14}$$

The last approximation in the right-hand side holds when the rate ξ_i is small compared to δ.

The unconditional probability, u, that an arbitrary update is cancelled by the collision detection mechanism, is given by

$$u = \sum_{i=1}^{T} p_i u_i \; . \tag{5.15}$$

The probability, v_k, that a transaction containing k updates is aborted because one of them is involved in a collision, is equal to

$$v_k = 1 - (1 - u)^k \; , \tag{5.16}$$

and the unconditional probability, v, that a transaction is aborted due to a collision is given by

$$v = \sum_{k=1}^{\infty} r_k v_k \ . \tag{5.17}$$

Now consider the average run time, a_k, of a transaction that contains k update operations. Assume that each update takes time b, on the average. Those times include read operations and computations, as well as network delays. If the first $j-1$ provisional updates are completed successfully but the j-th update is cancelled as a result of a collision, then the average run time would be jb. Hence, a_k is given by

$$a_k = \sum_{j=1}^{k} jb(1-u)^{j-1}u + kb(1-u)^k \ , \tag{5.18}$$

where u is given by (Equation (5.15))

With a little manipulation, this expression can be simplified to

$$a_k = b\sum_{j=1}^{k}(1-u)^{j-1} = b\frac{1-(1-u)^k}{u} \ . \tag{5.19}$$

The unconditional average run time of a transaction, a, is equal to

$$a = \sum_{k=1}^{\infty} r_k a_k \ . \tag{5.20}$$

If all provisional updates in a transaction are completed successfully, and if either there was only one update, or there were no predecessors, then the transaction commits. Otherwise it goes to the arbiter. The time a transaction spends queueing and being served by the arbiter will be referred to as the *arbitration time*.

Assume (this is another approximation) that each transaction joins the arbiter queue with probability α, independently of the others. That is, the arrival process is Poisson, with rate

$\lambda\alpha$. The arbiter's average service time, s, is a given parameter. Thus the offered load at the arbiter is $\rho = \lambda\alpha s$.

Treating the arbiter as an $M/M/1$ queue, we estimate the average arbitration time, w, as

$$w = \frac{s}{1-\rho}\,, \tag{5.21}$$

provided that $\rho < 1$. If $\rho \geq 1$, then $w = \infty$. The total average time that a transaction spends in the system is

$$W = a + \alpha w\,, \tag{5.22}$$

where a is given by (Equation (5.20)).

We shall now develop an iterative fixed-point approximation for α. Denote by $d_{j,k}$ the average lifetime of the j-th update within a transaction containing k updates, *excluding* any possible arbitration time. By an argument similar to the one that led to (Equation (5.19)), we obtain

$$d_{j,k} = b\sum_{i=1}^{k+1-j}(1-u)^{i-1} = b\frac{1-(1-u)^{k+1-j}}{u}\,. \tag{5.23}$$

The lifetime of a randomly chosen update within a transaction containing k updates, d_k (again excluding arbitration), is given by

$$d_k = \frac{1}{k}\sum_{j=1}^{k} d_{j,k} = b\frac{(k+1)u + (1-u)^{k+1} - 1}{ku^2}\,. \tag{5.24}$$

Hence, the total average time spent in the system by an arbitrary update (*including* the arbitration time), d, is equal to

$$d = \sum_{k=1}^{\infty} r_k d_k + \alpha w\,, \tag{5.25}$$

where w is given by (Equation (5.21)).

Now, let γ_i be the probability that an update of type i has a predecessor, i.e., the probability that such an update arrives while a preceding update for the same edge is still in the system.

Assuming that the update residence times are distributed exponentially with mean d given by (Equation (5.25)), this can be approximated as

$$\gamma_i = \frac{\xi_i d}{1 + \xi_i d} , \tag{5.26}$$

where ξ_i is given by (Equation (5.12)).

The unconditional probability, γ, that an arbitrary update has a predecessor, is

$$\gamma = \sum_{i=1}^{T} p_i \gamma_i . \tag{5.27}$$

If a transaction contains k updates, the probability that at least one of them has a predecessor, α_k, is

$$\alpha_k = 1 - (1 - \gamma)^k . \tag{5.28}$$

Remembering that a transaction goes to the arbiter if it has more than one update *and* all updates avoid collisions *and* at least one of them has a predecessor, we write

$$\alpha = \sum_{k=2}^{\infty} r_k (1 - u)^k \alpha_k . \tag{5.29}$$

Note that the right-hand side of (Equation (5.29)) depends on α, via (Equation (5.25)) and (Equation (5.26)). In other words, we have a fixed-point equation of the form

$$\alpha = f(\alpha) . \tag{5.30}$$

This can be solved by a simple iterative scheme. Start with an initial guess, α_0, say $\alpha_0 = 0$. At iteration n, compute

$$\alpha_n = f(\alpha_{n-1}) , \tag{5.31}$$

stopping when two consecutive iterations are sufficiently close to each other.

The probability α allows us to evaluate the offered load at the arbiter queue, and hence estimate the average response time of a transaction, W. Another important performance measure is the rate of aborts, R, i.e. the average number of transactions that are aborted per unit time. Note that a transaction may be aborted due to a collision, with probability v given by (Equation (5.17)), or it may be aborted because it finds itself on the arbiter's hit list. Denoting the probability of the latter occurrence by β, we can write

$$R = \lambda[v + (1-v)\beta] \,. \tag{5.32}$$

To find an expression for the probability β, note that a transaction, A, is aborted by the arbiter if (i) A goes to the arbiter and (ii) another successfully committing transaction, B, which arrived at the arbiter before A, had A in its list of predecessors (A would then have been added to the hit list). That is, B arrives in the system during the run time of A, tries to update one of the edges that A has updated, completes before A, goes to the arbiter and is allowed to commit.

Suppose that A contains k updates, and let t be the instant when the j-th of those updates is attempted. The average interval from t until the completion of A, given that all updates succeed, is $(k+1-j)b$. If the j-th update is of type i, let h_i be the average interval from t until the completion of the next transaction, B, that updates the same edge and then goes to the arbiter. That average can be estimated as

$$h_i = \frac{1}{\xi_i \alpha} + \frac{\kappa - r_1}{2(1-r_1)} b \,, \tag{5.33}$$

where α is given by (Equation (5.29)). The multiplier of b in the right-hand side is half of the average number of updates in a transaction, given that there are more than one.

Denote by β_{ijk} the probability that B arrives after the j-th update out of the k in A, and completes before A, and A goes to the arbiter but is aborted because B commits, given that

the j-th update is of type i. We write

$$\beta_{ijk} = \alpha(1-\beta)\frac{(k+1-j)b}{h_i+(k+1-j)b} \,. \tag{5.34}$$

Removing the conditioning on the type of update, we get the probability, β_{jk}, that A is aborted by the arbiter due to the j-th of its k updates:

$$\beta_{jk} = \sum_{i=1}^{T} \beta_{ijk} p_i \,. \tag{5.35}$$

The probability, β_k, that at least one of the k updates will cause A to be aborted, is

$$\beta_k = 1 - \prod_{j=1}^{k} (1-\beta_{jk}) \,. \tag{5.36}$$

Finally, the unconditional probability, β, that an arbitrary transaction is aborted by the arbiter, can be expressed as

$$\beta = \sum_{k=2}^{\infty} \beta_k r_k \,. \tag{5.37}$$

The right-hand side of this equation depends on α, which has already been computed, and also on β. Thus, we have another fixed-point equation which can be be solved by an iterative procedure of the type (Equation (5.31)).

One might wish to measure the performance of the system by a cost function of the form

$$C = c_1 W + c_2 R \,, \tag{5.38}$$

where c_1 and c_2 are some coefficients reflecting the relative importance given to the average response time and number of aborts. There are trade-offs that may need to be controlled. If, for example, the arbiter is overloaded, leading to large or infinite response times, a 'voluntary abort' policy may be introduced. If a transaction cannot commit upon completion (because

its predecessor list is non-empty), it tosses a biased coin and, with probability σ, aborts instead of going to the arbiter. The offered load at the arbiter queue would then be reduced to $\rho = \lambda\alpha(1-\sigma)s$. The optimal value of σ would be chosen so as to minimize the cost function C.

5.4.3 Evaluation

The purpose of this section is to assess the accuracy of the model estimates by comparing them with simulations. In order to reduce the number of parameters to be set, we focus on the smallest and most frequently accessed class of edges, ignoring the larger classes where conflicts are very unlikely to occur. The examples we have chosen contain a single class with N distributed edges, each of which is equally likely to be the target of an update. The size and traffic parameters are typical of a large scale-free graph database (see also [119]).

In the first example, N is varied between 5000 and 25000 edges. The arrival rate is fixed at $\lambda = 1000$ transactions per second. The average network delay is assumed to be $5ms$ (i.e., $\delta = 200$). That is also the value of b (the average time per update). The distribution of the number of updates in a transaction is geometric, with mean $\kappa = 5$. The average arbiter service time is $s = 0.01$ and that value will be kept fixed in the following examples.

In Figure 5.6, the total average number, R, of transaction aborted per unit time by the collision detection and by the order arbitration parts of the protocol, is plotted against the number of edges. The estimated points are computed by the algorithm described in Subsection 5.4.2, while each simulated point represents the result of a simulation run where one million transactions pass through the system.

Intuitively, we expect that when the number of edges increases, there will be fewer collisions and instances of overtaking, and therefore fewer aborts. Indeed, that is what is observed. The model consistently underestimates the number of aborts, but the relative errors are not large. They vary from 9% at $N = 5000$ to 5% at $N = 25000$. That underestimation is

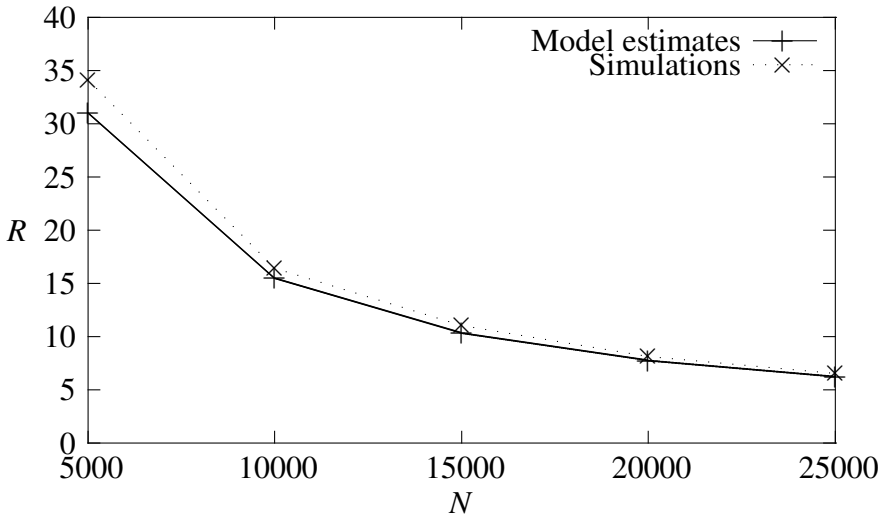


Fig. 5.6 Abort rate as a function of N
$\lambda = 1000$, $\kappa = 5$, $\delta = 200$, $b = 0.005$, $s = 0.01$

probably caused by the simplifying assumptions used in deriving the approximate estimates. On the other hand, the times taken to produce the two plots were vastly different: the model plot took a small fraction of a second to compute, while the simulation runs were several orders of magnitude slower.

From now on, the number of edges will be fixed at $N = 10000$ and the effect of different parameters will be explored. In the second example, the arrival rate λ is varied between 700 and 1200 transactions per second, while the other parameters are kept as before.

In Figure 5.7, the average number of aborted transactions per second, R, is plotted against the arrival rate λ, using both the model approximation and simulations. Each simulated point is again the result of a run where one million transactions pass through the system. Once more, we observe that the model slightly under-estimates the values of R, but the relative errors are quite small; they are of the order of 6% or less, over the entire range.

The average response time of a transaction, W, was about $25ms$; its value changed very little over this range of arrival rates.

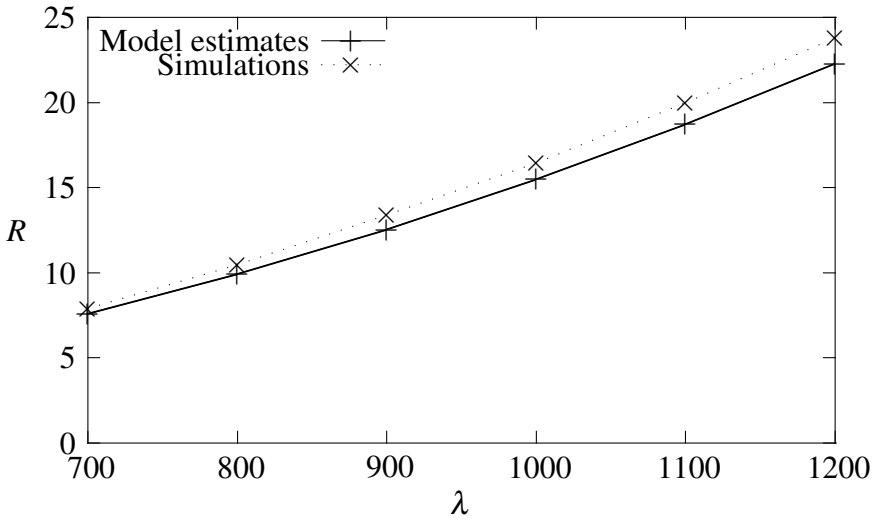


Fig. 5.7 Abort rate as a function of λ
$\kappa = 5$, $\delta = 200$, $b = 0.005$, $s = 0.01$

For these parameter values, the model predicts that the arbiter queue becomes unstable when the arrival rate is about $\lambda = 1500$. The simulation agrees. The observed rate at which transactions join the arbiter queue exceeds the service rate, $\mu = 100$, for that value of λ.

For the next experiment, the average network delay is doubled to $10ms$, $\delta = 100$. Intuitively, this should have the effect of increasing the rate at which transactions are aborted, and also should increase the offered load at the arbiter queue.

Figure 5.8 confirms our intuition. The relative errors of the model estimates are still quite low, of the order of 9% or less. The arrival rate is now varied between $\lambda = 600$ and $\lambda = 1000$. Both the model and the simulation agree that the arbiter queue becomes unstable when $\lambda = 1100$.

In the fourth experiment, the network delay is back to $5ms$, but the number of updates in a transaction, K, has a different distribution and mean. The assumption now is that K is uniformly distributed on the range [1,19], with a mean of 10. The results are illustrated in Figure 5.9.

