Tree-HP2PL: A Hierarchical Priority-Based Two-Phase Locking Protocol for Nested Transactions in Distributed Real-Time Database Systems

Meenu
Department of CSE, M. M. M. U. T., Gorakhpur, India
E-mail: myself_meenu@yahoo.co.in

Keywords: concurrency management protocol, distributed real-time database system, transaction commit protocol

Received: February 20, 2025

Modern applications require advanced concurrency control protocols to efficiently manage nested transactions within Distributed Real-Time Database Systems (DRTDBS). This paper introduces Tree-HP2PL, a Hierarchical Priority-Based Two-Phase Locking protocol designed to optimize conflict resolution within nested transactions. The proposed method employs a binary search tree structure for transaction management and incorporates a hybrid strategy combining Depth-First Search (DFS) and parallel Breadth-First Search (BFS) for efficient lock acquisition. Tree-HP2PL is capable of distinguishing and resolving various conflict scenarios, including no conflict, intra-transaction conflicts, and inter-transaction conflicts. To mitigate pseudo-priority inversion, it implements priority inheritance and transaction abort mechanisms as necessary. The protocol's effectiveness was evaluated through experimental simulations conducted in a MATLAB simulation environment, using transaction volumes ranging from 5 to 30 across 10 iterations. Results indicate that execution time remains efficient, with a recorded time of 0.0011 seconds for 5 transactions and showing controlled growth to 0.0049 seconds at 20 transactions. Memory usage demonstrated linear scalability, increasing from 0.006 MB at 5 transactions to 0.016 MB at 30 transactions. Although system throughput decreases under heavier transaction loads, it remains competitive, dropping from 1980 transactions per second (TPS) at 5 transactions to 998 TPS at 30 transactions. Notably, the success ratio consistently maintained a value of 100%, reflecting the protocol's high reliability. In conclusion, Tree-HP2PL effectively addresses the challenges of concurrency control in nested transaction environments by offering a scalable, structured, and conflict-aware locking mechanism. Its robust performance and methodological design validate its applicability in real-time distributed systems, marking it as a significant improvement over existing flat or unstructured locking techniques.

Povzetek: Prispevek predstavlja protokol Tree-HP2PL, ki z hierarhičnim, prioritetnim dvofaznim zaklepanjem učinkovito upravlja gnezdene transakcije v porazdeljenih realnočasovnih podatkovnih bazah ter zagotavlja visoko zanesljivost, razširljivost in nadzorovano porabo virov.

1 Introduction

This section outlines the transition from manual filing systems to computerized solutions, highlighting the development of centralized database systems (CDBS) and the evolution of distributed database systems (DDBS) that connect databases across computer networks. It further emphasizes the rise of real-time database systems (RTDBS) and categorizes transactions in real-time applications as hard, firm, or soft real-time, leading to the emergence of Distributed Real-Time Database Systems (DRTDBS). The discussion includes various transaction models developed by notable researchers and database experts, culminating in the Advanced Transaction model. Within the context of nested transactions, it identifies two main categories: closed nested transactions and open nested transactions and details the three primary advantages of nested transactions—failure independence, intra-transaction parallelism, and enhanced modularity. Lastly, the introduction outlines the paper's structure, encompassing various key areas such as Evolution of Concurrency Control Mechanisms, Challenges in Nested Transactions, Tree-Based Protocol for Nested Transactions, Tree-HP2PL Protocol, TreeHP2PL Model for DRTDBS, Lock Acquisition Algorithm Using DFS and BFS, Performance Analysis of the Tree-HP2PL Algorithm, Future Directions for Database Management Systems, and Key Findings and Future Research Directions. This comprehensive structure seamlessly guides the reader through the scholarly landscape of concurrency control in nested transactions. To provide a complete perspective on concurrency control in nested

transactions, this section first briefly reviews foundational concepts in database management systems (DBMS), which form the basis for advanced mechanisms such as two-phase locking and nested transaction models. This context helps frame the subsequent discussion of state-of-the-art protocols relevant to the Tree-HP2PL approach.

In the realm of information management, the transition from manual filing systems to computerized solutions has been marked by significant milestones. Early file-based and database systems sought to enhance data management efficiency but encountered limitations such as integrated data definitions and restricted data control. The evolution of these systems paved the way for the emergence of centralized database systems (DBS), empowering users to define, create, maintain, and control access to databases [1].

As technological landscapes expanded with the advent of network technologies, distributed database systems (DDBS) evolved to connect databases across computer networks [2]. In recent years, real-time database systems (RTDBS) have gained prominence, particularly in applications with time-sensitive data and transaction-specific time constraints [3]. The critical domains benefiting from RTDBS applications span aircraft control, stock trading, network management, and factory automation [4], [5].

Transactions in real-time applications are categorized as hard, firm, or soft real-time, based on their respective deadline requirements [6]. The extension of these concepts to interconnected databases has given rise to Distributed Real-Time Database Systems (DRTDBS) [7]. A fundamental unit in the context of database transactions, symbolized as T = << t, Ai, Ni >⋯ i=1…n>, involves operations such as data reads and writes and comprises predetermined steps [8]. Transaction models play a crucial role in database management systems, providing a framework for managing and ensuring the consistency and reliability of database transactions. In the dynamic landscape of database systems, various transaction models have been proposed over the years, each aiming to address specific challenges and optimize data management processes. The models, developed by notable researchers and database experts, showcase the diverse strategies employed to achieve efficient transaction processing. Gray's Model emphasizes efficient data management with a flat structure, offering high concurrency and robust recovery, but faces challenges in complexity and locking overhead [9], [10]. The Read/Write Model treats database objects as memory pages, with potential challenges in handling complex computations [11]. Relational Updates focuses on basic operations with atomic execution but may encounter consistency challenges during concurrent modifications [12]. Online/Batch Transactions classifies transactions based on duration, optimizing resource usage but facing challenges in balancing allocation [13]. General/Two-Step/Restricted Two-Step employs flat transactions with read and write classifications, posing challenges in managing specific constraints [14], [15], [16]. Single Uniform Distributed DBMS is a distributed model with a flat structure that prioritizes concurrency control algorithms, while grappling with challenges related to implementation and management complexity [17]. The Flat transaction model represents an advancement from manual file systems, catering to simple and short transactions, yet faces challenges in managing large datasets and lacks robust data integrity checks [18]. The Advanced Transaction model is suitable for today's complex applications but may introduce potential performance overhead and increased complexity in design and implementation [19]. SAGA adopts a hierarchical structure with compensating transactions and limited nesting [20]. The Workflow Model combines transactions hierarchically to fulfil application-specific correctness criteria but faces challenges in specifying and executing workflows efficiently [21]. Dynamic Restructuring introduces operations for adaptive recovery in a hierarchical model, potentially increasing complexity [22]. The Flex Transaction Model extends ACID properties hierarchically, offering enhanced flexibility but facing challenges in complexity and performance [23]. The Nested Model allows nesting, advantageous for complex applications but introducing complexity [24], while the Multilevel Transaction Model presents a balanced subtransactions tree with challenges in flexibility and scalability [25].In the realm of transaction models, these diverse approaches reflect the ongoing pursuit of efficient and effective data management. The evolution from the Flat transaction model to advanced transaction models has been driven by the recognition of the limitations posed by the former in addressing the complexities inherent in engineering applications, particularly in fields like CAD and software engineering. The Flat transaction model, with its single initiation and termination point and basic actions, falls short when confronted with intricate and prolonged processes. Among these advanced models, the Nested transaction model plays a pivotal role in complex engineering applications, offering operational abstractions and flexible ACID property handling. Moss presented the nested transaction model, a framework grounded in the concept of "spheres of control" originally suggested by Bjork and Davies [26], [27]. The Nested transaction model introduces a unique structure, featuring a top-level transaction and subtransactions forming a "transaction tree". Within the nested transaction model, the N-ACID properties are applicable to top-level transactions, with subtransactions possessing a restricted subset of these properties [28] . Two main categories of nested transaction models exist: closed nested transactions and open nested transactions. Closed nested transactions confine the impact of subtransactions to their parent's scope, with the

commitment of subtransactions contingent upon their parent's commitment [20], [29]. In contrast, open nested transactions permit subtransactions to execute and commit autonomously, releasing leaf-level locks early only when the operation semantics are understood [19], [30], [31]. This work adopts the closed model to ensure strict atomicity and consistency, enabling Tree-HP2PL to manage hierarchical locking and conflicts while preserving failure independence, parallelism, and modularity. Unlike flat transactions, nested transactions allow independent failure and rollback of subtransactions without impacting the parent transaction. This characteristic, referred to as "failure independence," mitigates the need for complete transaction rollbacks. Furthermore, nested transactions facilitate both intra-transaction and intertransaction parallelism, enhancing overall performance and modularity. This increased modularity provides benefits such as improved encapsulation and security. The three primary advantages of nested transactions – failure independence, intratransaction parallelism, and enhanced modularity – make them well-suited for demanding scenarios such as real-time, intricate, and distributed applications [32]. Together with other models like Sagas, Multilevel Transaction Model, Dynamic Restructuring, and Workflow Models, these advancements signify a pivotal shift in optimizing system performance for applications that demand a more sophisticated and adaptable transactional framework. In essence, the development of advanced transaction models addresses the pressing need for systems that can effectively handle the demands of complex engineering scenarios. This research paper extensively examines closed nested transaction models, with a specific focus on investigating their commit and concurrency control protocols. The study discerns between parent/child parallelism and sibling parallelism [32].

The subsequent sections seamlessly guide the reader through the scholarly landscape of concurrency control in nested transactions. Section 2 analyses the evolution of concurrency control mechanisms, covering concurrency control protocols, priority inversion in Two-Phase Locking (2PL), and improvements in concurrency control and commit protocols for nested transactions, underscoring their importance in effective concurrency management. Building on this foundation, Section 3 addresses the challenges posed by nested transactions in database management and advocates for customized solutions and further research to enhance performance and maintain data integrity. In Section 4, the discussion shifts to the Tree-Based Protocol for nested transactions, highlighting its hierarchical locking mechanism that enhances concurrency while ensuring both conflict serializability and deadlock prevention. Following this, Section 5 delves into the Tree-HP2PL protocol, a concurrency control mechanism that optimizes nested transaction management by improving system performance and conflict resolution. Section 6 then describes the TreeHP2PL model for Nested Distributed Real-Time Database Systems (DRTDBS), emphasizing its enhancements in concurrency control and resource utilization. To further explore practical implementations, Section 7 presents a lock acquisition algorithm for nested transactions using Depth-First Search (DFS) and parallel Breadth-First Search (BFS). This section addresses three scenarios: no conflict, intra-transaction conflicts within the same tree, and inter-transaction conflicts across different trees, ensuring effective concurrency management. Continuing with performance analysis, Section 8 presents the experimental evaluation of Tree-HP2PL, covering research goals, MATLAB simulation setup, baseline protocol comparison, key performance metrics, and analysis of the protocol's efficiency and scalability. Looking ahead, Section 9 highlights Tree-HP2PL's strong performance and scalability across key metrics, noting trade-offs and future optimization needs. Section 10 highlights essential future directions for database management systems, emphasizing the need for advanced solutions, improved mechanisms, and intelligent optimization to meet the evolving demands of modern applications. Finally, Section 11 emphasizes the paper's key findings and suggests promising future research directions in nested transactions and concurrency control protocols, highlighting the importance of ongoing exploration in this evolving field to enhance efficiency and scalability in transaction management.

2 Related work

This section reviews the evolution of concurrency control mechanisms, particularly focusing on the challenges encountered in nested transactions within distributed real-time databases. It emphasizes the critical need for enhanced protocols to ensure global serializability and real-time performance. The discussion includes concurrency control protocols, highlighting their role in maintaining data consistency and addressing issues like deadlocks and resource contention. Additionally, it examines priority inversion in the Two-Phase Locking (2PL) protocol and explores the evolution of concurrency control and commit protocols tailored for nested transactions. This review underscores significant advancements while pointing out ongoing challenges, such as cascading intra-aborts, thereby establishing the necessity for specialized protocols to improve transaction processing efficiency in distributed environments.

In the realm of database management, the facilitation of concurrent access by multiple users is a critical aspect, fostering advantages such as heightened throughput, reduced transaction waiting times, and overall improved system performance. While read operations typically do not generate conflicts, the simultaneous execution of write operations introduces complexities leading to issues like

lost updates, uncommitted dependencies (or dirty reads), and inconsistent analyses. To counteract these challenges and ensure consistency and reliability in database operations, concurrency control mechanisms are indispensable. Although substantial progress has been made in real-time concurrency control for various transaction models, the distinctive challenges presented by nested transactions in distributed real-time databases require specialized protocols. The proposed work seeks to address these challenges by focusing on global serializability and real-time performance in the context of nested distributed transactions. This initiative contributes significantly to the evolving landscape of concurrency control protocols. Through a thorough examination of existing research, the related work provides a comprehensive overview of the historical evolution of concurrency control protocols and the development of commit protocols tailored for nested distributed real-time database systems. The proposed work aims to build upon these foundational aspects, offering innovative solutions to the specific challenges posed by nested transactions within the dynamic environment of distributed real-time databases. In essence, this research strives to bridge existing gaps, pushing the boundaries of knowledge, and contributing valuable insights to the intricate domain of concurrency control in distributed real-time databases.

To sum up, this research builds on the evolution of concurrency control mechanisms by addressing the unique challenges of nested transactions in distributed real-time databases. It offers innovative solutions to enhance global serializability and real-time performance, contributing valuable insights to the field and bridging critical gaps in existing protocols.

2.1 Concurrency control protocols

This section outlines the importance of concurrency control protocols in managing transactions in multi-user environments. It briefly discusses Pessimistic and Optimistic protocols, emphasizing their role in ensuring data integrity and consistency. The section also highlights challenges and solutions in real-time and distributed systems, stressing the need for robust concurrency control protocols.

Concurrency control is a critical aspect of database management systems to ensure transactions are executed in a controlled and consistent manner in multi-user environments. This involves managing access to shared resources to prevent conflicts and maintain data integrity. Serializability is a key concept, aiming to identify schedules where transactions can run concurrently without conflicts [14].This ensures that the database state mirrors a sequential execution. Two primary approaches to concurrency control are Pessimistic and Optimistic [33]. Pessimistic approaches, including lock-based algorithms, order-based algorithms, and hybrid approaches, assume conflicts are likely. Two-Phase Locking (2PL), assume conflicts are likely and address issues such as lost updates, uncommitted dependencies, and inconsistent analysis. Optimistic approaches, including optimistic locking [34] and timestamp ordering [35], [36], assume conflicts are infrequent and involve read, validation, and write phases. The Two-Phase Locking (2PL) protocol consists of growing and shrinking phases to ensure conflict serializability [37]. If every transaction strictly follows the Two-Phase Locking protocol in a schedule, the resulting schedule is guaranteed to demonstrate conflict serializability. Challenges associated with 2PL, such as cascading rollbacks, are addressed by rigorous 2PL and strict 2PL variants. The applicability of 2PL extends to distributed databases through variants such as Primary site 2PL [38], Primary Copy 2PL [39], Voting 2PL (Majority Consensus 2PL) [35], and Distributed Two-Phase Locking (D2PL), contributing to the mitigation of data redundancy. Practical applications of D2PL include System R* [40]and NonStop SQL [41], [42], [43]. Distributed Two-Phase Locking (D2PL) algorithms play a crucial role in minimizing data redundancy in distributed database systems. Timestamp Ordering (TO) is another approach where pessimistic TO prioritizes transactions based on timestamps to address conflicts. Basic TO ensures conflict serializability but introduces overhead, mitigated by Conservative TO algorithms [44]. Multiversion TO allows multiple versions of data items, enhancing concurrency [45], [46]. When a transaction reads data, the system chooses the version based on the transaction's timestamp, exploring it as an alternative to nested concurrency control protocols [47], [48], [49]. Conservative TO Algorithms introduce a delay to minimize transaction restarts and potential deadlock situations, with specific implementations providing solutions for reducing system overhead. Optimistic TO algorithms link timestamps exclusively to transactions, providing higher concurrency but facing challenges like increased storage overhead. Real-time database systems (RTDBS) face challenges such as restarts, useless restarts, and chained blocking [50]. Transaction priority schemes, including FCFS, EDF, MSTF, and SJF, aim to provide predictability in these systems [51], [52], [53].Understanding and implementing effective concurrency control protocols is crucial for maintaining the integrity and reliability of database systems. The diverse approaches outlined showcase the flexibility and adaptability of these protocols in addressing various challenges associated with concurrent transactions. As technology evolves, the choice of concurrency control becomes increasingly nuanced, with considerations for both pessimistic and optimistic scenarios. This overview serves as a starting point for delving into the fascinating realm of concurrency control, offering insights into the tools and techniques that underpin the foundation of robust and concurrent database systems.

Ultimately, the various concurrency control protocols discussed highlight their essential role in ensuring consistency and integrity in multi-user database environments. The choice between pessimistic and optimistic approaches, along with their adaptations in real-time and distributed systems, underscores the need for tailored solutions to address specific challenges. As technology evolves, these protocols will continue to play a crucial role in maintaining reliable and efficient transaction processing in modern database systems.

2.2 Priority inversion in 2pl (two-phase locking)

This section addresses priority inversion in Two-Phase Locking (2PL) within Real-Time Database Systems (RTDBS). It highlights the challenge of high-priority transactions being delayed by locks held by lower-priority transactions. Solutions, such as the Priority Abort Protocol and Priority Inheritance Protocol, are discussed to mitigate this issue and ensure timely execution of high-priority transactions.

In Real-Time Database Systems (RTDBS), priority inversion poses a significant challenge within the Two-Phase Locking (2PL) protocol. This occurs when high-priority transactions wait for locks held by lower-priority transactions, potentially causing delays and missed deadlines [54]. Two forms of priority inversion exist: bounded and unbounded [55]. To address this, solutions such as the Priority Abort Protocol (2PL-HP) involve aborting lower-priority lock holding transactions, allowing high-priority transactions to acquire them [4]. This solutions aim to ensure timely execution of high-priority transactions in RTDBS, though they may introduce certain drawbacks, such as increased restart frequency in the case of 2PL-HP.Another approach is the Priority Inheritance Protocol, which propagates priority from higher- to lower-priority transactions, preventing indefinite waits caused by lower-priority transactions holding required locks [56]. In summary, addressing priority inversion in 2PL within the context of RTDBS is crucial for ensuring the timely execution of high-priority transactions and meeting stringent real-time constraints. The outlined solutions, such as the Priority Abort Protocol and Priority Inheritance Protocol, provide avenues for tackling this challenge and maintaining the desired priority assignment goals in real-time database systems.

Overall, addressing priority inversion in Two-Phase Locking (2PL) within Real-Time Database Systems (RTDBS) is essential for ensuring that high-priority transactions meet their deadlines. The discussed solutions, including the Priority Abort Protocol and the Priority Inheritance Protocol, offer effective strategies to mitigate delays caused by lower-priority transactions, thereby enhancing the performance and reliability of RTDBS in real-time environments.

2.3 Evolution of concurrency control and commit protocols in nested transactions

This section explores the evolution of concurrency control and commit protocols for nested transactions. It presents an overview of various protocols in Table 1, highlighting their unique features, benefits, and challenges. The ongoing development of these protocols underscores the pursuit of scalability and efficiency, promising advancements in managing nested transactions. Database professionals are encouraged to consider these protocols for enhanced performance.

The landscape of database management has witnessed remarkable developments in concurrency control protocols, especially those tailored for nested transactions. This Table 1 presents a comprehensive overview of various protocols designed by different developers to address the unique challenges posed by nested transaction scenarios. Each protocol brings its own set of merits, demerits, and addresses specific challenges, contributing to the diverse toolkit available for database administrators. Let's delve into the details of these protocols, exploring their origins, distinctive features, and the challenges they aim to overcome. From fundamental algorithms to specialized approaches for distributed real-time systems, these protocols showcase the ingenuity of developers in the pursuit of efficient and robust nested transaction concurrency control.

Table 1: Overview of concurrency control and commit protocols for nested transactions.

S. No.	Protocol	Description	Advantages	Limitations	Improvement via Tree-HP2PL
1.	2PL-NT Concurrency control protocol [24]	Developed Concurrency Control Algorithm for Nested Transactions by integrating Eswaran's [37] two-phase locking mechanism.	Ensures serializability	High blocking and abort rates	DFS/BFS traversal reduces blocking and enhances concurrency

2.	Exclusive Locking Algorithm for Nested Transactions [57] [58]	Extending the multi-granularity algorithm tailored for nested transactions.	Precise access control	High locking overhead, reduced concurrency	Binary tree model reduces granularity complexity
3.	Formalization and Generalized Locking Algorithm [59]	Formal proof of Moss's read/write algorithm, introducing a generalized read-update locking algorithm.	Rigor and correctness	Complex to implement and scale	Simplifies lock modelling via structured hierarchy
4.	Extension of Multi-granularity Algorithms [60]	Extended multi-granularity algorithms for nested transactions.	Reduces lock table size	Reduced concurrency under escalated states	Avoids escalation, uses consistent hierarchical locking
5.	Serialization Graph Construction for Nested Transactions [61]	Serialization graph construction based on I/O automaton models, contributing to nested transaction systems.	Theoretical correctness	Computationally expensive in large-scale systems	Replaces graphs with traversal for efficiency
6.	Formalization of Concurrency Control Algorithms [62]	Formalizing concurrency control algorithms for open and safe nested transactions, utilizing an I/O approach.	Strong conflict resolution foundation	Lacks practical scalability	Practical DFS/BFS model supports real-time enforcement
7.	Application of Nested Transactions in KBMSs [63]	Applies nested transactions in knowledge base systems (KBMSs).	High abstraction flexibility	Inefficient in real-time scenarios	Supports real-time nested DBMS with structured control
8.	Extension of Gifford's basic quorum consensus algorithm for data replication to incorporate nested transactions and transactions aborts [64]	Extends Gifford's algorithm for data replication to include nested transactions.	Fault-tolerant consistency	High coordination overhead	Focuses on intra/inter conflict resolution without quorum delays
9.	Concurrency Control Algorithm for B-trees within nested transactions [65]	Proposed concurrency control algorithm specifically designed for B-trees within nested transaction models.	Optimized tree access for indexing	Limited to index concurrency	Tree-HP2PL generalizes to all data types, not just index structures
10.	Concurrency Control Algorithm utilizing linear hash	Presented a concurrency control algorithm utilizing linear hash structures for nested transactions.	Lightweight hashing model	Limited lock semantics for complex nesting	Extends to complex locking and transaction trees

	structures for nested transactions. [66]				
11.	Multi-version Timestamp Concurrency Control [45]	Introduces a multi-version timestamp concurrency control algorithm for nested transactions.	High read throughput	Storage overhead, version management	Avoids versioning, uses efficient priority-based lock conflict resolution
12.	2PL-NT-HP concurrency control protocol [67] [68]	Introduces concurrency control protocol for Nested Distributed Real-Time Database Systems	Deadline awareness	Still experiences priority inversion	Uses priority inheritance and abort handling for inversion-free scheduling
13.	S-PROMPT commit protocol [67] [68]	Specifically crafted commit protocol for nested transactions to address intra-abort cascade issues.	Ensures commit order [69].	Image storage overhead, added latency	Resolves conflicts early without image storage

The diverse array of concurrency control protocols for nested transactions showcased in the Table 1 highlights the ongoing efforts to refine and optimize database management in intricate scenarios. Each protocol addresses specific challenges while introducing new dimensions to the field. As developers continue to innovate and refine their approaches, the future promises even more sophisticated solutions for nested transaction concurrency control. The ongoing pursuit of scalability, compatibility, and efficiency in diverse database architectures will undoubtedly shape the next generation of protocols. This exploration serves as a snapshot of the current landscape, inviting database professionals to consider the nuances of each protocol in their quest for resilient and high-performance nested transaction management.