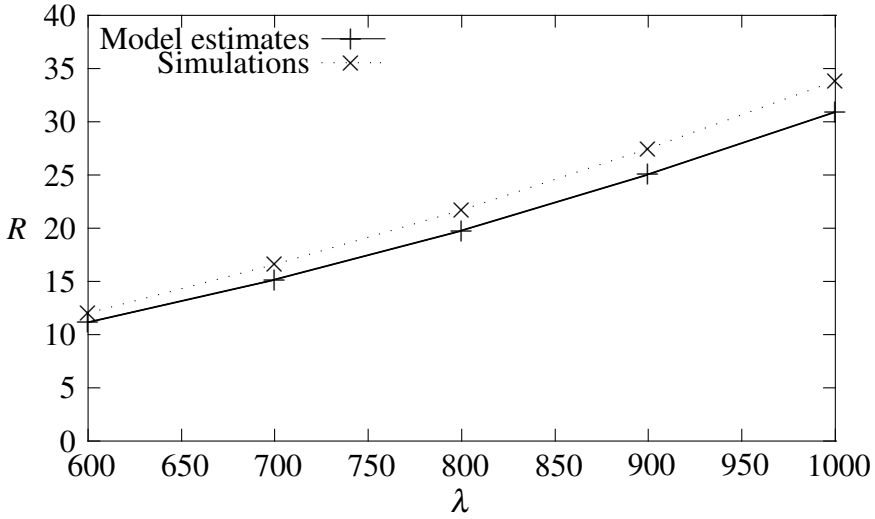


Fig. 5.8 Larger network delays
$\kappa = 5$, $\delta = 100$, $b = 0.01$, $s = 0.01$

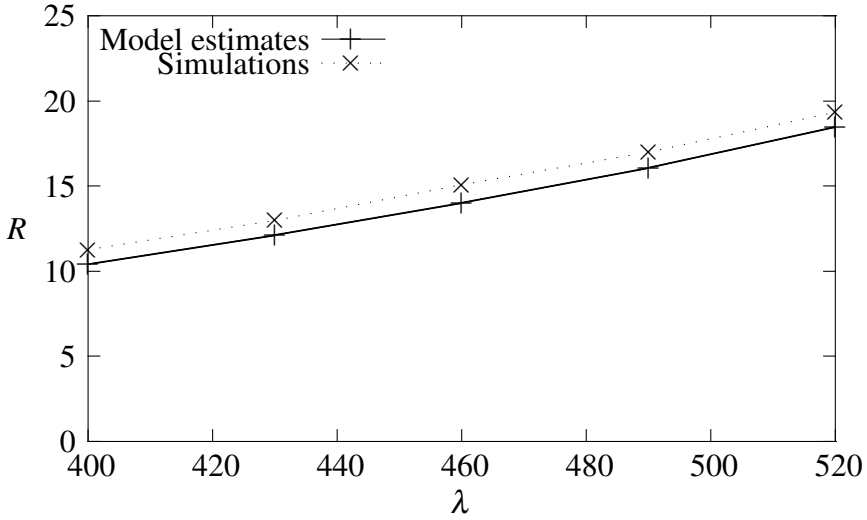


Fig. 5.9 Different distribution of updates
$\kappa = 10$, $\delta = 200$, $b = 0.005$, $s = 0.01$

The larger number of updates per transaction leads to both higher likelihood of collisions and more visits to the arbiter. The saturation point for the arbiter queue is now a little below $\lambda = 550$. As Figure 5.9 illustrates, the model approximation is still accurate, with relative errors of the order of 8% or less.

It is perhaps worth noting that in the last three examples, the rate of aborts increases roughly linearly with λ. For all arrival rates in example 2, between 1% and 2% of the incoming transactions are aborted. In example 3 that fraction is between 2% and 3%, while in example 4 it is between 3% and 4%.

5.5 Conclusion

Database concurrency control has been a long researched area, however to the best of our knowledge this is first attempt at developing protocols specific for distributed graph databases.

The Delta protocol is a lightweight protocol for providing Reciprocal Consistency and mitigating the problem of high contention in a distributed graph database. It leverages the fact that writes to distributed edges always consist of two sequential writes to entries in the adjacency lists of vertices that are connected by the edge. Since it is concerned only with edges (and not vertices) in a graph, it provides guarantees weaker than Read Uncommitted isolation (the weakest ANSI isolation level). The Delta protocol, we believe, is valuable in practice given the popularity of NoSQL distributed graph databases and the rate at which semantic corruption can spread if Reciprocal Consistency is left unchecked. Simulations indicate that when Δ values are chosen to be reasonably large, the protocol rules out corruption resulting from half-corrupted distributed edges while keeping the abort rate considerably small.

The Deterministic Edge Consistency Protocol is comprised of two distinct mechanisms: collision detection from Deterministic Reciprocal Consistency Protocol and order arbitration between transactions to provide Reciprocal Consistency and Edge-Order Consistency. The order arbitration mechanism is built on the same principles as the Wait-Hit Protocol from Chapter 3. To evaluate the protocol's impact on system performance, in terms of aborted transactions and load on the arbiter an approximate model was developed and solved. It provides estimates for the average number of transactions that are aborted per unit time, the

probability that a transaction will need to go to arbitration, and the average response time of a transaction. The accuracy of the solution was examined by comparisons with simulations and was found to be very high under a variety of parameter settings.

5.5.1 Further Work

Complex Constraints. The current generation of GDBMSs generally does not support constraints much more complex than Reciprocal Consistency; sometimes domain and primary key constraints (in case indexes are supported). In the future, it is anticipated that GDBMSs will evolve to support complex constraints. Beyond equivalents of the relational ones, graph databases might introduce graph-specific constraints, such as (partial) compliance to a schema formulated on top of property graphs, rules that guide the presence of labels or structural graph constraints such as connectedness of the graph, absence of cycles, or arbitrary well-formedness constraints [94]. Each poses an interesting challenge to database architects on how to ensure performance whilst maintaining correctness.

Empirical Performance Evaluation. The protocols can be implemented in a distributed graph database to assess the validity of the simulations and to enable comparisons with other concurrency control protocols.

Stronger Isolation. Moreover, we plan on investigating the suitability of higher isolation levels in distributed graph databases, Read Atomic isolation [9] seems particularly well suited and has been implemented in Facebook's graph store TAO [18]. Similar to the edge-order inconsistency examined here, there may also be node-order inconsistency, occurring when transactions interfere while updating the same set of nodes. Eliminating node-order inconsistencies will be addressed in future work. It is well known in the database literature that there is a hierarchy of approaches which achieve various degrees of isolation (see [2]). Selecting a level for a given application typically involves a trade-off between consistency

requirements and performance, which levels are suitable for distributed graph databases without introducing significant overheads or violating graph integrity remains an open question. For example, it is not uncommon for distributed graph transactions to access multiple partitions (often 5+ [17]) and it was established in [54] athat s the number of partitions accessed by MPTs increases performance severely degrades. Potentially, this makes Serializable isolation unfeasible. Another interesting direction is incorporating awareness of application-level invariants and types of common graph operations into the protocols to determine which can be preserved without coordination and which require it [10]. For example, concurrent edge insertion between two nodes is commutative and does not require coordination.

Chapter 6

A Performance Study of Epoch-based Commit Protocols

Summary

Distributed OLTP systems execute the high-overhead 2PC protocol at the end of every distributed transaction. Epoch-based commit proposes that 2PC be executed only once for all transactions processed within a time interval called an epoch. Increasing epoch duration allows more transactions to be processed before the common 2PC. It thus reduces 2PC overhead per transaction, increases throughput but also increases average transaction latency. Therefore, required is the ability to choose the right epoch size that offers the desired trade-off between throughput and latency. To this end, this chapter develops two analytical models to estimate throughput and average latency in terms of epoch size taking into account load and failure conditions. Simulations affirm their accuracy and effectiveness. This chapter then presents epoch-based multi-commit which, unlike epoch-based commit, seeks to avoid all transactions being aborted when failures occur, and also performs identically when failures do not occur. Our performance study identifies workload factors that make it more effective in preventing transaction aborts and concludes that the analytical models can be equally useful in predicting its performance as well.

6.1 Introduction

When a single-node database reaches its capacity limits, a common option is to partition data across multiple nodes forming a distributed database [97]. High levels of scalability ensue when transactions access data within a single node, there is no need for any coordination, and hence communication, between nodes. However, when workloads contain distributed transactions accessing data from multiple nodes, an atomic commitment protocol, typically (2PC) [12], needs to be executed so that distributed transactions are guaranteed of atomicity and durability when nodes are prone to failures.

As discussed in Subsection 2.1.5, executing 2PC generally extracts a high performance cost [52, 54]. It involves two sequential network round trips, one for each of its phases, and two sequential durable writes. Depending on hardware deployment, when 2PC is executed at the end of each distributed transaction, it increases the transaction latency by an additional delay of up to 10*ms*. In addition, as 2PC execution prolongs the lifetime of a distributed transaction within the database, the potential for data contention among transactions (distributed or otherwise) intensifies which, in turn, leads to further performance degradation. For these reasons, a significant strand of distributed transactions research focuses on minimizing the costs inflicted by 2PC executions (see Subsection 2.1.6).

A recent practical proposal is *epoch-based commit* [72]. Based on the widely-held notion that node failures are getting less frequent with modern hardware, it takes the view that 2PC need not be executed at the end of every distributed transaction. Instead, 2PC is executed once for all transactions that arrive and get processed within a time interval called an *epoch*. Thus, an epoch is the base unit for doing 2PC and all transactions processed within it either commit or abort through one common 2PC execution. (This idea is a distributed extension of *group-commit* proposed in [28] to reduce disk I/O latency for single-node databases.)

The cost of one 2PC execution is thus amortized over multiple transactions processed within an epoch. Moreover, transactions whose processing is completed can release their locks instead of waiting until they commit at the epoch end; this reduces scope for contention. They can also *asynchronously* log their writes to persistent storage. The effects of all these features are two-fold. On the up side, throughput increases substantially – by four times (4x) as per the experiments conducted in [72]. On the down side, a transaction's results cannot be released until the epoch end when all its writes have become durable. This means that the earlier a transaction arrives during an epoch, the longer it waits before its results can be released; i.e., the average latency of transactions increases.

It is important to note that amortization of 2PC cost over multiple transactions also tends to increase the throughput as epoch size increases, when nodes do not fail. However, if a node fails, then the number of transactions aborted at the end of an epoch (which is when that failure is globally detected) will be large if the epoch was chosen to be long; this wasted work obviously reduces throughput. Thus, the possibility of node failures favors shorter epochs. In fact, there is an optimum epoch length at which throughput is maximized which can be used if increased latency is a minor concern. (Increased latency is acceptable for many workloads, see [21, 80, 110].) On the other hand, a user may want a reasonably high throughput as well as an acceptably moderate latency. Such a trade-off requires the means to choose appropriate epoch length. These requirements for adopting epoch-based commit in clusters were left as future work in the original paper [72] and are being fully addressed here.

We derive analytical solutions for estimating throughput and average latency in terms of epoch length and some system and load parameters. Estimating average latency accounts for aborted transactions being completed in subsequent epochs. Our derivations make certain simplifying assumptions, such as times to node failures and recovery are exponentially distributed and the cluster has at most one failed node at any time which holds if failures are independent and less frequent and if a failed node recovers before another node fails.

The protocol of [72] proposes that all transactions executed during an epoch be aborted in case of a node failure. This assumes that each node has directly or indirectly accessed the failed node at some time during the epoch and therefore all transactions it executed accessed some data now lost due to failure. We examine and find this assumption to be overly pessimistic and develop an *epoch-based multi-commit* version. Each node maintains a list of nodes it interacted with, and uploads the list to the 2PC coordinator. The latter then constructs disjoint *commit groups* such that a node within a given commit group would have interacted, directly or transitively, only with other nodes in that group during the epoch. In

case of a node failure, only those nodes in the commit group containing the failed node will abort their transactions and the rest will commit the transactions they processed.

It is easy to see that when a workload contains no distributed transaction, each commit group will have just one node and all operative nodes can commit their transactions in the event of a failure. At the other extreme, if the workload has many distributed transactions that cause every node to interact with every other node during an epoch, then there is only one commit group and a failure will cause all operative nodes to abort all their transactions as in the epoch-based commit protocol. Thus, our multi-commit version can automatically take advantage of favorable workload conditions in the event of a failure and avoid excessive aborts, while performing identically to the original version during failure-free epochs.

This chapter makes three major contributions. Analytical models for estimating throughput and average latency are developed in Section 6.2 for the epoch-based commit protocol. They allow an appropriate epoch size to be judiciously chosen for maximum throughput or minimum latency or seeking a trade-off between the two. Secondly, the epoch-based multi-commit protocol is presented in Section 6.3, together with a popular benchmarking case study carried out to support its core design rationale. Finally, a range of simulation experiments involving a cluster of 64 nodes operating for a 100-day period are carried out. They (i) demonstrate the accuracy and efficacy of the models of Section 6.2 for choosing the appropriate epoch size, (ii) point to workload features that can aid the multi-commit protocol in minimizing aborts when failures occur, and (iii) affirm the effectiveness of the models in selecting the right epoch size also for multi-commit version. Section 6.4 presents the strategies adopted for performance study, followed by the presentation and discussion of results in Section 6.5. Section 6.6 concludes the chapter and summarizes future work.

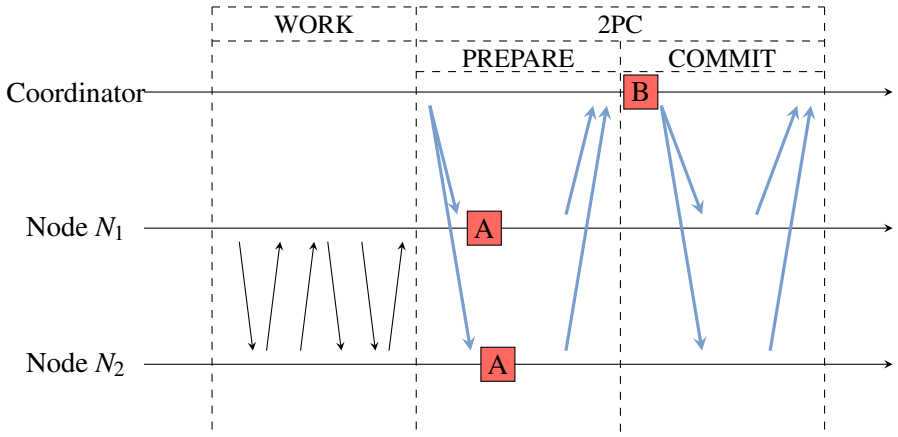


Fig. 6.1 Work and 2PC intervals of a cycle in the epoch-based commit protocol.

6.2 Analytical Models for Epoch-Based Commit

6.2.1 Protocol Description

The epoch-based commit protocol is briefly described before presenting analytical solutions for estimating protocol performance. For greater detail on the protocol, readers are referred to [72].

Consider a distributed system consisting of a *coordinator* node and $N, N > 1$ *participant* nodes that are simply called *nodes*. A large OLTP database is partitioned among the latter which execute transactions and 2PC under the control of the coordinator. A (participant) node can fail in a *fail-stop* manner: it functions correctly when operative and fails only by ceasing to be operative. The coordinator, however, is assumed to be built reliably and never fails. More precisely, we assume that the coordinator is internally replicated (primary-backup or state machine replication using atomic broadcast such as [82] or [68]), maintaining a single server abstraction.

The coordinator starts a *cycle* by starting an epoch timer for an interval of a and instructs the nodes to work on transactions. The epoch is referred to as *work* interval in Figure 6.1 during which transactions are executed but not made durable and their results also not released to end-users. A transaction will release the locks it holds at the end of its execution

even though it is not committed. This reduces lock contention but makes the effects of earlier transactions visible to the later ones. This is not a concern as all transactions of a given work interval commit or abort together at the end.

At the end of epoch timer *a*, the coordinator calls for an execution of 2PC which, as shown in Figure 6.1, has two phases: *prepare* and *commit*. In the prepare phase, the coordinator sends a `Prepare` message[1] to each participant node. (Messages exchanged during 2PC are shown by blue arrows in Figure 6.1.)

When a node receives `Prepare`, it force-logs: (i) ids of all ready-to-commit transactions it executed, and (ii) current epoch number. (These durable writes by participants are indicated in Figure 6.1 by red squares labeled A.) It then responds with `Prepare-Ack` to the coordinator. It is expected that the underlying concurrency control protocol has durably logged the individual writes by the ready-to-commit transactions *before* the prepared write record is logged. Note, some transactions executed within the epoch may have aborted earlier due to conflicts and violation of application-level integrity constraints. These aborted transactions do not cause the whole epoch to fail, but the result of them is still not released until the epoch terminates.

In the commit phase, the coordinator collects responses from participant nodes. If any node has not replied with a `Prepare-Ack` message, *all* nodes are instructed to abort all transactions they executed during the work interval. Else, the coordinator force-logs a *commit* record with the current epoch number (shown as a red square labeled B in Figure 6.1) and sends a `Commit` message to all participant nodes. When a node receives a `Commit` message, it commits the transactions it executed and releases the results to clients. It then sends `Commit-Ack` to the coordinator and awaits the latter to start the next cycle of work and 2PC intervals.

In summary, multiple transactions are executed during the work interval of each cycle; they are committed (or aborted) in a common 2PC execution. The design rationale behind

[1] Message types are indicated using mono-spaced font.

this approach is that the time taken to execute even a distributed transaction is negligibly small compared to the mean time before failure (MTBF) of nodes. So, the probability of several transactions being executed without encountering any node failure is fairly high. When the common 2PC execution leads to commit, its overhead per committed transaction becomes very small which, in turn, leads to an increased throughput.

Several studies [15, 48, 111] analyzing node failures in clusters confirm the assumption of MTBF being considerably larger than transaction processing times. For example, Garraghan et al. [48] analyze Google Cloud Platform traces and find 5056 out of 12532 nodes exhibiting 8954 failure events over a 29 day period. A node's MTBF turns out to be 12.71 hours! Another metric analyzed in [48] is also relevant for our analytical models: the mean time to repair (MTTR) is around 6-30 times smaller than MTBF. So, a failed node is most likely to have recovered well before another failure occurs.

6.2.2 Modeling Assumptions

We make two assumptions in our analytical modeling and derivation of expressions for estimating maximum throughput and average latency.

Assumption 1: A failed node recovers before another node in the system fails; i.e., between any two consecutive node failures, there is a recovery event of the first failed node.

A common observation in the literature (e.g., [15, 48, 111]) is that MTBF of nodes is much larger compared to their MTTR: a failed node is almost certain to be repaired before the next failure occurs. This near certainty is here assumed to be a certainty for the number of nodes typically found in a distributed OLTP system. Thus, there is at most one failed node in the system at any time.

Assumption 2: As soon as a node receives a `Prepare` message from the coordinator, it *instantly* completes any ongoing transaction execution and enters the 2PC execution.

In reality, some non-zero amount of time will elapse for a node to complete its ongoing execution (if any) and then respond with `Prepare-Ack`. This assumption can lead to an overestimation of throughput.

Our simulations do not have these assumptions and therefore will assess the accuracy of the expressions derived.

6.2.3 Maximum Throughput

We derive an analytical expression for estimating maximum attainable throughput by assuming that a node always has a transaction to execute and is never idle.

A participant node fails at exponentially distributed intervals with a low rate, ξ. The repair times are also distributed exponentially with a higher rate $\eta >> \xi$. Recall that nodes go through cycles, each consisting of a *work* interval, U, followed by a random 2PC interval V in which 2PC executed. To start with, we will regard U as random (as opposed to being fixed as a constant a as explained earlier). Denote the probability density functions of U and V by $f_U(x)$ and $f_V(x)$, respectively. Let also $f_W(x)$ be the convolution of $f_U(x)$ and $f_V(x)$, i.e., the p.d.f. of the sum $U+V$.

During a work interval, transactions are served at rate $N\mu$ (transactions per unit time), if all nodes are operative, and at rate $(N-1)\mu$ if one of them has failed. At the end of a work and 2PC cycle, either the commit operation completes successfully and all transactions executed during the work interval U depart from the system, or a node failure has occurred and all transactions executed during U are aborted and remain in the system for re-execution.

The probability, α, that N operative nodes complete one cycle successfully (i.e., with none of them failing), is

$$\alpha = \int_0^\infty f_W(x) e^{-N\xi x} dx = \tilde{w}(N\xi) = \tilde{u}(N\xi)\tilde{v}(N\xi)\,, \tag{6.1}$$

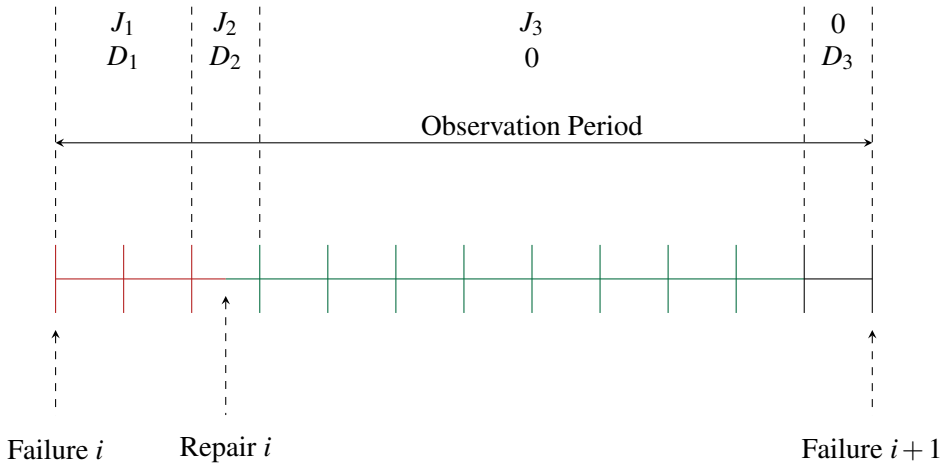


Fig. 6.2 Observation period with full cycles having $N-1$ operative nodes, $N-1$ and N operative nodes, and N operative nodes.

where $\tilde{u}(s)$, $\tilde{v}(s)$ and $\tilde{w}(s)$ are the Laplace transforms of $f_U(x)$, $f_V(x)$ and $f_W(x)$, respectively.

Consider an interval between two consecutive node failures, to be referred to as the *observation period*, indicated in Figure 6.2. The probability that exactly m consecutive work and 2PC cycles are completed successfully during the observation period is

$$p_m = \alpha^m(1-\alpha)\ . \tag{6.2}$$

Hence, the average number of successful cycles during that period is

$$\bar{m} = \frac{\alpha}{1-\alpha}\ . \tag{6.3}$$

The observation period begins with the repair period of the node that had failed. During that period there are only $N-1$ operative nodes. By analogy with (Equation (6.1)), the probability, β, that they will complete one work and 2PC cycle before the failed node recovers, is

$$\beta = \int_0^\infty f_W(x)e^{-\eta x}dx = \tilde{u}(\eta)\tilde{v}(\eta)\ . \tag{6.4}$$

Consequently, the average number of consecutive cycles during the repair period (colored red in Figure 6.2) is

$$\bar{m}_1 = \frac{\beta}{1-\beta} \, . \tag{6.5}$$

The total average number of transactions departing during these cycles, J_1, is equal to

$$J_1 = \bar{m}_1 E(U)(N-1)\mu \, , \tag{6.6}$$

where $E(U) = -u'(0)$ is the average length of work intervals.

Then we have a cycle that overlaps the repair completion instant (marked in parts with red and green in Figure 6.2). In addition, if the repair instant falls during the work interval of that cycle, then for the remainder of that interval there is an extra node (i.e., the repaired one) available. The average number of transactions, J_2, departing during this cycle, given that the repair is completed within it, can be expressed as

$$J_2 = E(U)(N-1)\mu + \frac{\mu}{1-\beta} E(U-R) \, , \tag{6.7}$$

where $E(U-R)$ is the expected remaining work interval, given that the repair completes within the cycle. Averaging over the distribution of U, we can write

$$E(U-R) = \int_0^\infty f_U(x) \int_0^x (x-y)\eta e^{-\eta y} dy dx \, . \tag{6.8}$$

After carrying out the integration and substituting the result into (Equation (6.7)), the latter becomes

$$J_2 = E(U)(N-1)\mu + \frac{\mu}{1-\beta} \left[E(U) - \frac{1-\tilde{u}(\eta)}{\eta} \right] \, , \tag{6.9}$$

where $\tilde{u}(s)$ is the Laplace transform of $f_U(x)$.

Finally, the total average number of departures during fail-free cycles with N operative nodes (marked green in Figure 6.2), with their average number being $\bar{m} - \bar{m}_1 - 1$, is given by

$$J_3 = (\bar{m} - \bar{m}_1 - 1)E(U)N\mu \ . \tag{6.10}$$

The system throughput, T, defined as the average number of departures per unit time, is obtained by dividing the total average number of departures during the observation period by the average length of the observation period:

$$T = (J_1 + J_2 + J_3)N\xi \ . \tag{6.11}$$

Now consider the total average number of transactions, D, that are lost during the observation period. Losses occur either during the initial repair period, when transactions attempt to access the failed node, or when the last cycle in the observation period is interrupted by the next node failure.

As soon as a transaction is admitted into a node, it produces a list of a random number, k, of other nodes that it would need to access. If the failed node is on that list, the transaction is instantaneously dismissed and is lost. That procedure is repeated with the transaction that follows, so following a service completion there may be a series of transactions lost instantaneously.

The probability, γ, that a transaction admitted into an operative node has the failed node on its list is

$$\gamma = \frac{E(k)}{N-1} \ , \tag{6.12}$$

where $E(k)$ is the average size of the list. This is a given parameter. The average number of transactions lost instantaneously following a service completion during the repair period is therefore equal to $\gamma/(1-\gamma)$.

We conclude that the average number of transactions lost during the $\bar{m}_1$ successful work and commit cycles within the repair period is

$$D_1 = J_1 \frac{\gamma}{1-\gamma} , \tag{6.13}$$

where J_1 is given by (Equation (6.6)).

To find the average number, D_2, of transactions lost during the work and 2PC cycle overlapping the repair instant, we proceed as in the derivation of (Equation (6.7)), but count only the transactions completed before the repair instant. This leads to the following expression:

$$D_2 = \frac{(N-1)\mu}{1-\beta} \frac{\gamma}{1-\gamma} \left[\frac{1}{\eta} [1 - \tilde{u}(\eta)] - \tilde{u}'(\eta) \right] . \tag{6.14}$$

Finally, we need the average number, D_3, of transactions that are lost from the last work and commit cycle in the observation period, the one that is interrupted by the next failure instant. Since all transactions executed during that cycle are lost, we can write

$$D_3 = E(U) N \mu . \tag{6.15}$$

The overall rate of transaction losses, D, is given by the total average number of losses during the observation period, divided by the average length of the observation period:

$$D = (D_1 + D_2 + D_3) N \xi . \tag{6.16}$$

The work interval U is set by the control policy as a constant, a. On the other hand, the commit operation is affected by communication delays, so it is more natural to assume that V is random, possibly distributed exponentially with mean b. In that case, we would have

$$\tilde{u}(s) = e^{-as} \;\; ; \;\; \tilde{v}(s) = \frac{1}{1+bs} . \tag{6.17}$$

6.2.4 Average response time

We will no longer regard that nodes are always busy but assume that transactions arrive in a Poisson stream with rate λ and, if there are no available nodes, wait in an external *first in first out* (FIFO) queue. After being processed, a transaction does not depart immediately but is held in the queue until the end of the current cycle. If commit is the 2PC outcome, all transactions that were executed during the work interval depart together. Otherwise, they are aborted and continue to remain in the queue. The performance measure of interest here is the steady-state average response time, W, defined as the interval between the arrival of a transaction into the system and its departure.

This type of system has been referred to in the literature as a *queue with bulk services*. At certain *service instants*, batches of transactions are removed from the queue. More precisely, if the size of the current batch is m and the number of transactions present just before the service instant was n, then just after the service instant there are $n - \min(n,m)$ transactions present. A model where the intervals between service instants have a general distribution and all batches have the same fixed size was analyzed by Bailey [7] more than half a century ago.

Bailey's result cannot be used in our case because the number of transactions departing at a service instant, i.e., when 2PC execution completes, depends on whether a breakdown occurred during the cycle or not, and also on whether there were N or $N-1$ operative servers. We propose two estimates for W: the first is pessimistic and can be treated as an upper bound on the response time; the second is clearly optimistic and will provide a lower bound.

6.2.5 Upper bound, W_u

The first estimate is obtained by assuming that the consecutive intervals between service instants are i.i.d. random variables distributed exponentially with mean $a+b$, where a is the average work interval and b is the average 2PC interval. The parameter of that distribution will be denoted by $\nu = 1/(a+b)$. The reason why this is a pessimistic assumption is that

the coefficient of variation of the exponential distribution is 1, while in practice the work interval is likely to be constant, or nearly constant. The 2PC interval tends to be much smaller than the work interval, so even if it is random, the coefficient of variation of a full cycle (comprising both work and 2PC intervals) would tend to be closer to 0 than to 1.

Under the exponential assumption, the probability that a full cycle is not interrupted by a node failure is now approximated by

$$\alpha = \frac{\nu}{N\xi + \nu} \,. \tag{6.18}$$

When $N\xi$ is small, this value is very close to the one produced by (Equation (6.1)).

Hence, a service batch is of size 0 with probability $q_0 = 1 - \alpha$.

Since the average period during which there are $N-1$ operative servers is $1/\eta$ and the average period during which there are N operative servers is $1/N\xi$, we can say that a 2PC interval has $N-1$ operative servers with probability $q_1 = \alpha N\xi/(N\xi + \eta)$, and has N servers with probability $q_2 = \alpha\eta/(N\xi + \eta)$. In the former case, an average of $m_1 = a(N-1)\mu$ transactions are served during the cycle, and in the latter case $m_2 = aN\mu$ transactions are served.

The above arguments support an assumption that the service batch size, m, is equal to

$$m = \begin{cases} 0 & \text{with probability} \quad q_0 \\ m_1 & \text{with probability} \quad q_1 \\ m_2 & \text{with probability} \quad q_2 \end{cases} . \tag{6.19}$$

If m_1 and m_2 are not integers, their integer parts are taken.