To sum up, the evolution of concurrency control and commit protocols for nested transactions highlights significant advancements in database management. The diverse protocols outlined in Table 1 showcase unique strategies to address challenges while offering distinct advantages. Ongoing innovations aim for scalability, compatibility, and efficiency, inviting database professionals to leverage these solutions for improved nested transaction management.

In summary, Section 2 analyses concurrency control mechanisms and commit protocols, revealing significant advancements in managing transaction complexities in distributed real-time databases. However, existing commit protocols frequently result in cascading intra-aborts within nested transactions, underscoring the urgent need for innovative solutions to enhance global serializability and real-time performance. This section emphasizes the necessity for further research into specialized protocols tailored for nested transactions, aiming to provide insights that contribute to efficient and reliable transaction processing in complex database environments.

3 Issues and challenges of nested transactions

This section examines the challenges of nested transactions in database management and proposes tailored solutions to address these complexities. It emphasizes the need for ongoing research to enhance performance and ensure data integrity in complex transactional environments.

The implementation of nested transactions introduces a unique set of challenges that demand innovative solutions for efficient database management. In the Table 2 below, we delve into specific issues encountered within the nested transaction model, along with their corresponding descriptions and proposed resolutions. Explore how developers and researchers have addressed challenges such as handling intra-transactions parallelism, prioritization policies for subtransactions, and the detection of deadlocks within nested transactions. Each issue is accompanied by thoughtful resolutions, ranging from concurrency control mechanisms to specialized protocols designed to preserve database coherence and ensure global serializability. This comprehensive overview sheds light on the complexities of managing nested transactions, providing insights into the ongoing efforts to enhance performance, maintain consistency, and overcome obstacles inherent in intricate transactional structures

Table 2: Nested transaction issues and remedies.

S. No	Issues in Nested Transaction Model	Description	Issues Resolution
I.	Handling of intra-transactions parallelism [32]	In nested transactions, both intra-transaction parallelism and inter-transaction parallelism are present. Intra-transaction parallelism in nested transaction models poses synchronization challenges during concurrent subtransaction execution, increasing the risk of deadlock and resource contention.	To address this issue, implement the following solutions: Concurrency Control Mechanisms, Isolation Levels, Transaction Scheduling Policies, Resource Management, Performance Monitoring, and Tuning, Documentation and Best Practices.
II.	Priority assignment policy for subtransactions [18]	Data sharing among subtransactions can cause delays, so prioritization is needed to avoid execution delays.	To address this issue, implement a priority assignment policy for subtransactions to ensure efficient execution and avoid delays.
III.	Detecting deadlock in nested transactions [70]	Nested transactions require both waits-for-lock and waits-for-commit relations for deadlock detection.	To address this issue, extend deadlock detection mechanisms to consider both waits-for-lock and waits-for-commit relations in the context of nested transactions.
IV.	Priority assignment policy [4] [71]	Traditional protocols for RTDBS often neglect transaction priorities.	To address this issue, develop and incorporate priority assignment policies in protocols for real-time database systems to consider transaction priorities.
V.	Concurrency control protocol for subtransactions [67], [68]	Existing concurrency control protocols designed for flat transactions may lead to issues when applied in a nested environment.	To address this issue, design concurrency control protocols specifically tailored for the concurrency of parent and child transactions in nested transaction models.
VI.	Commit protocol for subtransactions [67], [68]	Existing commit protocols may face challenges when applied to concurrent parent and child transactions in nested models.	To address this issue, develop commit protocols that address the unique challenges presented by concurrent execution of parent and child transactions in nested environments.
VII.	Handling of priority inversion [54]	Priority Inheritance and Priority Abort Protocols are used for priority inversion in advanced transaction models.	To address this issue, implement Priority Inheritance and Priority Abort Protocols to address priority inversion issues in nested transaction models.
VIII.	Preserving database coherence [28]	Nested transactions are not atomic; thus, the definition of "N-ACID" properties (nested-all-or-nothing, nested consistency, nested isolation, and nested durability) is necessary.	To address this issue, define and ensure N-ACID properties (N-A, N-C, N-I, N-D) for nested transactions to preserve database coherence.
IX.	Adjusting transaction recovery according to control structure [72]	Nested transactions have more splitable execution modules and finer control for recovery and concurrency compared to flat transactions.	To address this issue, adapt transaction recovery mechanisms to account for the specific control structure and finer granularity of recovery in nested transactions.
X.	Insurance of global serializability [67]	Nested transaction models have not been fully applied to real-time database systems, raising concerns about the global serializability of distributed real-time nested transactions.	To address this issue, research and develop methods to ensure the global serializability of distributed real-time nested transactions in real-time database systems.
XI.	Handling of transaction parameters in nested transaction [31].	The number of leaves and levels in nested transactions can impact performance.	To address this issue, consider and optimize transaction parameters, such as the number of leaves and levels, to enhance the system's performance in nested transaction models.

The Table 2 above encapsulates the multifaceted nature of challenges in nested transaction models and offers practical resolutions that span a spectrum of database management aspects. As we navigate the intricate landscape of nested transactions, it becomes evident that a tailored approach is crucial for ensuring seamless execution and maintaining data integrity. The ongoing research and development efforts highlighted in this exploration underscore the dynamic nature of database systems. By continually refining protocols, addressing deadlock scenarios, and introducing priority assignment policies, the database community strives to unlock the full potential of nested transactions. As technology evolves, so too will our ability to tackle the complexities of nested

transactions, fostering a future where robust and efficient database management becomes an integral part of intricate transactional scenarios.

In summary, the exploration of issues and challenges in nested transactions highlights the complexities inherent in managing such systems. By identifying key challenges and proposing targeted solutions, this section emphasizes the necessity for ongoing research and development to enhance performance and ensure data integrity. As the field evolves, the database community must continue to innovate and refine protocols to effectively navigate the intricacies of nested transactions, ultimately paving the way for more robust and efficient database management in complex environments.

4 Tree-based protocol for nested transactions

This section presents the Tree-Based Protocol for nested transactions, highlighting its hierarchical locking strategy, which enhances concurrency while ensuring conflict serializability and deadlock avoidance.

In the intricate realm of nested transactions, the choice of concurrency control protocols plays a pivotal role in shaping the efficiency and reliability of database systems. The Tree-Based Protocol for concurrency control in nested transactions, employs Exclusive Locks Only with a Parent-Child Relationship structure. Locks can be released during the transaction, promoting high concurrency. The approach assures Conflict Serializability, prevents deadlocks hierarchically, and reduces waiting times. Despite potential Locking Overhead, careful optimization is required. Overall, the protocol offers adaptability and effectiveness in various nested transaction scenarios with critical concerns like conflict serializability and deadlock prevention.

Thus, the Tree-Based Protocol for nested transactions enhances concurrency while ensuring conflict serializability and deadlock prevention. Its hierarchical locking mechanism promotes efficiency, though careful optimization is needed to manage potential locking overhead. Overall, it offers a strong framework for improving nested transaction management in complex database systems.

5 Tree-HP2PL: A tree-based concurrency control protocol for nested DRTDBS

This section explores the Tree-HP2PL protocol, a concurrency control mechanism designed to optimize nested transaction management. The Table 3 delves into key aspects of this protocol, highlighting its innovative features and advantages in enhancing system performance and conflict resolution, making it suitable for the complexities of nested transactions.

In the realm of nested transactions, the choice of concurrency control mechanisms profoundly impacts system efficiency and reliability. The Table 3 delves into key aspects of a sophisticated concurrency control protocol designed to optimize nested transaction management. TreeHP2PL protocol leverages a compatible tree data structure, employing a binary search tree for streamlined conflict resolution and efficient traversal. It addresses both intra-transaction and inter-transaction conflicts through innovative priority inheritance mechanisms, while strategically mitigating pseudo priority inversion. Explore the nuances of this protocol, from its utilization of graph traversal algorithms for lock-holding transaction identification to the advantages it brings in terms of performance optimization and simplified conflict resolution. Each aspect contributes to the overall efficiency of the system, making it well-suited for the intricacies of nested transactions. Tree-HP2PL ensures conflict serializability by acquiring locks using structured DFS and parallel BFS traversal, which prevents cycles and preserves the correct ordering of conflicting operations. Conflict serializability refers to the ability to reorder non-conflicting operations in a concurrent schedule to match a valid serial execution. Let's dissect the protocol's characteristics, highlighting its key points, and understanding how it contributes to the optimization of nested transaction management.

Table 3: Key Aspects of the TREEHP2PL in nested transactions

S.No.	Aspect	Description	Key Points
1.	Compatible Tree Data Structure	Utilizes a binary search tree for optimized system performance, enabling straightforward in-order traversal for conflict resolution.	Performance Optimization: Compatible tree structure enhances efficiency over array-based approaches.
2.	Intra-Transaction Data Conflict Handling	Employs binary search tree traversal method to streamline conflict resolution, avoiding multiple insertions into the waitlist of data items.	Simplified Conflict Resolution: Binary search tree traversal simplifies intra-transaction conflict handling.
3.	Priority Inheritance Mechanism	Upgrades priorities of nodes in a subtree for incoming high-priority sub-	Effective Priority Inheritance: Ensures effective resolution of intra- and inter-transaction

		transactions, minimizing extensive priority comparisons and enhancing efficiency.	conflicts through priority inheritance.
4.	Inter-Transaction Data Conflict Resolution	Uses priority inheritance and priority abort protocols at the parent transaction level to address conflicts, prioritizing nested transactions.	Effective Resolution of Inter-Transaction Conflicts: Priority inheritance and priority abort protocols employed for conflict resolution.
5.	Handling Pseudo Priority Inversion	Strategically employs priority inheritance and priority abort protocols, with a preference for priority inheritance in nested transactions to address pseudo priority inversion.	Priority Inversion Mitigation: Addresses pseudo priority inversion, enhancing overall system performance.
6.	Lock-Holding Transaction Identification	Employs Depth-First Search (DFS) and parallel Breadth-First Search (BFS) as independent yet complementary graph traversal methods. DFS enables deep dependency exploration, while parallel BFS supports concurrent conflict resolution in wider transaction hierarchies.	Effective Lock-Holding Transaction Identification: Graph traversal algorithms assist in identifying lock holding transactions
7.	Advantages and Contributions	- Performance Optimization: Adoption of a compatible tree structure enhances efficiency over array-based approaches. - Simplified Conflict Resolution: Binary search tree traversal simplifies intra-transaction conflict handling. - Effective Priority Inheritance: Ensures effective resolution of intra- and inter-transaction conflicts through priority inheritance. - Priority Inversion Mitigation: Addresses pseudo priority inversion, enhancing overall system performance.	Performance benefits through optimized data structure. Streamlined conflict resolution within and between transactions. Efficient handling of priority inheritance and pseudo priority inversion.