The average batch size, B, is

$$B = m_1 q_1 + m_2 q_2 = aN\mu\alpha \frac{(N-1)\xi + \eta}{N\xi + \eta} \,. \tag{6.20}$$

The necessary and sufficient condition for the stability of the bulk service queue is that the transaction arrival rate should be strictly less than the average service capacity:

$$\lambda < \nu B \,. \tag{6.21}$$

When the failure rate is small and the repair rate is significantly higher, the right-hand side of this inequality is very close to the maximum throughput, T, obtained in the previous section. Thus, requirement (Equation (6.21)) is almost identical to the more accurate stability condition $\lambda < T$.

Let π_n be the steady-state probability that there are n transactions present in the queue. Because of the bulk service assumption, any transactions that are in fact being served, are considered to be in the queue until the next service instant. The number n increases by 1 at arrival instants, and it decreases by 0, m_1 or m_2 at service instants. Equating the up and down transition rates across the boundary between states n and $n+1$, we obtain the following set of balance equations.

$$\lambda \pi_n = \nu \left[\sum_{j=1}^{m_1} (q_1 + q_2) \pi_{n+j} + \sum_{j=m_1+1}^{m_2} q_2 \pi_{n+j} \right]; n = 0, 1, \ldots \tag{6.22}$$

We shall obtain the general solution to this set of equations in geometric form:

$$\pi_n = C z_1^n \,, \tag{6.23}$$

where C and z_1 are some positive constants. Substituting (Equation (6.23)) into (Equation (6.22)), we find that the equations are satisfied as long as z is a zero of the following polynomial of degree m_2.

$$P(z) = \lambda - \nu \left[\sum_{j=1}^{m_1} (q_1 + q_2) z^j + \sum_{j=m_1+1}^{m_2} q_2 z^j \right] \,. \tag{6.24}$$

In addition, in order that we may obtain a probability distribution, z_1 must be a positive real number in the interval $0 < z_1 < 1$.

We have $P(0) = \lambda > 0$ and $P(1) = \lambda - \nu(m_1 q_1 + m_2 q_2) < 0$, according to (Equation (6.21)). Therefore, $P(z)$ has a real zero, z_1, in the interval $(0,1)$. This provides a normalizable solution to the set of balance equations and allows us to write

$$\pi_n = (1 - z_1) z_1^n \ ; \ n = 0, 1, \ldots \ . \tag{6.25}$$

It is possible to prove formally that $P(z)$ has no other zeros in the interior of the unit disk, but this also follows from the fact that an ergodic Markov process cannot have more than one normalizable distribution.

The steady-state average number of transactions in the system, L, is obtained from (Equation (6.25)):

$$L = \sum_{n=1}^{\infty} n\pi_n = \frac{z_1}{1 - z_1} \ . \tag{6.26}$$

The upper bound of the average response time, W_u, is then provided by Little's theorem:

$$W_u = \frac{L}{\lambda} \ . \tag{6.27}$$

6.2.6 Lower bound, W_d

A very simple lower bound is derived by making two optimistic assumptions. The first is that the work interval and 2PC interval are constant, of lengths a and b, respectively. The second is that the transactions arriving during the work interval are cleared at the end of that cycle, while those arriving during the commit operation are cleared during the next cycle, provided that no breakdown occurs in the meantime. That would be a reasonable assumption if the total average number of arrivals during work and 2PC intervals of a cycle is

smaller than the average number that can be served by $N-1$ servers during a work interval: $\lambda(a+b) < a(N-1)\mu$.

When the cycle duration is constant at $(a+b)$, the probability that a cycle does not involve a node failure is

$$\alpha = e^{-N\xi(a+b)} . \tag{6.28}$$

In that case a transaction arriving during a work interval remains in the system for an average of half a work interval plus 2PC interval, while an arrival during a 2PC interval remains in the system for an average of half 2PC interval plus a full cycle. The probabilities that an incoming transaction arrives during a work interval or a 2PC interval are $a/(a+b)$ and $b/(a+b)$, respectively. Hence, the average sojourn time given that the cycle is failure-free can be written as

$$\left[(\frac{a}{2}+b)\frac{a}{a+b} + (\frac{b}{2}+a+b)\frac{b}{a+b}\right] = \frac{a+3b}{2} .$$

In the event of a node failure, the average sojourn time is half a cycle plus a full cycle. Thus, the lower bound on the response time becomes

$$W_d = \alpha\frac{a+3b}{2} + (1-\alpha)\frac{3(a+b)}{2} . \tag{6.29}$$

Remember that the average work interval, a, is not a system characteristic but is set by the operating policy. It is natural to ask therefore, how should that interval be chosen in order to minimize W? On the one hand, increasing a may improve the throughput (although that effect is mitigated by an increase in the probability of a node failure during a full cycle). On the other hand, transactions are kept in the system for longer. Intuitively, there should be an optimal value for a.

Note that the lower bound (Equation (6.29)) tends to be an increasing function of a. Therefore, if W_d is taken as a criterion, the optimal a is the smallest value that justifies the assumption made above, i.e. that the average number of transactions arriving during a cycle

should be significantly smaller than the number that $N-1$ servers can serve during a work interval. We suggest as an empirical rule of thumb that one should chose the smallest a that satisfies the inequality $\lambda(a+b) < 0.8a(N-1)\mu$. This yields the value, a^*, that minimizes W_d as

$$a^* = \frac{\lambda b}{0.8(N-1)\mu - \lambda} \,. \tag{6.30}$$

If the upper bound is minimized, the optimal work interval will, in general, be different. These differences will be investigated experimentally.

6.3 Epoch-based Multi-commit

6.3.1 Rationale and Approach

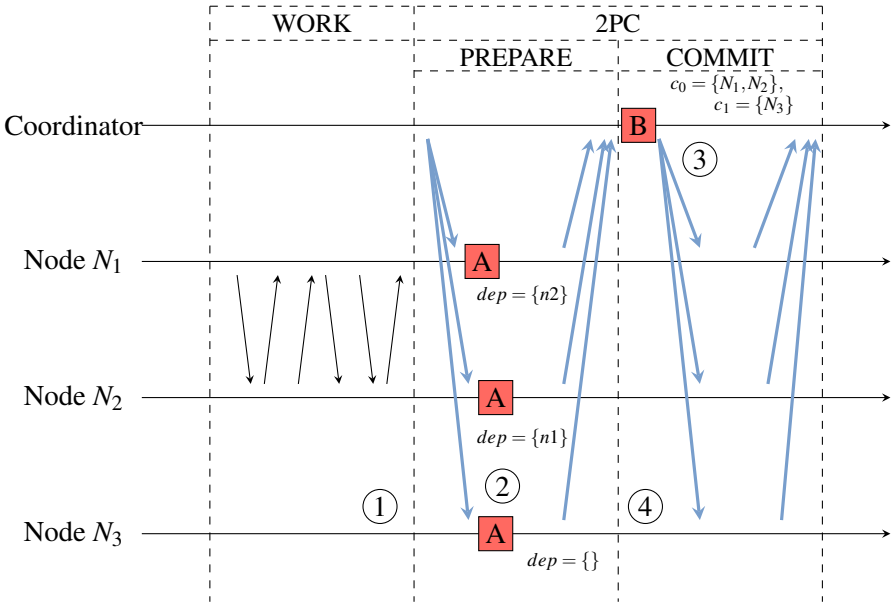


Fig. 6.3 A cycle in epoch-based multi-commit. A node's epoch-dependency list is indicated as *dep*.

Epoch-based commit assumes that uncommitted data within any node is used, directly or indirectly, by every other node during a work interval. Hence, all transactions executed are aborted in case of a node failure. In reality, this assumption is pessimistic and does not

always hold. For example, if a node is to fail shortly after it starts its work interval, it would be very unlikely that each operative node executes a distributed transaction in that short duration preceding the failure *and* uses uncommitted data held by the node that goes on to fail later.

Epoch-based multi-commit avoids, where possible, aborting all transactions and thereby seeks to improve throughput and reduce average latency. Interactions between nodes are monitored in a lightweight, low-overhead manner to determine whether a node needs to abort its transactions, when another node is found to have failed. Interactions that call for aborting of transactions need not just be directly with the failed node but can also be transitive in nature, as explained in the scenario below.

Suppose that node N_k updates an object; these updates do not become durable until N_k commits and will be lost if N_k fails before that. Let N_j process a (distributed) transaction by reading an uncommitted update at N_K and updating some local objects which are in turn read by a third N_i while processing another transaction. If N_k crashes, N_j must abort its transactions as some of them involved reading *dirty* data from N_k; this in turn leads to *cascading aborts* at N_i even though there is no direct interaction between N_i and N_k.

We will express the pattern of node interactions during each work interval as a symmetric and transitive binary relation between nodes. This relation is called *epoch dependency* and is defined as follows: Node N_i has epoch dependency with N_j during a given work interval iff: (i) N_i accesses data from N_j during that work interval or vice versa, or (ii) there is another N_k for that work interval such that N_i has epoch dependency with N_k and N_k with N_j.

Two remarks are in order. First, the above definition does not distinguish whether a node interaction involves accessing uncommitted or committed data, even though the latter does not call for cascading aborts. Despite some advantages in making this distinction, we avoid it in order to keep the overhead of monitoring node interactions as small as possible.

Secondly, epoch dependency is also reflexive by definition, as each node accesses data from itself. So, it is an *equivalence* relation defined on participant nodes. It therefore partitions the nodes into disjoint subsets which we call *commit groups*: each node is in exactly one group, any two nodes of a group are related by epoch dependency, and no node has epoch dependency with any node not in its own group. Therefore, if a commit group has no failed node, then its member nodes can commit during 2PC; otherwise, they must abort.

Referring to Figure 6.3, we see nodes N_1 and N_2 interacting with each other during the work interval and N_3 executing no distributed transactions. So, N_1 and N_2 have epoch dependency with each other and N_3 only with itself; thus, two commit groups, $c_0 = \{N_1, N_2\}$ and $c_1 = \{N_3\}$, emerge. If N_3 fails, N_1 and N_2 can commit their transactions because they did not access any data from N_3. If N_1 fails, $N_2 \in c_0$ must abort its transactions, while N_3 is unaffected.

A failed node can thus prevent nodes of only one group from committing. That is, if node interactions during a work interval lead to multiple commit groups emerging at the end, then all nodes do not have to abort their transactions in the event of a failure. On the other hand, if only one commit group exists at the end of a work interval, then a node failure will cause all operative nodes to abort, i.e., the epoch-based multi-commit defaults to the original, epoch-based commit. For example, had N_1 and/or N_2 in Figure 6.3 interacted with N_3 during the work interval, then $c_0 = \{N_1, N_2, N_3\}$ would be the only commit group and any node failure would mean all other nodes aborting their transactions.

6.3.2 Motivation: TPC-C Case Study

For multi-commit protocol to minimize aborts, multiple commit groups should emerge at the end of a work interval. Such an outcome depends primarily on three workload characteristics: (i) proportion of distributed transactions, (ii) average number of remote nodes accessed by distributed transactions, and (iii) *node affinity* or the likelihood of a transaction in a given

node accessing data in another given node due to correlations between data partitions hosted by the two nodes.

The smaller the proportion in (i), the larger is the likelihood of multiple groups. In the limit, if there are no distributed transactions at all, then each commit group is a singleton with a distinct node - as c_1 in Figure 6.3. The smaller the average in (ii) and the stronger the affinity, the more likely it is that multiple groups emerge even if the workload has a higher proportion of distributed transactions. In what follows, we consider a canonical benchmark system to motivate the multi-commit scheme can be very useful in practical settings and its performance is worthy of a detailed evaluation.

TPC-C is the canonical benchmark for evaluating performance of OLTP databases [109]. It models a warehouse order-processing application and consists of five transaction types. Only two types, `Payment` and `NewOrder`, involve accessing remote nodes. A `Payment` transaction involves updating the payment amounts for a given warehouse and then updating customer information. The customer belongs to remote warehouse on another server with a 15% probability. In short, the `Payment` transaction accesses at most two partitions. A `NewOrder` transaction updates 5-15 items in the stock table. Of these items, 99% are local to its home partition, while 1% are at a remote partition.

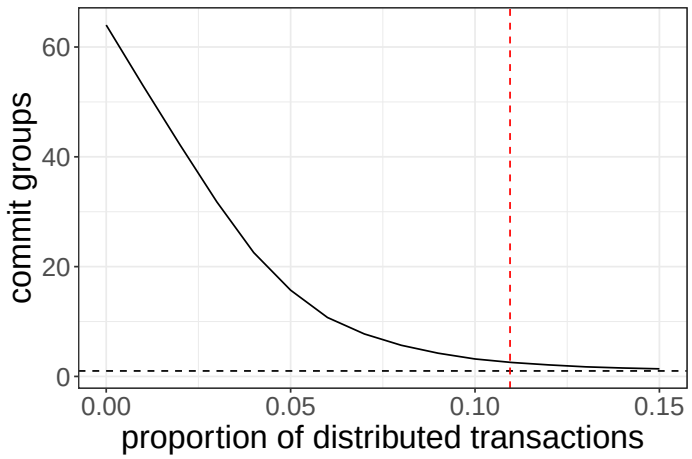


Fig. 6.4 Number of commit groups *vs* proportion of distributed transactions. The red line indicates the threshold after which single commit group is the only outcome.

In our experiment, the epoch size was fixed to 10*ms* and a throughput of 300K transactions per second in a cluster with 64 servers was assumed. Figure 6.4 depicts the number of commit groups formed when the the proportion of distributed transactions is varied from 1% to 15%. Commit groups are numerous when the proportion of distributed transactions does not exceed 8% which is the typical proportion of distributed transactions encountered in practical settings. Thus, multi-commit is indeed a practical alternative to the original scheme.

6.3.3 Multi-Commit Protocol Description

This protocol is identical to epoch-based commit described in Section 6.2, except for the following two additions.

Monitoring node interactions. When node N_i executes a distributed transaction and sends a `Remote-Op` message to another node N_j, it enters the remote N_j in its local *epoch-dependency* list. Similarly, when N_j receives `Remote-Op` message from N_i, it enters the latter in its list. Thus, interacting nodes have each other in their *epoch-dependency* lists.

Computing commit groups. This is done by the coordinator node if some participant node has not responded to it with `Prepare-Ack` in the prepare phase of 2PC execution. Recall that the coordinator executes 2PC at the end of work interval by sending a `Prepare` message to all participant nodes. (2PC messages are shown in blue in Figure 6.3.) Each operative node will, in turn, respond to the coordinator by sending a `Prepare-Ack` message which will now include its epoch-dependency list as indicated in Figure 6.3.

If the coordinator receives `Prepare-Ack` from all nodes, then a `Commit` message is sent to all nodes. Otherwise, it constructs a graph where a vertex represents a participant node and an edge represents an epoch-dependency as reported within the epoch-dependency lists it received. Commit groups are then found by using Tarjan's algorithm to identify strongly connected components [101]. Participant nodes of commit groups that contain no failed node are sent a `Commit` message and the rest an `Abort` message. Based on the descriptions

in Subsection 6.3.1, it is easy to see that a node is sent `Commit`, if and only if it has not interacted with a failed node directly or transitively, during the work interval.

6.4 Performance Evaluation Strategies

The principal aim of simulations is two-fold: to assess the accuracy of the analytical models and explore circumstances in which the epoch-based multi-commit can perform better than the original epoch-based commit. The former will assist database administrators in choosing a suitable work interval, a, to accomplish their performance targets and the latter will help to demonstrate that the multi-commit protocol is a practical alternative to the original. Recall that multi-commit cannot perform worse than the original in any circumstance; this is because the former differs from the latter only when a node fails at which time the coordinator expends a small computational cost on trying to minimize aborts.

In assessing the efficacy of the expression for maximum throughput, our discrete event-based simulations [78] will follow the experimental setup in [72]: nodes will spawn a new transaction as soon as they finish executing the current one. Thus, the nodes are never idle and the resulting throughput will be the maximum attainable. In measuring the average latency, simulations will consider such values for transaction arrival rates that the system is kept in a steady state; i.e., the number of transactions waiting to be processed will not grow monotonically with time. Under this steady state condition, throughput is the same as the arrival rate and hence not measured.

In our simulations, incoming transactions are distributed ones with 10% probability which is larger than the threshold 8% observed in Figure 6.4 for having multiple commit groups at the end of a work interval. Thus, we seek to explore the effects of node-affinity when the proportion of distributed transactions does not favor the emergence of multiple groups.

A distributed transaction interacts with one remote node and we consider two policies for choosing that remote node: *random* and *paired affinity*. In the former, the remote node is

randomly chosen; in the latter, nodes are paired and a distributed transaction originating in a given node accesses the paired node with 90% probability and a randomly chosen one with 10% probability. Node-pairing captures data correlations between the partitions hosted by the paired nodes.

Note that if a remote node chosen for data access is crashed, then processing of that transaction ceases and all effects of having processed it are undone. Such a transaction is called *'dropped'* in Subsection 6.2.3 and not counted in throughput. Also, 10% of transactions accessing one remote node sets $E(\kappa) = 0.1$ in (Equation (6.12)).

Table 6.1 Parameters of the analytical models and simulation.

Symbol	Meaning	Values
N	Number of participant nodes	64
a	Work interval (ms)	4-1800
b	Mean time to commit (ms)	1.7
μ	Node's transaction service rate (txn/ms)	1
$1/\xi$	Mean time between failure (hr)	12
$1/\eta$	Mean time to repair (min)	30
$E(\kappa)$	Remote servers accessed by transactions	0.1
$\lambda^{\dagger}$	Transaction arrival rate (txn/s)	30000,40000

† Average response time model only.

The parameters of the analytical models and their values used in simulations are summarized in Table 6.1. The selected values were chosen in a way to be representative of a practical OLTP database configuration and workload. A cluster size of 64 nodes and one coordinator is similar to that used in the experimental analysis of concurrency control protocols in [54]. The choice of $\mu = 1$ is guided by the fact that OLTP transactions' useful work consumes about one millisecond and they seldom have user stalls, rather they are executed as stored procedures [97]. The mean time to execute 2PC is represented by b. To estimate b, it was assumed a disk flush takes $10\mu s$ and database nodes are co-located within the same datacenter with a round trip time of $1ms$. Thus, as 2PC operations involve 2 sequential disk writes and 1.5 network calls before results are released back to clients, b is set to $1.7ms$. Following [48],

the mean time between failure $1/\xi$ and the mean time to repair $1/\eta$ are taken to be 12 hours and 30 minutes respectively.

Given that $N = 64$, it is possible to encounter more than one failed node in simulations even though $1/\xi \gg 1/\eta$. (Note: the larger the value of N, the less likely it is for Assumption 1 to hold.) Thus, any loss of accuracy due to Assumption 1 in Subsection 6.2.2 is assessed. Simulations measure the following metrics:

System throughput (T): Number of transactions committed per second.

Lost transaction rate (D): Rate at which transactions are being aborted or dropped due to failures.

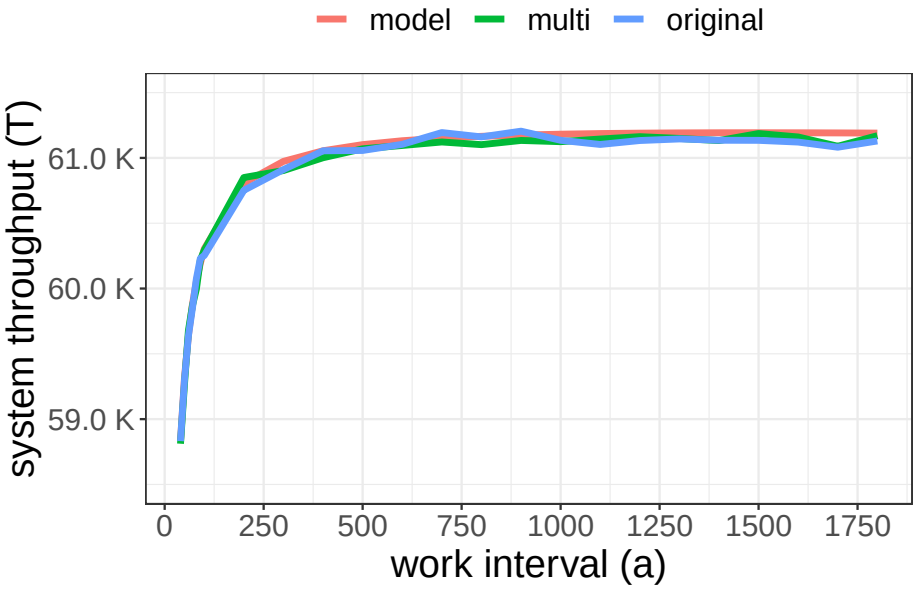
Committed transactions during failures (CT_f): Average number of transactions committed in cycles with failures.

Average Response time (W): The response time is measured from the point when a transaction enters the system, to the point when it departs, after potentially several retries. Since transactions are processed in batches and some may not be committed in their first execution, the average value (in *ms*) is computed over the simulation period.

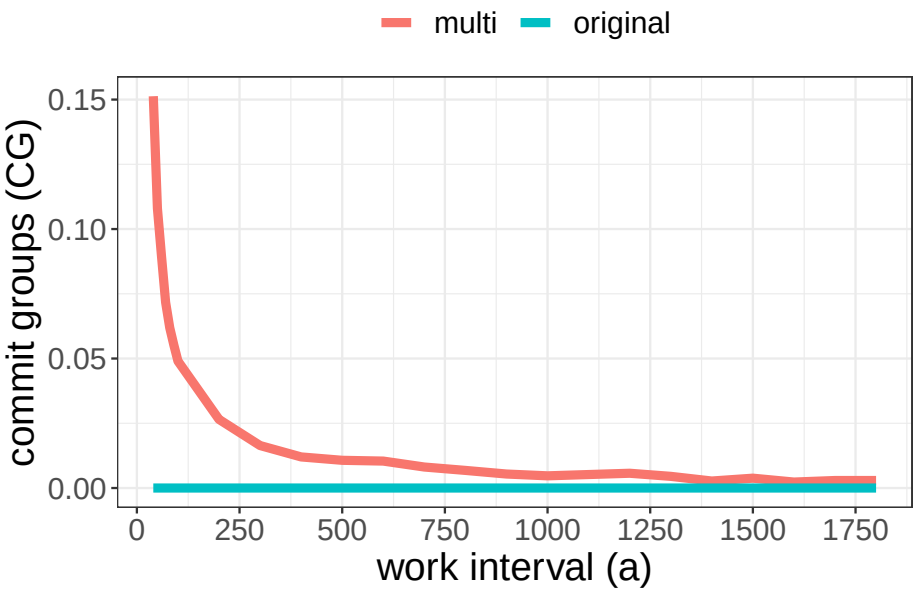
Operational commit groups (CG): the number of commit groups that do not contain a failed node, given that node failure has occurred in a given cycle. It is zero for the epoch-based commit where nodes always form one single commit group.

6.5 Evaluation

Each simulation run took approximately 12 hours to complete and simulated a cluster operational period of 100 days in order to have thousands of cycles with node failures. (Note: Experiments in [72] did not study the impact of node failures.) For example, when $a = 40$ *ms*, 10,972 cycles had failures out of a total of 207 million cycles simulated. So, the number of operational commit groups reported here would be an average of at least 10,000 values obtained. We observed up to 10 cycles having multiple node failures when $a = 1800$ *ms*.



(a) System throughput *vs*. *a* in *ms*.



(b) Operational commit groups *vs*. *a* in *ms*.

Fig. 6.5 Maximum throughput as work interval *a* varied from 40 to 1800 *ms*.

6.5.1 Maximum Throughput

Figure 6.5(a) plots the maximum attainable throughput values against the work interval a which is varied from 40 to 1800 *ms*. Throughput estimated using the expression in Subsection 6.2.3 is referred to as *'model'* and those measured in simulations for epoch-based commit and epoch-based multi-commit are labeled as *'original'* and *'multi'* respectively. Simulations used the random assignment policy when a distributed transaction sought to interact with a remote node.

We can make three observations from Figure 6.5(a). First, the estimated throughput very closely tracks the simulated values at all a. In fact, the maximum difference ever observed was around 3.8%. This suggests that Assumptions 1 and 2 of Subsection 6.2.2 have a negligible impact on the accuracy and the analytical model is nearly exact.

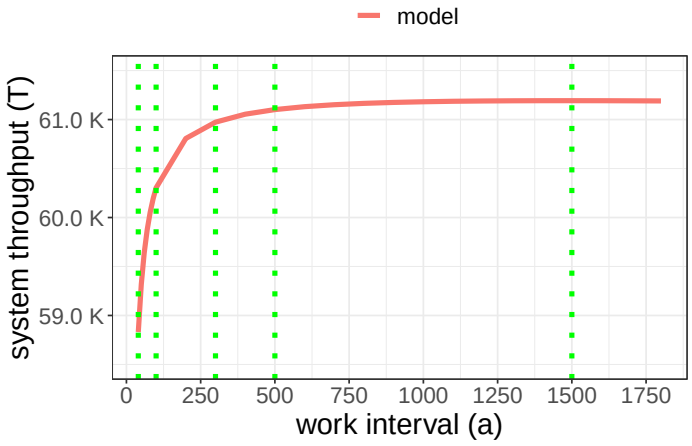


Fig. 6.6 System throughput estimates *vs*. a. Green dotted lines indicate regions with different gradients.

Secondly, throughput values of both protocols are nearly identical. This is explained by Figure 6.5(b) that presents the number of operative commit groups (CG) formed in cycles with node failures. CG takes the maximum value of just 0.15 and rapidly falls as a increases. Such insignificantly small values of CG in multi-commit are due to the proportion of distributed transactions (10%) in the workload and the random policy employed for choosing the node

to interact with. These two factors lead to almost all operative nodes interacting, directly or indirectly, with the failed node by the time the work interval a completes. This effect is more pronounced for larger a values. Consequently, multi-commit cannot perform significantly better than the original when nodes fail. Moreover, even this minute performance advantage of multi-commit during cycles with failures nearly vanishes when average throughput is taken over all cycles, because failure-free cycles far out-number those with failures. (Recall, when there are no failures, multi-commit performance is identical to the original.)

Finally, the analytical expression of Subsection 6.2.3 can be reliably used in choosing appropriate a_T when maximum throughput is the primary concern. For convenience, Figure 6.5(a) is reproduced in Figure 6.6 without simulated throughput values so that throughput estimates for various a are clear. Referring to Figure 6.6, we observe that throughput does not decrease until $a = 1500\ ms$; thus, optimal a_T^* is 1500 *ms*. For some workloads, a work interval around 1500 *ms* will offer unsatisfactory latency and be unacceptable. Thus, finding a smaller a_T that still offers an acceptable maximum throughput may be desirable and can be guided by the observation that increasing a need not fetch a proportional increase in throughput. Figure 6.6 shows four distinct regions where the rate of throughput increase is markedly different: the gradient is very large, fairly large, small and very small when $a \in [40, 100)$, $a \in [100, 300)$, $a \in [300, 500)$ and $a \in [500, 1500]$ *ms* respectively.

6.5.2 Average Response Time

Our second set of experiments focuses on assessing the effectiveness of the average response time analytical models in Subsections 6.2.5 and 6.2.6. Simulations retain the random assignment policy for distributed transactions; transaction arrival rate per second is taken to be $\lambda = 30,000$ which is approximately 90% of the maximum throughput when $a = 5ms$ to ensure the system is in a steady state. Figure 6.7 plots the model estimates of the lower (W_d) and upper(W_u) bounds and the simulation values measured for the epoch-based commit

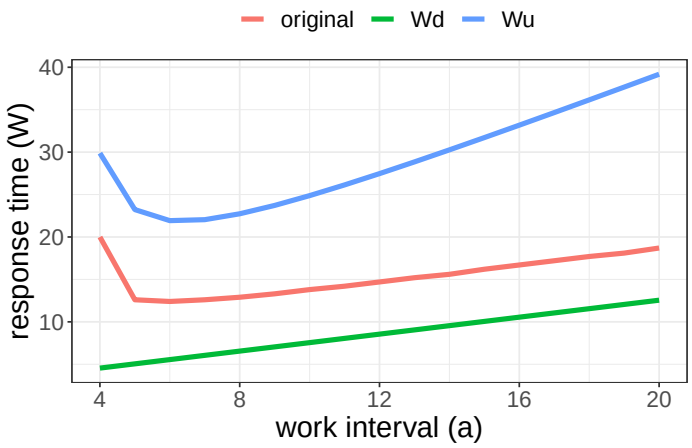


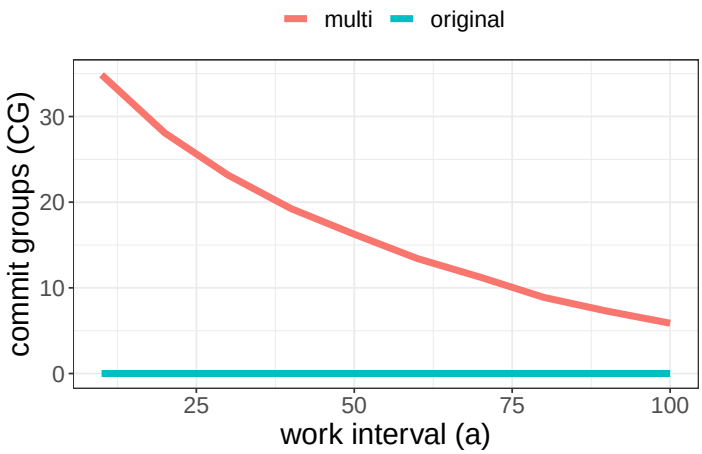
Fig. 6.7 Average response time (*ms*) in epoch-based commit *vs*. a in *ms*.

protocol as a is varied from 4 to 20 *ms*. (The range choice for a is guided by [72] where experiments used $a = 10$ *ms*.)