The detailed exploration of the optimized concurrency control protocol for nested transactions reveals a sophisticated approach to addressing conflicts and enhancing system performance. As we step beyond the Table 3, it's crucial to consider the broader implications and potential applications of the protocol. The advantages and contributions highlighted, ranging from performance optimization to effective resolution of inter-transaction conflicts, underscore the protocol's versatility in diverse nested transaction scenarios. The strategic use of priority inheritance mechanisms and graph traversal algorithms for lock-holding transaction identification showcases a holistic approach to concurrency control. Database administrators and developers navigating complex transactional environments are encouraged to assess how these nuanced features align with their specific requirements. Considerations extend beyond the technical aspects, touching upon scalability, adaptability, and ease of integration within existing database architectures. This exploration invites further scrutiny, experimentation, and adaptation as the database community strives to optimize nested transaction management. Beyond the structured presentation, there lies a landscape of possibilities for refinement and customization based on unique system demands and evolving technology.

In summary, the Tree-HP2PL protocol presents a robust and innovative approach to managing concurrency in nested transactions, significantly enhancing system performance and conflict resolution. By utilizing a compatible tree structure and incorporating advanced mechanisms such as priority inheritance and graph traversal algorithms, the protocol effectively addresses both intra- and inter-transaction conflicts. The insights from Table 3 reinforce its potential to optimize nested transaction management, making it a valuable consideration for database administrators and developers facing the challenges of complex transactional environments. Overall, this exploration underscores the importance of selecting appropriate concurrency control mechanisms to ensure the reliability and efficiency of nested transactions.

6 TREE-HP2PL model

This section outlines the experimental evaluation of the Tree-HP2PL protocol for nested transactions. It begins with the research objectives and simulation setup, detailing the use of MATLAB to model varying transaction volumes and conflict scenarios. The next part introduces the baseline protocol (2PLNTHP) and the evaluation criteria used for comparison. Key performance metrics—such as

execution time, response time, memory usage, system throughput, and success ratio—are then defined. The final part provides a detailed analysis of these metrics to assess the efficiency, scalability, and reliability of the Tree-HP2PL protocol in real-time distributed environments.

The TreeHP2PL model, designed for Nested Distributed Real-Time Database Systems (DRTDBS), comprises several key components that collectively contribute to its effectiveness in managing concurrency control. The system consists of four main components: the Source Module, responsible for simulating external workload; the Transaction Manager, handling transactions and assigning priorities; the Concurrency Control Manager, managing conflicts and coordinating transactions; and the Resource Manager, controlling physical resources for efficient allocation. Together, they ensure effective transaction management, modelling, and resource utilization within the computing environment.

Ultimately, the TreeHP2PL model significantly enhances the management of concurrency in Nested Distributed Real-Time Database Systems. By integrating its core components, the model optimizes transaction management and resource utilization, ensuring robust performance in complex distributed environments. This comprehensive approach positions the model as a valuable solution for addressing the challenges of concurrency control in real-time applications.

7 Proposed algorithm

This section introduces a lock acquisition algorithm for nested transactions, focusing on conflict management at various levels. It employs Depth-First Search (DFS) and parallel Breadth-First Search (BFS) to enhance scalability and conflict resolution. The algorithm is structured to handle three key conflict scenarios: no conflict, intra-transaction conflicts within the same transaction tree, and inter-transaction conflicts among different transaction trees. The algorithm emphasizes efficiency and adaptability through a systematic approach, which includes initialization, lock checks, and conflict resolution steps. By employing nested functions, the algorithm ensures a thorough mechanism for lock acquisition that can be evaluated and customized for practical implementation in real-world database systems. The combination of DFS and parallel BFS strategies demonstrates a commitment to optimizing transaction management within nested structures, encouraging further exploration and refinement by database professionals.

7.1 Lock acquisition

Effective lock acquisition is a critical aspect of nested transaction management, ensuring both concurrency and data integrity. The proposed algorithm outlined below provides a comprehensive strategy for handling various conflict scenarios within nested transactions.

This algorithm addresses three key cases:

1) No Conflict Among Transactions

In the absence of conflicts, the algorithm grants locks to transactions that have not yet acquired them.

2) Intra-Transactions Conflict

When conflicts arise within the same transaction tree, the algorithm employs a sophisticated conflict resolution mechanism utilizing Depth-First Search (DFS) and parallel Breadth-First Search (BFS) strategies.

3) Inter-Transactions Conflict

In situations where conflicts occur among different transaction trees, the algorithm leverages the same conflict resolution mechanism used for intra-transaction conflicts. The algorithm employs nested functions, utilizing DFS and parallel BFS to resolve conflicts efficiently. It initializes data structures, transaction trees, priorities, and shared data to facilitate a systematic approach to lock acquisition. As we delve into the algorithm's intricacies, it becomes apparent that the proposed approach is not only geared towards resolving conflicts but is also designed for scalability and adaptability within diverse nested transaction scenarios. Let's explore the detailed steps and mechanisms embedded in the algorithm, shedding light on its capacity to optimize lock acquisition and contribute to the overall efficiency of nested transaction systems.

Here are the step-by-step instructions for the lock acquisition algorithm.

1. Step 1: Initialize Global Variables:
   - Create a set named visited to keep track of visited transactions.
   - Create a dictionary named transactionTree to represent the transaction tree.
   - Create a list named transactionPriorities to store transactions prioritized for conflict resolution.
   - Create a dictionary named sharedData to store the status of shared data.
2. Step 2: Check if any Transaction has not Acquired a Lock:
   - Implement A Function or check condition to verify if any transaction has not acquired a lock.
3. Step 3: Grant a Lock to a Transaction:

- Implement a function or set of conditions to grant a lock to a transaction if it has not acquired one.

4. Step 4: Handle Intra-Transaction Conflict Resolution:
   - Initialize necessary data structures and variables.
   - Initialize the transaction tree and transaction priorities.
   - Implement a depth-first search (DFS) and parallel breadth-first search (BFS) combination for resolving intra-transaction conflicts within the same transaction tree.
   - Prioritize transactions based on a specified order for resolution.
5. Step 5: Handle Inter-Transaction Conflict:
   - Defer to the mechanism used for intra-transaction conflict resolution to handle conflicts among different transaction trees.
6. Step 6: Main Algorithm Execution:
   - Check if any transaction has not acquired a lock and grant the lock if necessary.
   - Handle intra-transaction conflicts using the previously defined mechanism.
   - Handle inter-transaction conflicts by deferring to the intra-transaction conflict resolution mechanism.

These steps provide a high-level overview of the lock acquisition algorithm.

7.2 Algorithm

The lock acquisition algorithm adopts a systematic and adaptive approach to manage conflicts within nested transactions. Its versatility is highlighted through its ability to address conflicts at different levels within the transaction hierarchy.

Input:

T: Set of nested transactions

transactionTree: Hierarchical structure of parent-child transactions

sharedData: Lock status for each data item

visited: Set to keep track of visited transactions

Output:

Locks granted or conflicts resolved

BEGIN

CASE 1: No Conflict Among Transactions

For each transaction Ti ∈ T do

If Ti has not acquired a lock and sharedData[Ti.dataItem] is free then

grant_lock(Ti)

End If

End For

CASE 2: Intra-Transaction Conflict (Same Tree)

Procedure handle_intra_transaction_conflict()

initialize_data_structures()

initialize_transaction_tree()

initialize_transaction_priorities()

For each transaction in transactionPriorities do

initialize_shared_data(transaction)

End For

Start conflict resolution:

start_DFS_or_parallel_BFS(root_node, selected_transaction, sharedData)

Function DFS(transactionIndex, sharedData)

resolve_conflict(transactionIndex, sharedData)

Function resolve_conflict(transactionIndex, sharedData)

If transactionIndex ∉ visited then

visited ← visited ∪ {transactionIndex}

process_transaction(transactionIndex)

For each child ∈ transactionTree[transactionIndex] do

```
If child is unresolved then
DFS(child, sharedData)
Else
resolve_intra_transaction_conflict(transactionIndex, child)
End If
End For
End If

Function parallel_BFS(sharedData)
queue ← ∅
initialize_data_structures()
While queue ≠ ∅ do
transaction ← dequeue(queue)
For each child ∈ transactionTree[transaction] do
If child is unresolved then
process_transaction(child)
Else
resolve_intra_transaction_conflict(transaction, child)
End If
enqueue_unvisited_children(queue, transaction)
    End For
    End While

CASE 3: Inter-Transaction Conflict (Different Trees)
Procedure handle_inter_transaction_conflict()
Call handle_intra_transaction_conflict()

END
```

The intricacies of the proposed lock acquisition algorithm unveil a systematic and adaptive approach to managing conflicts within nested transactions. As we move beyond the algorithm, it's essential to reflect on the broader implications and considerations for implementation. The algorithm's versatility shines through its ability to handle conflicts at different levels, ensuring that transactions progress smoothly even in complex nested structures. The combination of DFS and parallel BFS strategies showcases a commitment to efficiency and scalability, crucial for real-world applications. Database administrators and developers are encouraged to evaluate how this algorithm aligns with the specific demands of their nested transaction scenarios. Considerations extend beyond the algorithmic steps, touching upon factors like system performance, adaptability to varying transaction loads, and ease of integration into existing databases. This exploration invites further examination, experimentation, and customization as the database community strives to optimize lock acquisition in the intricate landscape of nested transactions. Beyond the structured presentation of the algorithm, lies a realm of possibilities for refinement, tailoring the approach to unique system requirements and evolving technological landscapes.

In summary, the proposed lock acquisition algorithm for nested transactions offers a robust framework for managing conflicts through a systematic approach that integrates Depth-First Search (DFS) and parallel Breadth-First Search (BFS) strategies. By effectively addressing no conflicts, intra-transaction conflicts, and inter-transaction conflicts, the algorithm ensures efficient lock management while maintaining data integrity. Its adaptability to various transaction scenarios highlights its potential for real-world application, encouraging database professionals to customize and refine the algorithm to meet specific system needs. This exploration serves as a foundation for further research and development in optimizing transaction management within complex nested structures.

8 Implementation and results

This section outlines the experimental evaluation of the Tree-HP2PL protocol for nested transactions. It begins with the research objectives and simulation setup, detailing the use of MATLAB to model varying transaction volumes and conflict scenarios. The next part introduces the baseline protocol (2PLNTHP), and the evaluation criteria used for comparison. Key performance metrics—such as execution time, response time, memory usage, system throughput, and success ratio—are then defined. The final part provides a detailed analysis of these metrics to assess the efficiency, scalability, and reliability of the Tree-HP2PL protocol in real-time distributed environments.