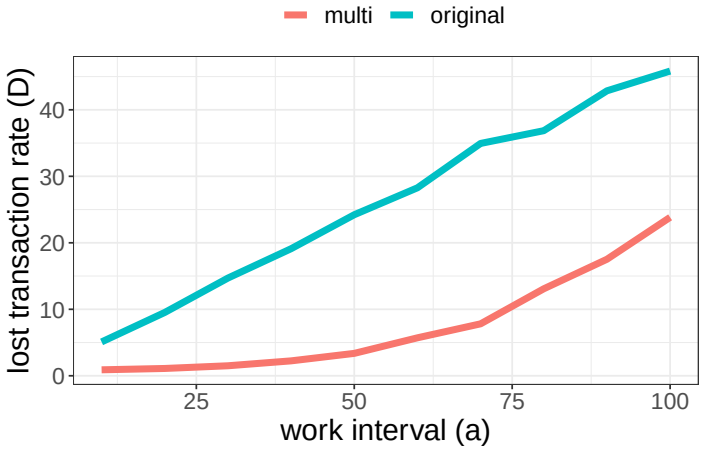
The response times measured in simulations are well within the upper and lower bound estimates. The latter increase linearly with a as expected. The W_u plot predicts the optimum a for minimizing the average response time as: $a^* = 6$ *ms* which is consistent with simulations. Though the analytical expression for W_u (Subsection 6.2.5, Equation (6.27)) identifies a^* reasonably accurately, the actual response times are much closer to W_d as a increases and the maximum difference we observed was 6 *ms*. So, to summarize, analytical expressions for W_u and W_d are reasonably accurate in predicting a^* and the actual response times for $a \geq a^*$ respectively. The simulation response times obtained for multi-commit were very close to those presented for the original for reasons explained in Subsection 6.5.1 and hence they are not shown.

6.5.3 Paired Affinity

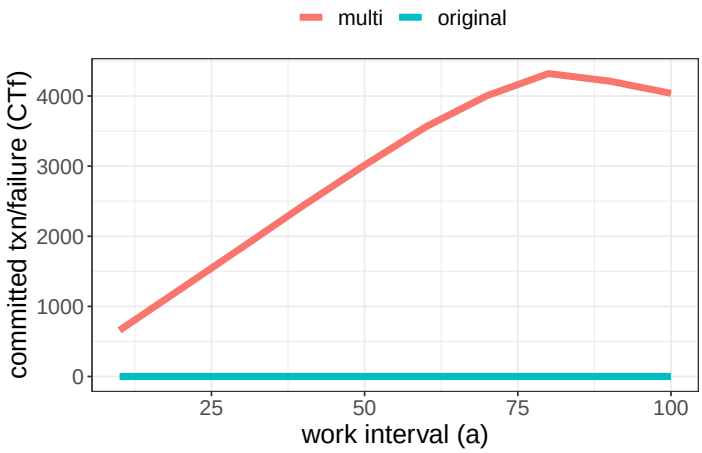
We ran the maximum throughput simulation experiment using the paired affinity selection policy for distributed transactions. Figure 6.8(a) displays the number of operational commit groups (*CG*) formed whenever failures occurred in a cycle. In sharp contrast to Figure 6.5(b),



(a) Commit groups *vs*. *a*.



(b) Lost transaction rate *vs*. *a* in *ms*.



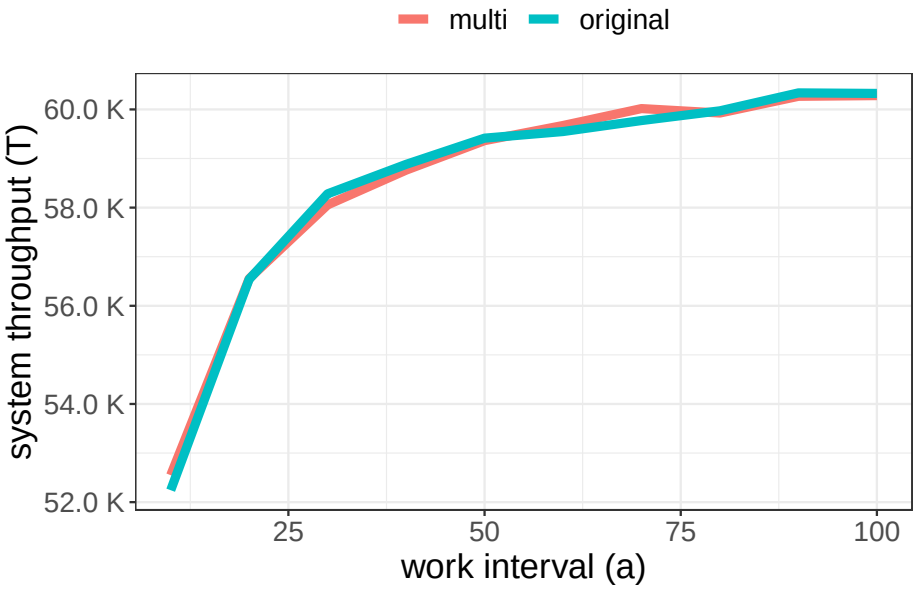
(c) Transactions committed in cycles with node failure CT_f *vs*. *a*.

Fig. 6.8 Simulations under paired-affinity as work interval a varied from 10 to 100 *ms*.

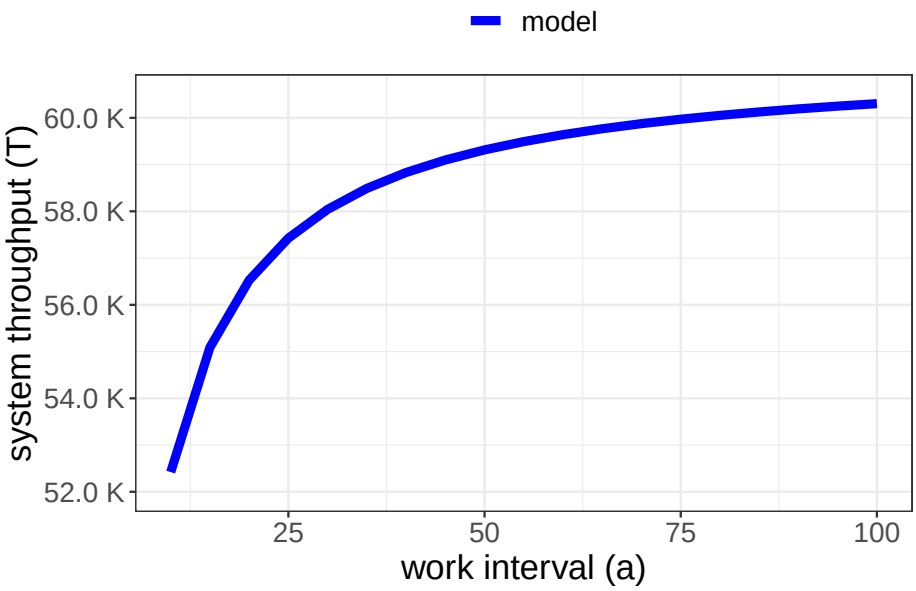
CG for multi-commit starts at a much larger value of 35 when $a = 10$ *ms* and falls steadily to 5 as a increases to 100 *ms*. This suggests a strong potential for reducing the number of aborts when failures occur.

Figure 6.8(b) plots the lost transaction rates (D) for both protocols and shows that D for multi-commit is consistently smaller than the original. For small a, the difference is small, because the number of transactions lost due to failures is small for both protocols. But as a increases, D for the original increases almost linearly while that for multi-commit does not start increasing significantly until $a = 50$ *ms*; at that point, multi-commit shows 83% reduction in D with the corresponding *CG* being around 17 in Figure 6.8(a). As a increases further, *CG* for multi-commit starts dropping significantly in Figure 6.8(a) and consequently D for multi-commit starts increasing rapidly.

Finally, Figure 6.8(c) shows the average number of transactions committed by the protocols in those cycles where node failures occur (CT_f). From Figure 6.8(c) it is clear that multi-commit avoids a significant number of aborts; note that $CT_f = 0$ in the original. However, these differences in CT_f make insignificant difference when throughput and latency are averaged over the simulation period, because cycles without failures far outnumber (by four orders of magnitude) those with failures and both protocols perform identically in fail-free cycles. Thus, the average maximum throughput and average latency for both protocols, plotted in Figures 6.9(a) and 6.10(a), are almost identical. This implies that the analytical expressions obtained for epoch-based commit under the random policy have a wide applicability: comparing Figures 6.9(a) and 6.10(b) with Figures 6.9(b) and 6.7 respectively suggests that those expressions are equally applicable for (i) obtaining appropriate a for epoch-based multi-commit, and (ii) epoch-based protocols under paired affinity policy.

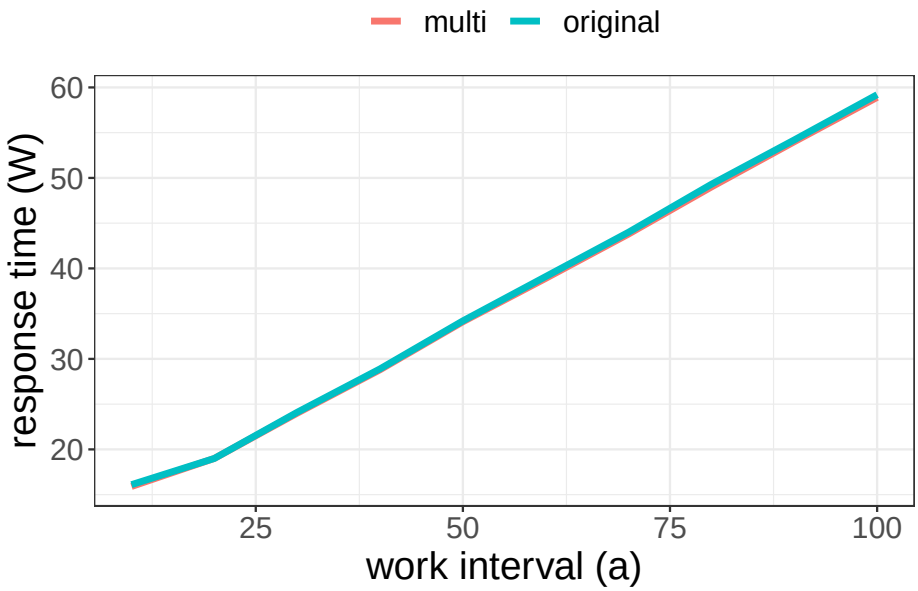


(a) Simulations using paired affinity.

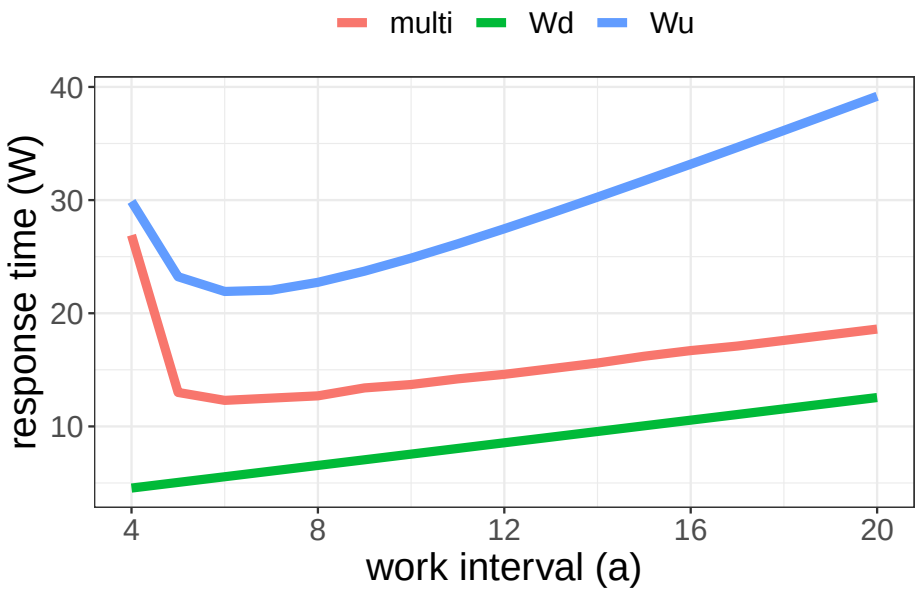


(b) Model using random affinity.

Fig. 6.9 Maximum throughput as work interval a varied from 10 to 100 ms.



(a) Simulations with paired affinity as a is varied from 10 to 100 ms ($\lambda = 40,000$).



(b) Multi-commit with paired affinity and models with random affinity as a is varied from 4 to 20 ms ($\lambda = 30,000$).

Fig. 6.10 Average response time (ms).

6.6 Conclusion

In this chapter two analytical models for the epoch-based commit protocol have been developed which allow database operators to maximize throughput, minimize average response time, or seek a trade-off between them. The accuracy of these models has been validated through a simulation study that considered a cluster of 64 nodes operating for 100 days. We also developed epoch-based multi-commit, which aims to minimize transaction aborts in the event of node failures, but performs identically to the original version under other circumstances. Our simulation study affirms that multi-commit performs better when distributed transactions originating at a given node tend to access specific other nodes in their remote interactions. When failures are rare, the analytical expressions derived for the original protocol can also be used in determining the right epoch intervals for the multi-commit version as well. Thus, we offer a practical alternative to epoch-based commit and analytical solutions to efficiently tune the parameter of epoch-based commit protocols in practical settings.

6.6.1 Further Work

Dynamic work intervals. For a variety of reasons, users often operate their databases at a near full load. So when a failure occurs they can experience a significant spike in transactions' average latency whilst the problem is resolved. However, these systems seldom implement an appropriate back-pressure mechanism, i.e., the rate of new transactions into the work queue remains unchanged. As a consequence, the system takes a long time to return to the average latency prior to the failure event (if plotted, average latency would follow a short spike followed by a slow decline to normal behaviour). This is unaligned with user expectations: they desire a return to "normal" behaviour swiftly. A potential avenue for future work is to explore the efficacy of dynamically adjusting the size of the work interval in response to the occupancy levels of the external work queue. Such a mechanism could

prove useful in smoothing average latency in response to failures. Additionally, workloads vary over time due to a number of factors. An example of this is demonstrated in the LDBC Interactive workload [37], which features *spiking trends*, increased activity in the social network in response to real world events, e.g., elections and natural disasters. Thus, *dynamic epoch-based commit* may be also of practical use during normal operation.

Decentralized epoch-based multi-commit. A drawback of epoch-based commit is its sensitivity to imbalanced workloads, a straggler node processing a long-running transaction can prevent the whole epoch from committing, as a result increasing the average latency of all transactions. Future work could explore how to decentralize the coordinator node across the cluster. In a decentralized iteration of the protocol, each participant node would manage its own local epoch that it periodically attempts to increment. In effect, different nodes would move through epochs at different rates. Multi-commit's epoch dependency tracking mechanism would remain the same, with the difference that now when a node tries to increment its local epoch counter it must determine its epoch commit group and elect a leader to manage the commitment of said epoch. This could be achieved by a simple approach such as the node with the highest process id being elected leader. Note, due to the transitive nature of epoch dependencies, leader election could take a long time as nodes discover the members of the commit group. This approach would allow nodes that have not interacted with any straggler nodes processing long-running transactions to continue to make forward progress, which should lower the average latency of transactions.