Tree-HP2PL algorithm's performance in nested transactions is assessed focusing on system throughput, success ratio, response time, execution time, and memory usage. These metrics offer deep insights into protocol performance and efficiency in handling concurrent transactions. Specifically for tree-based protocols such as Tree-HP2PL in nested transactions, focusing on these metrics enables a comprehensive evaluation of their real-world applicability and effectiveness. To validate our approach, we implemented the algorithm in a MATLAB simulation environment and conducted extensive

experiments using various transaction scenarios [73]. The study investigates the suitability of Tree-HP2PL in nested transaction scenarios, employing the Breadth-First Search (BFS) algorithm for tree and graph data structure traversal and search operations. Additionally, this protocol utilizes Depth-First Search (DFS) and parallel Breadth-First Search (BFS) strategies to effectively manage conflicts and improve scalability. Evaluating system throughput helps in understanding the protocol's ability to process transactions efficiently within a given timeframe. Success ratio indicates transaction completion rates and reliability. Response time measures system responsiveness to transaction requests, impacting user experience. Execution time reflects transaction processing efficiency and resource utilization. Assessing memory usage is crucial for scalability and resource management. Table 4 summarizes these performance metrics, providing a quick reference. The subsequent sections delve into a detailed analysis of Tree-HP2PL's average system throughput, success ratio, response time, execution time, and memory usage under varying transaction workloads. The protocol is scrutinized as it processes varying numbers of transactions (5, 10, 15, 20, 25, and 30) over ten iterations, simulating nested transactions in each iteration. Table 4 summarizes the analysis of average execution time trends across different transaction numbers over ten iterations, while Table 6 presents a similar analysis of average memory consumption. Tables 7 to 9 depict the trends in average system throughput, success ratio, and response time under varying transaction volumes and iterations. Analyzing the nested transaction performance within the experimental transaction tree involves a detailed examination of the shapes illustrated in Figures 1–6. Memory-resident and disk-resident databases represent different approaches to database storage, each with its own set of considerations. In the case of a memory-resident database, the entire database is assumed to be stored in the main memory. This simplifies the study as it eliminates the need to model disks or manage I/O scheduling. On the other hand, in a disk-resident database, we assume a more realistic scenario where the database is stored on secondary storage (disks). Disk-resident database approach is chosen for our study because it provides a practical representation of how data is typically stored and accessed. Additionally, the database may be on a single disk or partitioned across multiple disks, with each disk having its own service queue.

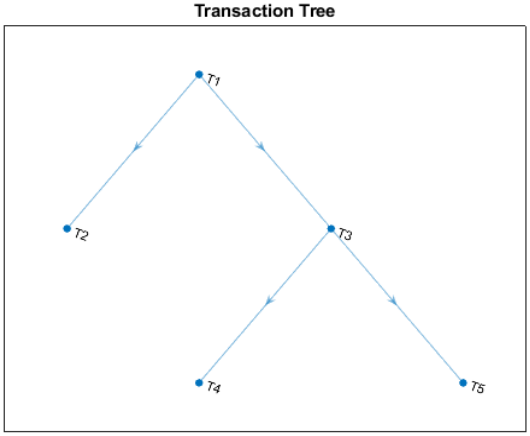

Figure 1: Five node transaction structure.

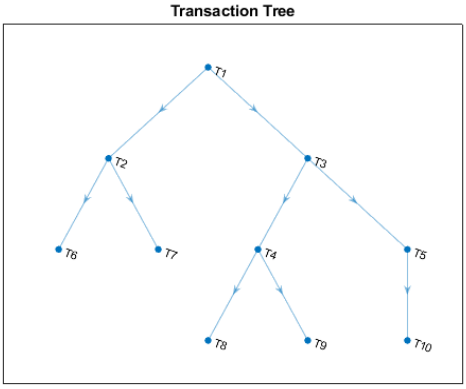

Figure 2: Ten node transaction structure.

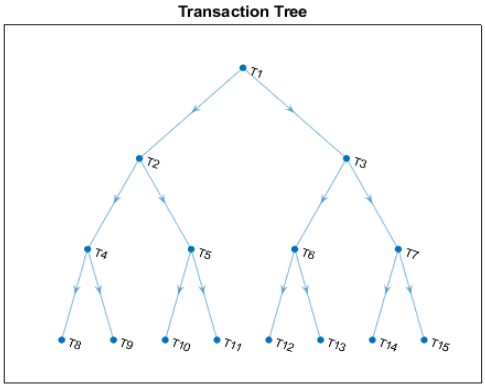

Figure 3: Fifteen node transaction structure.

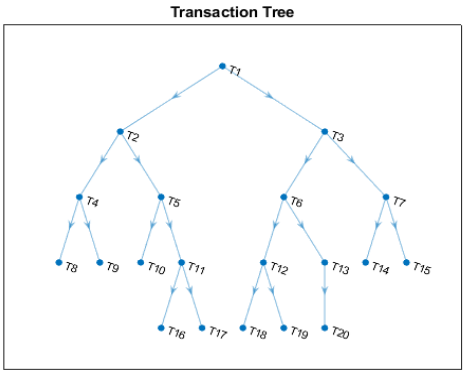

Figure 4: Twenty node transaction structure

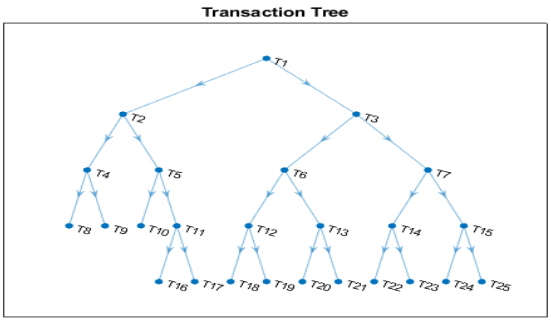

Figure 5: Twenty-five node transaction structure.

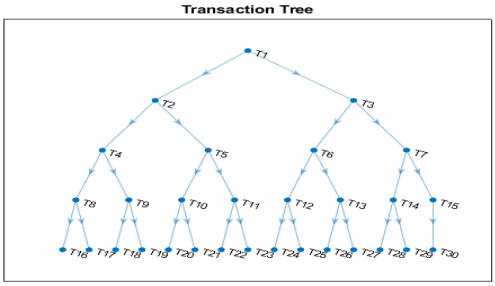

Figure 6: Thirty node transaction structure.

Overall, this section underscores the significance of the Tree-HP2PL algorithm within tree-based concurrency control protocols for nested transactions. It emphasizes the need to analyze various performance metrics in both memory-resident and disk-resident database contexts, thereby establishing a practical framework for understanding how these protocols function in real-world scenarios. The insights gained pave the way for future research to enhance the efficiency and applicability of concurrency control methods in nested transactions.

8.1 Research objectives and simulation setup

This section outlines the goals of evaluating Tree-HP2PL and describes the simulation environment used to assess its performance under nested transaction workloads.

This study aims to evaluate the effectiveness of the proposed Tree-HP2PL protocol for managing nested transactions in distributed real-time database systems (DRTDBS). The research is guided by the following questions☹1) How does Tree-HP2PL impact system throughput, execution time, memory usage, and response time across varying transaction volumes?(2) Can a hybrid Depth-First Search (DFS) and parallel Breadth-First Search (BFS) lock acquisition strategy enhance conflict detection and resolution in nested transactions?(3) What trade-offs arise from using hierarchical priority inheritance in real-time concurrency control? To answer these questions, Tree-HP2PL was implemented and evaluated in MATLAB R2023b on a system with an Intel Core i7-11800H processor (2.3 GHz), 16 GB RAM, and Windows 11 Pro (64-bit). The simulation modelled nested transaction workloads as binary trees with up to four levels of nesting. In each iteration, 5 to 30 transactions were executed, across 10 iterations. Conflict scenarios were introduced using random transaction priorities and shared dependencies to simulate intra- and inter-transaction contention. The workload abstracts real-world scheduling patterns but excludes specific database queries or logic. The five measured performance metrics include success ratio, system throughput, average execution time, average response time, and memory usage. The simulation assumes firm deadlines, a centralized lock manager with no network delays, and failure due to lock denial or timeout only, thus isolating concurrency control behavior under realistic DRTDBS-like constraints.

8.2 Baseline protocols and evaluation criteria

This section compares the proposed Tree-HP2PL protocol with the 2PLNTHP protocol using key performance metrics, based on analytical evaluation and literature data.

To evaluate the effectiveness of the proposed Tree-HP2PL protocol, its performance was compared analytically with one of the established concurrency control protocols for nested transactions: 2PLNTHP (Two-Phase Locking for Nested Transactions with High Priority), which incorporates transaction priorities into the traditional two-phase locking mechanism to improve deadline adherence. This protocol was selected due to their relevance to real-time nested transaction environments in distributed database systems. Since source code or implementation models for this protocol is not publicly available, direct benchmarking was not feasible. Therefore, the comparative analysis is based on performance trends reported in prior literature [67], [68] and is supported by analytical reasoning based on their documented operational characteristics. The evaluation considered five key performance metrics: average execution time, average response time, system throughput

(transactions per second), memory usage, and success ratio. While Tree-HP2PL was implemented and tested in MATLAB, data for the baseline protocols were derived from related studies under comparable transaction volumes and workload conditions, with the aim of identifying performance patterns rather than providing exact numerical equivalence.

8.3 Performance metrics

This section discusses key performance metrics for evaluating tree-based concurrency control protocols in nested transactions. In the evaluation of concurrency control protocols for transactions, various performance metrics are employed to assess the effectiveness and efficiency of the system [71]. These metrics provide a comprehensive understanding of different aspects of performance, aiding stakeholders in making informed decisions and implementing improvements. Table 4 summarizes the key performance metrics used to evaluate concurrency control protocols for transactions. These metrics play a crucial role in assessing the efficiency, success rate, and overall performance capacity of the system under evaluation.

Table 4: Performance metrics for concurrency control protocols

S.No.	Performance Metric	Description
1.	Average System Throughput	Measures the average number of transactions processed per unit of time, indicating system capacity.
2.	Average Success Ratio	Average ratio of successful transactions to total transactions submitted, showing system success rate.
3.	Average Response Time	Average elapsed time from command initiation to completion, assessing transaction performance.
4.	Average Execution Time	Average typical duration for completing each transaction, reflecting system efficiency.
5.	Average Memory Usage	Average amount of memory utilized per transaction, indicating system memory efficiency.

In summary, the evaluation of tree-based concurrency control protocols for nested transactions, specifically using the Tree-HP2PL algorithm, highlights the importance of key performance metrics. The analysis of these metrics provides valuable insights into the system's efficiency in handling nested transactions. This research underscores the effectiveness of BFS in managing transaction complexity and provides a foundation for further exploration of concurrency control mechanisms.

8.4 Performance metrics analysis

This section explores key performance metrics—System Throughput, Success Ratio, Response Time, Execution Time, and Memory Usage to assess system efficiency and optimize concurrency control protocols.

Before delving into the analysis of these performance metrics, it's essential to highlight their significance in evaluating concurrency control protocols. These metrics, including Average System Throughput, Average Success Ratio, Average Response Time, Average Execution Time, and Average Memory Usage, provide a comprehensive view of the system's efficiency, success rate, transaction performance, and resource utilization.

Understanding these metrics is key to making informed decisions and implementing improvements in concurrency control protocols for transactions. The following section will delve into a detailed analysis of each metric, shedding light on their impact and implications for system performance.

Thus, these metrics are crucial for optimizing concurrency control protocols by improving transaction management, resource utilization, and overall system performance.

8.4.1 Analysis of execution time

This section analyses average execution time across varying transaction volumes, focusing on the system's processing efficiency. It highlights trends and fluctuations in execution times, providing insights into system behavior under different workloads.

Table 5: Execution time for simulation.

Number of Transactions	Average Execution Time (seconds)
5	0.001125305
10	0.001727616
15	0.002153896
20	0.004914052
25	0.00276094
30	0.004611946

The analysis of average execution time provides crucial insights into the system's processing efficiency under varying transaction workloads. Table 5 summarizes the recorded average execution times for different

numbers of transactions over ten iterations, offering a detailed view of how execution time scales with transaction volume. This data is visually presented in Figure 7, showcasing the execution time trends, and highlighting any notable patterns or outliers across the transaction workload spectrum.

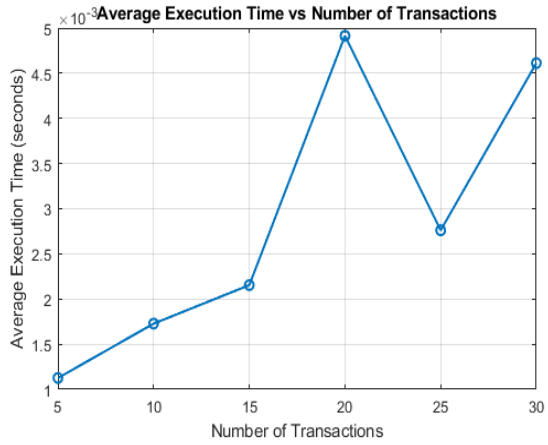

Figure 7: Execution time for simulation.

The analysis of average execution time reveals efficient processing at lower transaction volumes, with execution times as follows: 0.00799869 seconds for 5 transactions, 0.01193914 seconds for 10 transactions, and 0.01595354 seconds for 15 transactions. However, a significant increase is observed at 20 transactions, with execution time reaching 0.03583039 seconds, indicating potential strain or bottlenecks under heavier workloads. The execution time then slightly decreases for 25 transactions (0.02059183 seconds) and increases again for 30 transactions (0.03478573 seconds).

Overall, the analysis indicates efficient processing at lower transaction volumes, while a significant increase at higher volumes suggests potential bottlenecks, highlighting the need for optimizations to improve system performance under heavier workloads.

8.4.2 Analysis of memory usage

This section analyses average memory usage across varying transaction volumes, highlighting trends and the need for efficient memory management strategies to address increasing resource demands.

Table 6: Memory usage for simulation

Number of Transactions	Average Memory Usage (MB)
5	0.006073418
10	0.007920208
15	0.009810867
20	0.011739674
25	0.013970184
30	0.015982914

The analysis of average memory usage provides valuable insights into the system's memory, requirements under varying transaction workloads. Table 6 presents the recorded average memory usage values for different numbers of transactions over ten iterations, illustrating how memory usage scales with transaction volume. This data is visually represented in Figure 8, offering a clear depiction of memory usage trends, and highlighting any notable patterns or fluctuations across different transaction volumes.

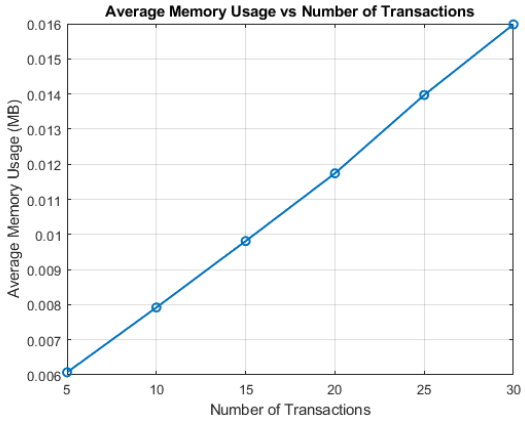

Figure 8: Memory usage for simulation.

Memory usage exhibits a linear growth pattern, increasing from 0.006073418 MB for 5 transactions to 0.015982914 MB for 30 transactions. This consistent increase emphasizes the importance of efficient memory management strategies to handle escalating resource demands effectively.

In summary, the analysis reveals a linear increase in memory usage with higher transaction volumes, underscoring the need for efficient memory management strategies to accommodate escalating resource demands and maintain optimal system performance.

8.4.3 Analysis of system throughput

This section examines average system throughput to evaluate processing capacity under varying transaction workloads, highlighting trends and emphasizing the need for optimizations to address potential bottlenecks.

The assessment of average system throughput provides essential insights into the system's processing capacity under varying transaction workloads. Table 7 summarizes the average throughput values recorded for different numbers of transactions across ten iterations, showcasing how throughput changes with transaction volume. This information is visually depicted in Figure 9, offering a graphical representation of throughput trends, and highlighting any significant fluctuations or trends observed across the range of transaction volumes.

Table 7: System throughput for simulation

Number of Transactions	Average System Throughput (TPS)
5	1980.904429
10	1563.57411
15	1235.399642
20	721.2252535
25	1374.468946
30	998.2999971

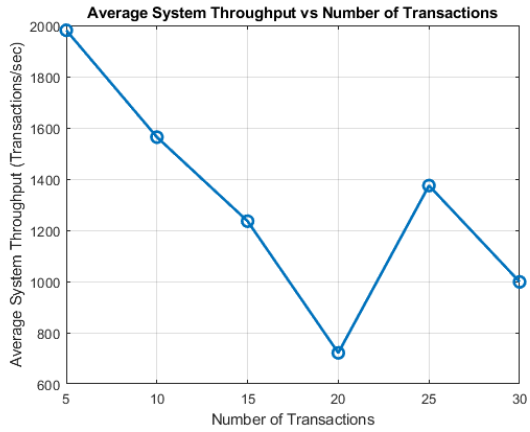

Figure 9: System Throughput for simulation.

System throughput shows a decline as transaction volumes increase, dropping notably from 1980.904429 TPS at 5 transactions to 721.2252535 TPS at 20 transactions. This decline suggests potential processing constraints or bottlenecks under higher transaction volumes, highlighting the need for optimizations to maintain optimal processing capacity.

Thus, the analysis indicates a significant decline in average system throughput with increasing transaction volumes, highlighting the need for optimizations to address potential bottlenecks and maintain optimal processing capacity.

8.4.4 Analysis of success ratio

This section evaluates the average success ratio to assess the system's reliability and transaction completion rates across varying workloads, highlighting the system's effectiveness in handling transactions.

The evaluation of average success ratio offers critical insights into the system's reliability and transaction completion rates across varying transaction workloads. Table 8 compiles the average success ratio values recorded for different numbers of transactions over ten iterations, providing a comprehensive view of success rates across transaction volumes. This data is visually represented in Figure 10, presenting success ratio trends graphically and highlighting any notable patterns or variations observed in the system's performance regarding transaction completions.

Table 8: Success ratio for simulation

Number of Transactions	Average Success Ratio
5	1
10	1
15	1
20	1
25	1
30	1

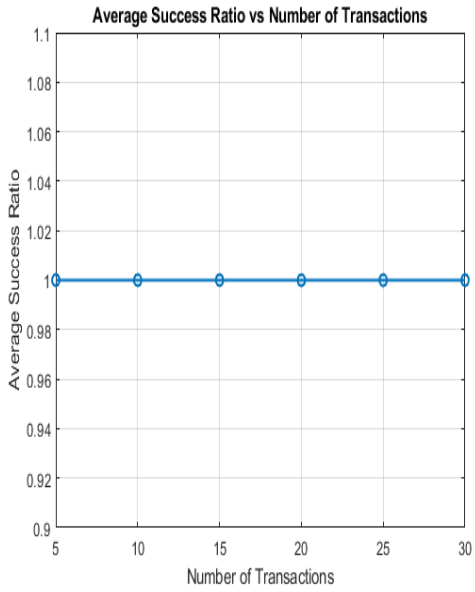

Figure 10: Success ratio for simulation.

The success ratio remains consistently high at 1 (or 100%) across all transaction volumes, indicating reliable transaction completion rates without errors or failures.

In summary, the analysis of the average success ratio underscores the system's reliability and effectiveness in managing transactions across varying workloads, indicating a robust performance in achieving transaction completions without errors.

8.4.5 Analysis of response time

This section analyses average response time, highlighting fluctuations in system responsiveness as transaction workloads change. It emphasizes the correlation between transaction volume and response time, suggesting implications for processing efficiency and resource management.

The assessment of average response time reveals insights into the system's responsiveness under varying transaction workloads. Table 9 summarizes the recorded average response time values for different numbers of transactions over ten iterations, illustrating how response time fluctuates with transaction volume. This information is graphically depicted in Figure 11, providing a visual

representation of response time trends, and highlighting any significant changes or patterns observed in the system's responsiveness across different transaction volumes.

Table 9:Response time for simulation.

Number of Transactions	Average Response Time (seconds)
5	0.00799869
10	0.01193914
15	0.01595354
20	0.03583039
25	0.02059183
30	0.03478573

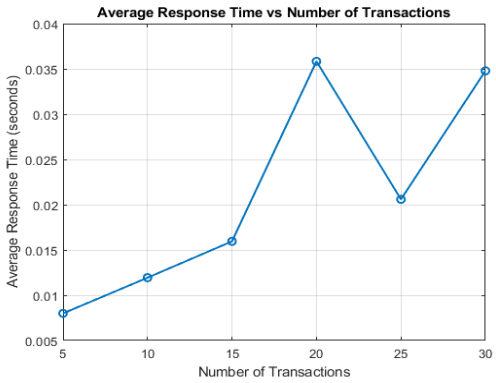

Figure 11: Response time for simulation.

Response time follows a pattern like execution time, with lower values at lower transaction volumes and increased times at higher volumes. Specifically, response times are 0.00799869 seconds for 5 transactions, 0.01193914 seconds for 10 transactions, and 0.01595354 seconds for 15 transactions. A notable increase is observed at 20 transactions (0.03583039 seconds), suggesting potential processing complexities or resource constraints impacting system responsiveness. The response time then decreases slightly for 25 transactions (0.02059183 seconds) and increases again for 30 transactions (0.03478573 seconds).

Thus, the analysis of average response time reveals that system responsiveness declines as transaction volumes increase, indicating potential processing challenges. Effective resource management is essential to maintain optimal performance under varying workloads.

Overall, the in-depth analysis of the system's performance across different transaction volumes revealed significant patterns and trends. The findings highlighted the effectiveness of the Tree-HP2PL algorithm, leveraging Depth-First Search (DFS) and parallel Breadth-First Search (BFS) strategies, in managing nested transactions and complex structures. The BFS approach demonstrated positive correlations among key metrics—average system throughput, success ratio, response time, execution time, and memory usage—relative to transaction volume. The analysis emphasized the critical role of workload management strategies, showcasing efficient processing at lower volumes with consistently high success rates, indicative of robust reliability. However, increased workloads led to notable spikes in execution and response times, suggesting potential bottlenecks impacting responsiveness. The linear growth in memory usage underscored the necessity for effective memory management, while the decline in system throughput with higher workloads highlighted the need for optimizations to maintain processing capacity. Ultimately, enhancing resource utilization and system responsiveness are imperative for sustaining performance across varying workloads.

9 Discussion

This section discusses the Tree-HP2PL protocol's strong performance and scalability based on five key metrics—success ratio, system throughput, execution time, response time, and memory usage—while noting trade-offs and opportunities for future real-world optimization.

The Tree-HP2PL protocol was evaluated through MATLAB simulation environment across transaction volumes ranging from 5 to 30, focusing on five performance metrics: success ratio, system throughput, average execution time, average response time, and memory usage. The protocol consistently achieved a 100% success ratio, demonstrating robust conflict resolution without deadlocks or aborts across all tested scenarios. System throughput declined gradually from approximately 1980 transactions per second (TPS) at 5 transactions to about 998 TPS at 30 transactions, reflecting controlled degradation under increasing load. Average execution time increased from 0.0011 seconds at 5 transactions to 0.0049 seconds at 20 transactions, while average response time remained below 0.017 seconds up to 15 transactions but rose to 0.036 seconds as contention and nesting depth increased. Memory usage exhibited a linear growth trend, from 0.006 MB to 0.016 MB, in line with the number of active transactions and the size of transaction tree structures maintained during execution.

Upon deeper analysis, the sharp rise in execution time beyond 20 transactions is primarily attributed to the algorithmic complexity inherent in Tree-HP2PL's conflict resolution strategy, which employs Depth-First Search (DFS) and parallel Breadth-First Search (BFS). The worst-case time complexity of both DFS and BFS components is O (V + E), where V is the number of transactions and E is the number of transaction dependencies. As transaction volume increases, so does the traversal workload, with more frequent updates to inherited priorities and increased BFS queue sizes, leading to higher processing time. The

observed linear memory usage is reasonable for the current scale but may become super-linear with larger workloads (e.g., 100+ transactions), due to recursive depth growth, expanding BFS queues, and cumulative metadata storage for conflict tracking and priority management. The decline in throughput is linked not only to increased lock contention but also to architectural constraints in the simulation. MATLAB's single-threaded environment lacks parallelism and asynchronous processing, meaning lock requests are handled sequentially and transaction overlap is restricted—conditions that do not fully reflect real-world DBMS capabilities. As a result, throughput suffers from compounding delays caused by synchronous lock acquisition and conflict resolution.

While Tree-HP2PL introduces some computational overhead, particularly through priority inheritance and hierarchical traversal, it avoids cascading aborts and image storage overheads associated with protocols like 2PLNTHP and S-PROMPT. These comparisons are analytical, based on performance trends and behavior reported in prior literature [67], [68] rather than from direct experimental benchmarking. Overall, Tree-HP2PL demonstrates a strong balance of correctness, scalability, and responsiveness under increasing load in nested transaction environments. However, future work should focus on deploying the protocol in multi-threaded or distributed settings to validate its operational robustness and further optimize its performance in realistic real-time systems.

In addition to these empirical findings, the complexity and design of the Tree-HP2PL lock acquisition algorithm also warrant discussion. The algorithm combines Depth-First Search (DFS) and parallel Breadth-First Search (BFS) to manage lock conflicts in nested transactions. The worst-case time complexity of both DFS and BFS components is O (V + E), where V is the number of transactions and E is the number of transaction dependencies. DFS enables efficient deep traversal of hierarchical parent-child relationships to resolve intra-transaction conflicts, while parallel BFS handles inter-transaction or sibling-level conflicts concurrently. This combination enhances scalability by addressing both vertical and horizontal conflict patterns within transaction trees. However, as transaction volume increases, particularly beyond 20 transactions, traversal overhead and dynamic lock and priority state maintenance introduce execution time and memory overhead. These trends are evident in the simulation results, which show increased execution and response times at higher transaction volumes. While this trade-off impacts scalability, the DFS-BFS approach provides a deterministic, modular method of resolving conflicts—particularly beneficial in complex, real-time nested systems. In comparison, hybrid optimistic-pessimistic locking schemes may reduce overhead in low-contention environments but often suffer from validation delays and less predictable behavior under load. Tree-HP2PL's use of priority inheritance, though effective in mitigating priority inversion, introduces additional computational costs due to recursive priority propagation and increased memory usage. Nonetheless, this strategy ensures fairness and deadlock avoidance, which are critical for correctness in distributed real-time systems. Overall, Tree-HP2PL's algorithm design strikes a practical balance between responsiveness, correctness, and scalability, though further optimizations may be required for extremely high-throughput scenarios.

The Tree-HP2PL protocol was evaluated in a MATLAB simulation environment, chosen for its strengths in rapid algorithm prototyping, matrix manipulation, and visualization. While MATLAB does not replicate the full concurrency behavior of modern DBMSs like PostgreSQL or Oracle, it serves as a practical platform for modelling complex transaction hierarchies, simulating resource contention, and analyzing conflict resolution strategies in a controlled setting. One aspect influencing performance is the distribution of transaction priorities. Tree-HP2PL performs consistently under uniform or balanced priority distributions, with minimal aborts and stable throughput. However, when priorities are heavily skewed or dynamically shifting—such as a concentration of high-priority transactions—lock contention intensifies, and priority inversion risks increase. The protocol's integrated priority inheritance and selective abort mechanisms mitigate these effects to maintain fairness, but performance sensitivity to workload characteristics remains, suggesting a need for adaptive tuning in real-time systems. Beyond simple priority-based conflict resolution, Tree-HP2PL handles deadlocks through a dual detection strategy using DFS and parallel BFS to identify dependency cycles. This approach analyses both waits-for-lock and waits-for-commit graphs to detect and resolve complex deadlocks typical in nested environments. Under high-contention scenarios, where multiple transactions simultaneously request overlapping resources, the system escalates to priority inheritance and carefully chosen aborts to break deadlock chains. The use of parallel BFS distributes the conflict resolution load across sibling nodes, improving responsiveness. Nonetheless, performance degradation can occur if contention remains persistently high, underscoring the importance of incorporating resource-aware scheduling or load-throttling mechanisms in practical implementations. Overall, Tree-HP2PL demonstrates robustness across variable workloads and complex conflict scenarios, though its effectiveness can be further enhanced through integration with adaptive and distributed runtime strategies.

10 Future directions and challenges

This section identifies critical future areas in database management systems, focusing on advanced extended transaction models, adaptive concurrency control, and overhead optimization. It also emphasizes the potential of hybrid approaches, enhancements in distributed real-time systems, and the integration of machine learning for intelligent optimization, addressing the evolving needs of modern applications.

In the rapidly advancing field of database management systems, future directions and challenges play a pivotal role in shaping the landscape of technology. As we delve into the next phase of innovation, several key areas emerge where further exploration and development are essential to meet the evolving demands of modern applications and systems.

10.1 Extended transaction models

As we look ahead, an exciting avenue for research involves the exploration of more intricate extended transaction models. Tailoring these models for specific application domains, such as CAD/CAM and software engineering, can lead to heightened efficiency and improved support for specialized workflows. For instance, the Tree-HP2PL protocol could be enhanced with semantic-aware locking strategies, allowing it to adapt to domain-specific object hierarchies and dependencies in structured design systems.

10.2 Adaptive concurrency control

The future beckons the development of adaptive concurrency control mechanisms. These mechanisms would dynamically adjust based on transaction characteristics and workload patterns, ushering in a new era of responsiveness and performance optimization within database systems. A promising direction is to extend Tree-HP2PL with a feedback-based adaptation mechanism, where the system monitors runtime metrics (e.g., lock wait times or subtree density) and switches between DFS and parallel BFS strategies, accordingly, optimizing performance in real-time.

10.3 Optimizing overhead

Addressing the overhead associated with concurrency control protocols, especially in optimistic approaches, remains a pressing challenge. Future research efforts could concentrate on optimizing storage and processing requirements, paving the way for more streamlined and resource-efficient database systems. One approach could involve lock compaction, in which sibling subtransactions with similar lock access patterns share metadata, reducing redundancy and memory footprint without sacrificing isolation guarantees.

10.4 Hybrid concurrency control

In the quest for versatile and efficient solutions, there is a need to investigate hybrid approaches that combine the strengths of both optimistic and pessimistic concurrency control. Such hybrid models could offer a balanced approach, enhancing flexibility and performance across diverse application scenarios. For example, a hybrid version of Tree-HP2PL could be devised where critical parent-level transactions follow the pessimistic path, while leaf-level subtransactions execute under optimistic assumptions, validated upon commit.

10.5 Distributed real-time database systems

The evolution of distributed real-time database systems is critical in response to the increasing prevalence of distributed computing and the growing demands of real-time applications. Ongoing research in this domain should continue to refine and expand the capabilities of these systems to meet the challenges of an interconnected, real-time world. Enhancing Tree-HP2PL with deadline-aware scheduling algorithms—such as Earliest Deadline First (EDF) could significantly improve its responsiveness under time-constrained, distributed workloads.

10.6 Machine learning integration

The integration of machine learning techniques into database systems holds immense promise for the future. Exploring ways to leverage machine learning for adaptive concurrency control and predicting transaction behaviors could usher in a new era of intelligent, self-optimizing database systems, further enhancing their adaptability to dynamic workloads. Specifically, Tree-HP2PL could be extended with a reinforcement learning (RL) module trained to adjust transaction priorities based on system history and conflict likelihood, thereby preventing bottlenecks before they occur.

In summary, the future of database management systems hinges on addressing emerging challenges through innovative solutions, such as advanced transaction models, adaptive concurrency control, and the integration of machine learning. These developments are essential for enhancing system efficiency, responsiveness, and adaptability to meet the dynamic requirements of modern applications and industries. Continued research in these areas—especially through the development of semantic locking, adaptive traversal strategies, hybrid control models, and AI-powered prioritization—will pave the way for more robust and intelligent database systems.

11 Conclusion

This section emphasizes the Tree-HP2PL protocol's strong performance in nested transactions while identifying the need for better resource management to

mitigate increased execution and response times with higher transaction volumes.

This paper provides a comprehensive evaluation of the Tree-HP2PL concurrency control protocol within nested transactions, showcasing its exceptional performance across key metrics. The experimental results underscore the protocol's efficiency in managing crucial aspects such as system throughput, success ratio, response time, execution time, and memory usage within nested transaction scenarios. The analysis demonstrates that the performance of the Tree-HP2PL protocol is directly influenced by the number of transactions, with noticeable impacts on execution time, memory usage, system throughput, and response time. As the number of transactions increases, several key performance metrics experience notable changes. The average execution time shows a gradual rise, indicating increased processing overhead and potential bottlenecks, particularly evident at 20 transactions. Memory usage exhibits a linear growth pattern, directly correlating with transaction volume, highlighting the system's escalating resource demands. System throughput demonstrates a decrease, suggesting potential processing constraints under higher transaction volumes. Despite these challenges, the success ratio remains consistently high, showcasing the protocol's reliability in completing transactions successfully. However, response time increases with transaction volume, emphasizing the need for optimized system responsiveness. These findings underscore the importance of efficient resource management and conflict resolution mechanisms within the Tree-HP2PL protocol, especially when handling varying transaction workloads. Looking ahead, addressing these performance nuances, and exploring alternative algorithms can further enhance the protocol's scalability, efficiency, and overall performance in real-world transaction management scenarios.