Geo-distributed epoch-based multi-commit. Geo-distributed databases replicate data across the world to keep data close to users, for fault-tolerance, and for regulatory requirements, e.g., GDPR. Discussions in this chapter thus far have been restricted to deployment in a single datacenter. Another area for further work is exploring how multi-commit could be used in a geo-distributed database. Consider a company that operates in three jurisdictions,

US, Germany, and Japan. The company's customer data is partitioned to reflect users in each area, i.e., partitions contain data solely from one location. These partitions are then replicated across three data-centers, one in each location. One anticipates that the majority of transactions would access data for a single location. Therefore, an epoch coordinator could be deployed in each datacenter, primarily responsible for coordinating commit over the partitions holding data for that location. If transactions access data from multiple locations then some degree of coordination would be needed between each location's coordinators in a similar vein to decentralized multi-commit.

Chapter 7

Conclusions

Summary

This chapter summarizes the work presented in this thesis, addressing the inefficiencies and limitations of existing research in database concurrency control and distributed atomic commitment. The contributions and limitations of these works are outlined, before we discuss open challenges in the area.

7.1 Thesis Summary

In this thesis we have explored concurrency control in the context of many-core and distributed OLTP databases. Additionally, we have investigated how the limitations of existing work on distributed atomic commitment can be addressed.

In Chapter 3, we developed the wait-hit protocol, an optimistic Serializable concurrency control protocol that scales vertically, with the core count, and horizontally, with the cluster size. The motivation behind this work was an attempt to create a concurrency control protocol catered for modern cloud environments that does not need to be re-adapted for different scale points, thus is better suitable to an environment in which users can easily scale from a many-core machine, to a deployment spread across data centers. We demonstrated that the Wait-Hit Protocol has comparable performance with the state-of-the-art many-core concurrency control protocols.

Next in Chapter 4 we focused on efficient implementation of weak isolation on servers with many-cores. Mixed Serialization Graph Testing minimizes the number of unnecessary aborts, and crucially permits concurrent transactions to be executed at a range of isolation levels. The performance of MSGT is demonstrated using several popular transaction processing benchmarks.

Chapter 5 focuses on concurrency control in specialized distributed graph databases, specifically the design of two protocols that preserve Reciprocal Consistency and one which

preserves Edge-Order Consistency are presented. The Delta protocol provides probabilistic guarantees of data integrity, whereas Deterministic Reciprocal Consistency Protocol and Deterministic Edge Consistency Protocol ensure there is no chance of inconsistencies. Deterministic Edge Consistency Protocol builds on the same principles as those presented in Chapter 3. Approximate models were developed for each protocol to allow for a comprehensive performance evaluation.

Lastly, in Chapter 6 we investigated epoch-based distributed databases and developed two analytical models of epoch-based commit. These models facilitate the choosing of an epoch size that maximizes throughput, minimizes average response time, or seeks a trade-off between them. We also developed epoch-based multi-commit which can minimize transaction aborts in the event of node failures depending on various workload characteristics.

7.2 Limitations

The lack of performance evaluation of the Distributed Wait-Hit Protocol, either through modeling, simulations, or implementation, is an important limitation of this thesis as it prevents us from truly ascertaining the performance benefits compared to Distributed Serialization Graph Testing. However, in the future a full implementation of Distributed Wait-Hit Protocol is planned and will allow for comparison with a range of protocols, similar to the study executed in [54].

A key limitation of the work performed in Chapter 4 is the lack of comparison with other mixed concurrency protocols such as Mixed-2PL. Additionally, there still remains a question mark over the practical utility of such a protocol and of weak isolation in general [99].

Another limitation of the work conducted in the thesis is concerned with the protocols developed in Chapter 5. All three protocols are concerned only with maintaining the consistency of one class of objects, edges, however, any practical graph databases must preserve

the integrity of nodes, labels, and properties. It remains unclear how the protocols developed in Chapter 5 could be efficiently integrated into a wider concurrency control protocol.

Additionally, the work on the Distributed Wait-Hit Protocol in Chapter 3 and the edge consistency protocols in Chapter 5 do not consider replication. Any practical system typically employs some level of replication to enable fault-tolerance or availability. It is unclear how these protocols would be efficiently integrated into a replicated DBMS.

Lastly, the benefits of epoch-based multi-commit appear limited to specific situations. Additionally, the work in Chapter 6 does not address other shortcomings of epoch-based commit, such as increased latency owing to stragglers and/or unbalanced workloads.

7.3 Future Research Directions

In this section, future areas of research are outlined arising from the lessons learnt throughout the PhD.

Additional Implementation and Experiments Extending our evaluation framework described in Chapter 2 would allow for the implementation of distributed concurrency control protocols. Therefore, the performance of the (i) Distributed Wait-Hit Protocol, (ii) Delta protocol, (iii) Deterministic Reciprocal Consistency Protocol, and (iv) Deterministic Edge Consistency Protocol could be empirically evaluated and the results used to corroborate the conclusions drawn in the thesis. Additionally, it would be useful to expose each protocol developed in this thesis to a wider range of workloads, such as, TPC-C [109] or RAMP-TAO [18].

Comparing Distributed Transactions Approaches One observation made when conducting the work in the thesis was the challenge of comparing different distributed transaction protocols. There are several obvious dimensions on which protocols differ, e.g., isolation

level provided, but across protocols that provide the same guarantees it is hard to disambiguate between what can be attributed to superior protocol design, and what is merely a better implementation. This is not helped by the partial evaluation of the breadth of protocols in most studies, with each typically comparing a subset of existing approaches with their "new" approach. The challenge is compounded as different papers use different hardware and system configurations, it truly is oranges vs apples comparison.

Therefore, an interesting area of future research would be developing a framework to reason about the space of distributed transaction protocols. This would help better understand the trade-offs of each design decision and help to better quantify the magnitude of the performance gain each optimization could have. For example, if the read/write access set of a workload can be known, how much will the throughput and latency improve by if I factor this into my protocol design.

Such a framework would allow database designers to compose protocols from the set of optimizations bespoke for their workload or data model, which could lead to the discovery of several interesting new distributed transaction protocols.

Understanding Graph Transactions and Workloads Transactions are important in graph databases, however determining what consistency requirements they need provide to avoid violating graph integrity remains an open question and an interesting area for future research.

Read Atomic isolation [9] was found suitable for TAO [18], but it is unclear how generalizable this is; Sortledon, a recent transactional graph data structure provides Serializable isolation [47]. The uncertainty here can be largely attributed to the lack of publicly available graph transactional workloads which inhibits workload-driven protocol design, existing transactional workloads, e.g., LDBC Interactive [37], only include insert operations.

Another interesting direction is incorporating awareness of application-level invariants and types of common graph operations into the protocols to determine which can be preserved without coordination, e.g., concurrent edge insertion between two nodes is commutative, but

concurrent node deletion and edge insertion requires coordination. Extending the CRDT developed in [10] to a *partitioned* graph database is an appealing direction.

Epoch-based Databases Epoch-based databases are a promising direction for achieving high performance distributed transactions, but current designs have several drawbacks: (i) they are sensitive to imbalanced workloads, one long-running transaction can block the complete epoch, (ii) they are sensitive to straggler nodes, one node slow to reply to a `Prepare` message can increase latency for all transactions, (iii) all transactions executed in an epoch are aborted if one nodes fails, leading to a lot of wasted work, and (iv) retrying the work of a failed epoch could result in a metastable failure [57] if users are operating their databases at a near full load.

One direction for ameliorating drawbacks (i) and (ii) is decentralizing the coordinator node across the cluster, this would allow database nodes that have not interacted with any straggler nodes or ones processing long-running transactions to continue to potentially make forward progress. Case (iii) was addressed in Chapter 6 through epoch-based multi-commit. As regards case (iv), dynamically adjusting the size of the epoch in response to the occupancy levels of the external work queue could help smooth average latency in response to failures.

References

[1] Adya, A. (1999). Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. *PhD Thesis*.

[2] Adya, A., Liskov, B., and O'Neil, P. E. (2000). Generalized isolation level definitions. In Lomet, D. B. and Weikum, G., editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78. IEEE Computer Society.

[3] Alomari, M., Cahill, M. J., Fekete, A. D., and Röhm, U. (2008). The cost of serializability on platforms that use snapshot isolation. In Alonso, G., Blakeley, J. A., and Chen, A. L. P., editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 576–585. IEEE Computer Society.

[4] Angles, R., Antal, J. B., Averbuch, A., Boncz, P. A., Erling, O., Gubichev, A., Haprian, V., Kaufmann, M., Larriba-Pey, J., Martínez-Bazan, N., Marton, J., Paradies, M., Pham, M., Prat-Pérez, A., Spasic, M., Steer, B. A., Szárnyas, G., and Waudby, J. (2020). The LDBC social network benchmark. *CoRR*, abs/2001.02299.

[5] Angles, R., Arenas, M., Barceló, P., Boncz, P. A., Fletcher, G. H. L., Gutiérrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J. F., van Rest, O., and Voigt, H. (2018). G-CORE: A core for future graph query languages. In Das, G., Jermaine, C. M., and Bernstein, P. A., editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432. ACM.

[6] Apache Cassandra (2019). Apache cassandra. http://cassandra.apache.org/. (Accessed on 11/12/2019).

[7] Bailey, N. T. J. (1954). On queueing processes with bulk service. *Journal of the Royal Statistical Society. Series B (Methodological)*, 16(1):80–87.

[8] Bailis, P. et al. (2013). Highly available transactions: Virtues and limitations. *VLDB*.

[9] Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. (2016). Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.*, 41(3):15:1–15:45.

[10] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., and Preguiça, N. M. (2018). IPA: invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418.

[11] Berenson, H., Bernstein, P. A., Gray, J., Melton, J., O'Neil, E. J., and O'Neil, P. E. (1995). A Critique of ANSI SQL Isolation Levels. In Carey, M. J. and Schneider, D. A., editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press.

[12] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[13] Besta, M. et al. (2019a). Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. *CoRR*, abs/1912.12740.

[14] Besta, M., Peter, E., Gerstenberger, R., Fischer, M., Podstawski, M., Barthels, C., Alonso, G., and Hoefler, T. (2019b). Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *CoRR*, abs/1910.09017.

[15] Birke, R., Giurgiu, I., Chen, L. Y., Wiesmann, D., and Engbersen, T. (2014). Failure analysis of virtual and physical machines: Patterns, causes and characteristics. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 1–12. IEEE Computer Society.

[16] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95.

[17] Cheng, A., Shi, X., Kabcenell, A. N., Lawande, S., Qadeer, H., Chan, J., Tin, H., Zhao, R., Bailis, P., Balakrishnan, M., Bronson, N., Crooks, N., and Stoica, I. (2022). Taobench: An end-to-end benchmark for social networking workloads. *Proc. VLDB Endow.*, 15(9):1965–1977.

[18] Cheng, A., Shi, X., Pan, L., Simpson, A., Wheaton, N., Lawande, S., Bronson, N., Bailis, P., Crooks, N., and Stoica, I. (2021). RAMP-TAO: layering atomic transactions on facebook's online TAO data store. *Proc. VLDB Endow.*, 14(12):3014–3027.

[19] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., and Chen, E. (2012). Kineograph: taking the pulse of a fast-changing and connected world. In Felber, P., Bellosa, F., and Bos, H., editors, *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 85–98. ACM.

[20] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In Hellerstein, J. M., Chaudhuri, S., and Rosenblum, M., editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM.

[21] Crooks, N., Burke, M., Cecchetti, E., Harel, S., Agarwal, R., and Alvisi, L. (2018). Obladi: Oblivious serializable transactions in the cloud. In Arpaci-Dusseau, A. C. and Voelker, G., editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 727–743. USENIX Association.

[22] Crooks, N., Pu, Y., Alvisi, L., and Clement, A. (2017). Seeing is Believing: A Client-Centric Specification of Database Isolation. In Schiller, E. M. and Schwarzmann, A. A., editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 73–82. ACM.

[23] Curino, C., Zhang, Y., Jones, E. P. C., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1):48–57.

[24] Das, S., Agrawal, D., and Abbadi, A. E. (2010). G-store: a scalable data store for transactional multi key access in the cloud. In Hellerstein, J. M., Chaudhuri, S., and Rosenblum, M., editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 163–174. ACM.

[25] Dashti, M., John, S. B., Shaikhha, A., and Koch, C. (2017). Transaction repair for multi-version concurrency control. In Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., and Suciu, D., editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 235–250. ACM.

[26] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. In Bressoud, T. C. and Kaashoek, M. F., editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM.

[27] Deutsch, A., Francis, N., Green, A., Hare, K., Li, B., Libkin, L., Lindaaker, T., Marsault, V., Martens, W., Michels, J., Murlak, F., Plantikow, S., Selmer, P., Voigt, H., van Rest, O., Vrgoc, D., Wu, M., and Zemke, F. (2021). Graph pattern matching in GQL and SQL/PGQ. *CoRR*, abs/2112.06217.

[28] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M., and Wood, D. A. (1984). Implementation techniques for main memory database systems. In Yormark, B., editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 1–8. ACM Press.

[29] Dgraph (2021). Dgraph. https://www.dgraph.io/. (Accessed on 16/12/2021).

[30] Dhulipala, L., Blelloch, G. E., and Shun, J. (2019). Low-latency graph streaming using compressed purely-functional trees. In McKinley, K. S. and Fisher, K., editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 918–934. ACM.

[31] Difallah, D. E., Pavlo, A., Curino, C., and Cudré-Mauroux, P. (2013). Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288.

[32] Ding, B., Kot, L., and Gehrke, J. (2018). Improving optimistic concurrency control through transaction batching and operation reordering. *Proc. VLDB Endow.*, 12(2):169–182.

[33] Du, J., Elnikety, S., and Zwaenepoel, W. (2013). Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*, pages 173–184. IEEE Computer Society.

[34] Durner, D. and Neumann, T. (2019). No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 734–745. IEEE.

[35] Eifrem, E. (2016). Graph databases: The key to foolproof fraud detection? *Computer Fraud & Security*, 2016:5–8.

[36] Elnikety, S., Zwaenepoel, W., and Pedone, F. (2005). Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005),26-28 October 2005, Orlando, FL, USA*, pages 73–84. IEEE Computer Society.

[37] Erling, O. et al. (2015). The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630. ACM.

[38] Escriva, R., Wong, B., and Sirer, E. G. (2015). Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815.

[39] Eswaran, K. P., Gray, J., Lorie, R. A., and Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633.