Overall, the analysis of the Tree-HP2PL concurrency control protocol within nested transactions highlights its strong performance and reliability, reflected in consistently high success ratios. However, the increasing execution and response times at higher transaction volumes indicate a pressing need for improved resource management and optimization strategies. To enhance the protocol's scalability and efficiency, future research should focus on addressing these performance challenges and exploring alternative algorithms that can better accommodate varying transaction workloads in practical applications

References

[1] Connolly, T., & Begg, C. (1998). *Database systems: A practical approach to design, implementation and management* (2nd ed.). Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

[2] Özsu, M. T., & Valduriez, P. (2020). *Principles of distributed database systems* (4th ed.). Springer, pp. 1–674. https://doi.org/10.1007/978-3-030-26253-2.

[3] Ramamritham, K. (1993). Real-time databases. *Distributed and Parallel Databases*, 1(2), 199–226. https://doi.org/10.1007/BF01264051.

[4] Abbott, R., & Garcia-Molina, H. (1988). Scheduling real-time transactions: A performance evaluation. *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*.

[5] Stankovic, J. A., & Zhao, W. (1988). On real-time transactions. *Proceedings of the ACM SIGMOD Conference*, March 1988. https://doi.org/10.1145/44203.44204.

[6] Choudhary, M., & Haritsa, J. M. (1992). Data access scheduling in firm real-time database systems. *International Journal of Real-Time Systems*, 4(3). https://doi.org/10.1007/BF00365312.

[7] Mukkamala, M., & Shanker, U. S. (2008). Distributed real-time database systems: Background and literature review. *International Journal of Distributed and Parallel Databases*, 23(2), 127–149. https://doi.org/10.1007/s10619-008-7024-5.

[8] Gray, J. (1980). A transaction model. *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*. https://doi.org/10.1007/3-540-10003-2_78.

[9] Gray, J. (1978). Notes on database operating systems. In *Operating Systems—An Advanced Course* (Lecture Notes in Computer Science, Vol. 60, pp. 393–481). Berlin: Springer-Verlag. https://doi.org/10.1007/3-540-08755-9_9.

[10] Gray, J. (1981). The recovery manager of the System R database manager. *ACM Computing Surveys*, 13, 223–244. https://doi.org/10.1145/356842.356847.

[11] Rastogi, S., Bhargava, H. K., & Silberschatz, A. (1993). On correctness of non-serializable executions. *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. https://doi.org/10.1145/153850.153859.

[12] Abiteboul, S., & Vianu, V. (1988). Equivalence and optimization of relational transactions. *Journal of the ACM*, 35, 70–120. https://doi.org/10.1145/42267.42271.

[13] Gray, J. (1987). Why do computers stop and what can be done about it. *CIPS (Canadian Information Processing Society) Edmonton '87 Conference Tutorial Notes*, Edmonton, Canada.

[14] Papadimitriou, C. H. (1979). Serializability of concurrent database updates. *Journal of the ACM*,

26(4), 631–653. https://doi.org/10.1145/322154.322158.

[15] Papadimitriou, C. H., Bernstein, P. A., Rothnie, J. B., & Stearns, R. E. (1976). Concurrency controls for database systems. *Proceedings of the 17th Symposium on Foundations of Computer Science (FOCS)*.

[16] Kung, H. T., & Papadimitriou, C. H. (1979). An optimality theory of concurrency control for databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/582095.582114.

[17] Carey, M. J., & Livny, M. (1988). Distributed concurrency control performance: A study of algorithm, distribution, and replication. *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, California.

[18] Chung, Y., & Gruenwald, L. (1994). Research issues for a real-time nested transaction model. *Proceedings of the 2nd IEEE Workshop on Real-Time Applications*, July 1994.

[19] Medjahed, M. O., & Elmagarmid, A. K. (2009). Generalization of ACID properties. In *Encyclopedia of Database Systems*. Boston, MA: Springer. https://doi.org/10.1007/978-0-387-39940-9_736.

[20] Garcia-Molina, H., & Salem, K. (1987). Sagas. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/38714.38742.

[21] Rusinkiewicz, M., & Sheth, A. (1995). Specification and execution of transactional workflows. In W. Kim (Ed.), *Proceedings of the International Conference on Management of Data* (pp. 592–620). New York, NY: ACM Press/Addison-Wesley.

[22] Pu, C. (1988). Superdatabases for composition of heterogeneous databases. *Proceedings of the 4th International Conference on Data Engineering*. https://doi.org/10.1109/ICDE.1988.105502.

[23] Zaslavsky, A., & Bhargava, B. (2009). *Flex transactions*. In Ö. M. Liu, L. (Ed.), Boston, MA: Springer.

[24] Moss, J. E. B. (1981). *Nested transactions: An approach to reliable distributed computing*. Cambridge, MA.

[25] Weikum, G., & Schek, H. (1991). Multi-level transactions and open nested transactions. *Proceedings of the International Workshop on Database Systems for Advanced Applications*.

[26] Bjork, L. A. (1973). Recovery scenario for a DB/DC system. *Proceedings of the ACM Annual Conference*. https://doi.org/10.1145/800192.805695.

[27] Davies, C. T. (1973). Recovery semantics for a DB/DC system. *Proceedings of the ACM Annual Conference*. https://doi.org/10.1145/800192.805694.

[28] Guerraoui, R. (1995). Nested transactions: Reviewing the coherence contract. *Elsevier Science Journal*, 84, 161–172. https://doi.org/10.1016/0020-0255(94)00118-U.

[29] Karabatis, G. (2017). Nested transaction models. In *Encyclopedia of Database Systems*. New York, NY: Springer. https://doi.org/10.1007/978-1-4899-7993-3_716-2.

[30] Buchmann, A. (2016). Open nested transaction models. In *Encyclopedia of Database Systems*. New York, NY: Springer. https://doi.org/10.1007/978-1-4899-7993-3_717-2.

[31] El-Sayed, H. S. H., & Ahmed, M. E. E.-S. A. A. (2001). Effect of shaping characteristics on the performance of nested transactions. *Information and Software Technology*, 43(10), 579–590. https://doi.org/10.1016/S0950-5849(01)00164-1.

[32] Haerder, T., & Rothermel, K. (1993). Concurrency control issues in nested transactions. *VLDB Journal*, 2(1), 39–74. https://doi.org/10.1007/BF01231798.

[33] Bernstein, P. A., & Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), 185–222. https://doi.org/10.1145/356842.356846.

[34] Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency control and recovery in database systems*. Reading, MA: Addison-Wesley.

[35] Thomas, R. H. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*. https://doi.org/10.1145/320071.320076.

[36] Kung, H. T., & Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), 213–226. https://doi.org/10.1145/319566.319567.

[37] Eswaran, K. P., Gray, J. N., Lorie, R. A., & Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), 624–633. https://doi.org/10.1145/360363.360369.

[38] Alsberg, P. A., & Day, J. D. (1976). A principle for resilient sharing of distributed resources. *Proceedings of the 2nd International Conference on Software Engineering*.

[39] Stonebraker, M. (1979). Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software*

Engineering, 5(3), 188–194. https://doi.org/10.1109/TSE.1979.234180.

[40] Lindsay, B., Mohan, C., & Obermarck, R. (1986). Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4), 378–396. https://doi.org/10.1145/7239.7266.

[41] Tandem Computers. (1987). NonStop SQL: A distributed high-performance, high-availability implementation of SQL. *Proceedings of the International Workshop on High-Performance Transaction Systems*.

[42] Tandem Computers. (1988). A benchmark of NonStop SQL on the debit credit transaction. *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[43] Borr, A. (1988). High performance SQL through low-level system integration. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/971701.50244.

[44] Hehner, D. H., & Verjus, J. P. (1979). An algorithm for maintaining the consistency of multiple copies. *Proceedings of the 1st International Conference on Distributed Computing Systems*.

[45] Reed, D. P. (1978). *Naming and synchronization in a decentralized computer system*. Cambridge, MA.

[46] Reed, D. P. (1983). Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1, 3–23. https://doi.org/10.1145/357353.357355.

[47] Batory, D., & Mahbod, B. (1988). Generalized version control in an object-oriented database. *Proceedings of the IEEE 4th International Conference on Data Engineering*, February 1988.

[48] Chou, H. T., & Kim, W. (1986). A unifying framework for versions in a CAD environment. *Proceedings of the International Conference on Very Large Data Bases*, Kyoto, Japan.

[49] Chou, H. T., & Kim, W. (1988). Versions and change notification in an object-oriented database system. *Proceedings of the Design Automation Conference*. https://doi.org/10.1109/DAC.1988.14770.

[50] Kao, B., Lee, K. Y., & Chiu, A. L. (1997). Comparing two-phase locking and optimistic concurrency control protocols in multiprocessor real-time databases. *Proceedings of the 5th International Workshop on Parallel and Distributed Real-Time Systems and the 3rd Workshop on Object-Oriented Real-Time Systems*.

[51] Carey, M. J., & Livny, M. (1987). Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, December 1987.

[52] Ramamritham, K., & Stankovic, J. A. (1985). Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers*, 34(12), 1130–1143. https://doi.org/10.1109/TC.1985.6312211.

[53] Peterson, J. L., & Silberschatz, A. (1985). *Operating system concepts*. Reading, MA: Addison-Wesley Publishing Company.

[54] Stankovic, J. A., & Huang, D. T. J. (1991). On using priority inheritance in real-time databases. *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*.

[55] Davari, S., & Sha, L. (1992). Sources of unbounded priority inversion in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, 26(2), 110–120. https://doi.org/10.1145/142111.142126.

[56] Haque, W. U. (1993). *Transaction processing in real-time database systems*.

[57] Lynch, N. (1986). Concurrency control for resilient nested transactions. *Advances in Computing Research*, 3, 335–376.

[58] Lynch, N., & Merritt, M. (1986). *Introduction to the theory of nested transactions*. Theoretical Computer Science, 62, 123–185. https://doi.org/10.1016/0304-3975(86)90014-9.

[59] Fekete, A., Lynch, N., Merritt, M., & Weihl, W. E. (1987). Nested transactions and read/write locking. *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, San Diego, CA. https://doi.org/10.1145/28659.28669.

[60] Fekete, A., & Lynch, N. (1991). Multi-granularity locking for nested transaction systems. *Proceedings of the MFDBS'91 Conference*.

[61] Fekete, A., Lynch, N., Merritt, M., & Weihl, W. E. (1990). Commutativity-based locking for nested transactions. *Journal of Systems Sciences*, 41, 65–156. https://doi.org/10.1016/0022-0000(90)90034-I.

[62] Madria, S. N., & Kumar, B. C. S. (1997). Formalization and correctness of a concurrency control algorithm for an open and safe nested transaction model using I/O automaton model. *Proceedings of the 8th International Conference on Management of Data (COMAD'97)*, Madras, India.

[63] Rundensteiner, E. A., & Harder, T. (1995). Concurrency control in nested transactions with enhanced lock modes for KBMSs. *Proceedings of the 6th International Conference on Database and Expert Systems Applications (DEXA'95)*, London, UK.

[64] Garcia-Molina, H., & Lynch, N. (1994). Nested transactions and quorum consensus. *ACM Transactions on Database Systems*, 19, 537–585. https://doi.org/10.1145/195664.195666.

[65] Fekete, A., & Kameda, T. (1989). Concurrency control of nested transactions accessing B-trees. *Proceedings of the 8th ACM Symposium on Principles of Database Systems*.

[66] Madria, S. N., & Kumar, B. C. S. (1997). Formalization of linear hash structures using nested transactions and I/O automaton model. *Proceedings of the IADT '98 Conference*, Berlin, Germany.

[67] Abdouli, A. M., & Abdouli, M. A. (2005). A system supporting nested transactions in DRTDBSs. *Proceedings of the 1st International High-