[40] Ezhilchelvan, P. D., Mitrani, I., Waudby, J., and Webber, J. (2019). Design and evaluation of an edge concurrency control protocol for distributed graph databases. In Gribaudo, M., Iacono, M., Phung-Duc, T., and Razumchik, R., editors, *Computer Performance Engineering - 16th European Workshop, EPEW 2019, Milan, Italy, November 28-29, 2019, Revised Selected Papers*, volume 12039 of *Lecture Notes in Computer Science*, pages 50–64. Springer.

[41] Ezhilchelvan, P. D., Mitrani, I., and Webber, J. (2018). On the degradation of distributed graph databases with eventual consistency. In Bakhshi, R., Ballarini, P., Barbot, B., Castel-Taleb, H., and Remke, A., editors, *Computer Performance Engineering - 15th European Workshop, EPEW 2018, Paris, France, October 29-30, 2018, Proceedings*, volume 11178 of *Lecture Notes in Computer Science*, pages 1–13. Springer.

[42] Faleiro, J. M., Abadi, D., and Hellerstein, J. M. (2017). High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624.

[43] Fan, H. and Golab, W. M. (2019). Ocean vista: Gossip-based visibility control for speedy geo-distributed transactions. *Proc. VLDB Endow.*, 12(11):1471–1484.

[44] Fekete, A., Liarokapis, D., O'Neil, E. J., O'Neil, P. E., and Shasha, D. E. (2005). Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528.

[45] Francis, N. et al. (2018). Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445. ACM.

[46] Fraser, K. (2004). *Practical lock-freedom*. PhD thesis, University of Cambridge, UK.

[47] Fuchs, P., Giceva, J., and Margan, D. (2022). Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.*, 15(6):1173–1186.

[48] Garraghan, P., Townend, P., and Xu, J. (2014). An empirical failure-analysis of a large-scale cloud computing environment. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*, pages 113–120. IEEE Computer Society.

[49] Giraph (2021). Giraph. https://giraph.apache.org/. (Accessed on 16/12/2021).

[50] Graphx (2021). Graphx. https://spark.apache.org/graphx/. (Accessed on 16/12/2021).

[51] Gray, J., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. (1976). Granularity of Locks and Degrees of Consistency in a Shared Data Base. In Nijssen, G. M., editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394. North-Holland.

[52] Guerraoui, R. and Wang, J. (2017). How fast can a distributed transaction commit? In Sallinger, E., den Bussche, J. V., and Geerts, F., editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 107–122. ACM.

[53] Guo, J., Cai, P., Wang, J., Qian, W., and Zhou, A. (2019). Adaptive optimistic concurrency control for heterogeneous workloads. *Proc. VLDB Endow.*, 12(5):584–596.

[54] Harding, R., Aken, D. V., Pavlo, A., and Stonebraker, M. (2017). An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564.

[55] Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., Wei, W., Liu, C., Zhang, J., Li, J., Wu, X., Song, L., Sun, R., Yu, S., Zhao, L., Cameron, N., Pei, L., and Tang, X. (2020a). Tidb: A raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084.

[56] Huang, J. and Abadi, D. (2016). LEOPARD: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.*, 9(7):540–551.

[57] Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T., and Charapko, A. (2022). Metastable failures in the wild. In Aguilera, M. K. and Weatherspoon, H., editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 73–90. USENIX Association.

[58] Huang, Y., Qian, W., Kohler, E., Liskov, B., and Shrira, L. (2020b). Opportunities for optimism in contended main-memory multicore transactions. *Proc. VLDB Endow.*, 13(5):629–642.

[59] Huppler, K. (2009). The art of building a good benchmark. In Nambiar, R. O. and Poess, M., editors, *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, volume 5895 of *Lecture Notes in Computer Science*, pages 18–30. Springer.

[60] Issa, S., Viegas, M., Raminhas, P., Machado, N., Matos, M., and Romano, P. (2020). Exploiting symbolic execution to accelerate deterministic databases. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*, pages 678–688. IEEE.

[61] Janusgraph (2020). Janusgraph. https://janusgraph.org/. (Accessed on 10/08/2020).

[62] Kim, K., Wang, T., Johnson, R., and Pandis, I. (2016). ERMIA: fast memory-optimized database system for heterogeneous workloads. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1675–1687. ACM.

[63] Kimura, H. (2015). FOEDUS: OLTP engine for a thousand cores and NVRAM. In Sellis, T. K., Davidson, S. B., and Ives, Z. G., editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 691–706. ACM.

[64] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. D. (2013). MDCC: multi-data center consistency. In Hanzálek, Z., Härtig, H., Castro, M., and Kaashoek, M. F., editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 113–126. ACM.

[65] Kumar, P. and Huang, H. H. (2020). Graphone: A data store for real-time analytics on evolving graphs. *ACM Trans. Storage*, 15(4):29:1–29:40.

[66] Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226.

[67] Kyrola, A., Blelloch, G. E., and Guestrin, C. (2012). Graphchi: Large-scale graph computation on just a PC. In Thekkath, C. and Vahdat, A., editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46. USENIX Association.

[68] Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

[69] Lim, H., Kaminsky, M., and Andersen, D. G. (2017). Cicada: Dependably fast multi-core in-memory transactions. In Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., and Suciu, D., editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 21–35. ACM.

[70] Lin, Q., Chang, P., Chen, G., Ooi, B. C., Tan, K., and Wang, Z. (2016). Towards a non-2pc transaction management in distributed database systems. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM.

[71] Lu, Y., Yu, X., Cao, L., and Madden, S. (2020). Aria: A fast and practical deterministic OLTP database. *Proc. VLDB Endow.*, 13(11):2047–2060.

[72] Lu, Y., Yu, X., Cao, L., and Madden, S. (2021). Epoch-based commit and replication in distributed OLTP databases. *Proc. VLDB Endow.*, 14(5):743–756.

[73] Lu, Y., Yu, X., and Madden, S. (2019). STAR: scaling transactions through asymmetric replication. *Proc. VLDB Endow.*, 12(11):1316–1329.

[74] Maiyya, S., Nawab, F., Agrawal, D., and Abbadi, A. E. (2019). Unifying consensus and atomic commitment for effective cloud data management. *Proc. VLDB Endow.*, 12(5):611–623.

[75] Maria, A. (1997). Introduction to modeling and simulation. In Andradóttir, S., Healy, K. J., Withers, D. H., and Nelson, B. L., editors, *Proceedings of the 29th conference on Winter simulation, WSC 1997, Atlanta, GA, USA, December 7-10, 1997*, pages 7–13. ACM.

[76] Melton, J. (1994). ANSI/ISO SQL-92 Specification. http://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/sql-foundation-aug94.pdf. (Accessed on 06/04/2020).

[77] Mhedhbi, A., Lissandrini, M., Kuiper, L., Waudby, J., and Szárnyas, G. (2021). LSQB: a large-scale subgraph query benchmark. In Kalavri, V. and Yakovets, N., editors, *GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Virtual Event, China, 20 June 2021*, pages 8:1–8:11. ACM.

[78] Mitrani, I. (1982). *Simulation techniques for discrete event systems*, volume 14 of *Cambridge computer science texts*. Cambridge University Press.

[79] Mu, S., Nelson, L., Lloyd, W., and Li, J. (2016). Consolidating concurrency control and consensus for commits under conflicts. In Keeton, K. and Roscoe, T., editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 517–532. USENIX Association.

[80] Narula, N., Cutler, C., Kohler, E., and Morris, R. T. (2014). Phase reconciliation for contended in-memory transactions. In Flinn, J. and Levy, H., editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 511–524. USENIX Association.

[81] Nawab, F., Arora, V., Agrawal, D., and Abbadi, A. E. (2015). Minimizing commit latency of transactions in geo-replicated data stores. In Sellis, T. K., Davidson, S. B., and Ives, Z. G., editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1279–1294. ACM.

[82] Ongaro, D. and Ousterhout, J. K. (2014). In search of an understandable consensus algorithm. In Gibson, G. and Zeldovich, N., editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association.

[83] openCypher (2020). opencypher. https://www.opencypher.org/. (Accessed on 10/08/2020).

[84] Oracle (2022). Property graph query language. https://pgql-lang.org/. (Accessed on 20/04/2022).

[85] Pavlo, A. (2017). What are we doing with our lives?: Nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, page 3. ACM.

[86] Pavlo, A. and Aslett, M. (2016). What's really new with NewSQL? *SIGMOD Rec.*

[87] PostgreSQL (2023). PostgreSQL: The world's most advanced open source relational database. https://www.postgresql.org/. (Accessed on 20/06/2023).

[88] Prasaad, G., Cheung, A., and Suciu, D. (2020). Handling highly contended OLTP workloads using fast dynamic partitioning. In Maier, D., Pottinger, R., Doan, A., Tan, W., Alawini, A., and Ngo, H. Q., editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 527–542. ACM.

[89] Pritchett, D. (2008). BASE: an acid alternative. *ACM Queue*, 6(3):48–55.

[90] Ren, K., Li, D., and Abadi, D. J. (2019). SLOG: serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761.

[91] Ren, K., Thomson, A., and Abadi, D. J. (2014). An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 7(10):821–832.

[92] Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

[93] Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., and Özsu, M. T. (2020). The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.*, 29(2):595–618.

[94] Semeráth, O. et al. (2017). Formal validation of domain-specific languages with derived features and well-formedness constraints. *Softw. Syst. Model.*, 16(2):357–392.

[95] Sheng, Y., Tomasic, A., Sheng, T., and Pavlo, A. (2019). Scheduling OLTP transactions via machine learning. *CoRR*, abs/1903.02990.

[96] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. (2011). Transactional storage for geo-replicated systems. In Wobber, T. and Druschel, P., editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400. ACM.

[97] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. (2007). The end of an architectural era (it's time for a complete rewrite). In Koch, C., Gehrke, J., Garofalakis, M. N., Srivastava, D., Aberer, K., Deshpande, A., Florescu, D., Chan, C. Y., Ganti, V., Kanne, C., Klas, W., and Neuhold, E. J., editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160. ACM.

[98] Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., Bardea, P., Ranade, A., Darnell, B., Gruneir, B., Jaffray, J., Zhang, L., and Mattis, P. (2020). Cockroachdb: The resilient geo-distributed SQL database. In Maier, D., Pottinger, R., Doan, A., Tan, W., Alawini, A., and Ngo, H. Q., editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1493–1509. ACM.

[99] Tang, C., Wang, Z., Zhang, X., Yu, Q., Zang, B., Guan, H., and Chen, H. (2022). Ad hoc transactions in web applications: The good, the bad, and the ugly. In Ives, Z., Bonifati, A., and Abbadi, A. E., editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 4–18. ACM.

[100] Tang, D., Jiang, H., and Elmore, A. J. (2017). Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org.

[101] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160.

[102] Technology, S. I. (2011). Telecommunication Application Transaction Processing (TATP) Benchmark Description. http://tatpbenchmark.sourceforge.net/. (Accessed on 26/04/2022).

[103] Thomson, A. and Abadi, D. J. (2010). The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1):70–80.

[104] Thomson, A., Diamond, T., Weng, S., Ren, K., Shao, P., and Abadi, D. J. (2012). Calvin: fast distributed transactions for partitioned database systems. In Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L., and Fuxman, A., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM.

[105] TiDB (2022). TiDB Transaction Isolation Levels. https://docs.pingcap.com/tidb/dev/transaction-isolation-levels. (Accessed on 09/05/2022).

[106] TigerGraph (2021). Tigergraph. https://www.tigergraph.com/. (Accessed on 16/12/2021).

[107] TinkerPop, A. (2022). Gremlin query language. https://tinkerpop.apache.org/gremlin.html. (Accessed on 20/04/2022).

[108] TitanDB (2020). thinkaurelius/titan: Distributed graph database. https://github.com/thinkaurelius/titan. (Accessed on 10/08/2020).

[109] TPC (2010). TPC Benchmark C, revision 5.11. Technical report, TPC. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[110] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. (2013). Speedy transactions in multicore in-memory databases. In Kaminsky, M. and Dahlin, M., editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 18–32. ACM.

[111] Wang, G., Zhang, L., and Xu, W. (2017). What can we learn from four years of data center hardware failures? In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 25–36. IEEE Computer Society.

[112] Wang, J., Ding, D., Wang, H., Christensen, C., Wang, Z., Chen, H., and Li, J. (2021). Polyjuice: High-performance transactions via learned concurrency control. In Brown, A. D. and Lorch, J. R., editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 198–216. USENIX Association.

[113] Wang, T. and Kimura, H. (2016). Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60.

[114] Wang, Z., Mu, S., Cui, Y., Yi, H., Chen, H., and Li, J. (2016). Scaling multicore databases via constrained parallel execution. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1643–1658. ACM.

[115] Waudby, J. (2022). High performance mixed graph-based concurrency control. In Bao, Z. and Sellis, T. K., editors, *Proceedings of the VLDB 2022 PhD Workshop co-located with the 48th International Conference on Very Large Databases (VLDB 2022), Sydney, Australia, September 5, 2022*, volume 3186 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[116] Waudby, J., Ezhilchelvan, P. D., Webber, J., and Mitrani, I. (2020a). Preserving reciprocal consistency in distributed graph databases. In Fekete, A. D. and Kleppmann, M., editors, *7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2020, Heraklion, Greece, April 27, 2020*, pages 2:1–2:7. ACM.

[117] Waudby, J., Steer, B. A., Karimov, K., Marton, J., Boncz, P. A., and Szárnyas, G. (2020b). Towards testing ACID compliance in the LDBC social network benchmark. In Nambiar, R. and Poess, M., editors, *Performance Evaluation and Benchmarking - 12th TPC Technology Conference, TPCTC 2020, Tokyo, Japan, August 31, 2020, Revised Selected Papers*, volume 12752 of *Lecture Notes in Computer Science*, pages 1–17. Springer.

[118] Waudby, J., Steer, B. A., Prat-Pérez, A., and Szárnyas, G. (2020c). Supporting dynamic graphs and temporal entity deletions in the LDBC social network benchmark's data generator. In Arora, A., Salihoglu, S., and Yakovets, N., editors, *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*, pages 8:1–8:8. ACM.

[119] Webber, J., Ezhilchelvan, P. D., and Mitrani, I. (2019). Modeling corruption in eventually-consistent graph databases. *CoRR*, abs/1904.04702.

[120] Wu, Y., Chan, C. Y., and Tan, K. (2016). Transaction healing: Scaling optimistic concurrency control on multicores. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1689–1704. ACM.

[121] Wu, Y. and Tan, K. (2016). Scalable in-memory transaction processing with HTM. In Gulati, A. and Weatherspoon, H., editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 365–377. USENIX Association.

[122] Yu, X., Pavlo, A., Sánchez, D., and Devadas, S. (2016). Tictoc: Time traveling optimistic concurrency control. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1629–1642. ACM.

[123] Yuan, Y., Wang, K., Lee, R., Ding, X., Xing, J., Blanas, S., and Zhang, X. (2016). BCC: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proc. VLDB Endow.*, 9(6):504–515.

[124] Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurthy, A., and Ports, D. R. K. (2015). Building consistent transactions with inconsistent replication. In Miller, E. L. and Hand, S., editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM.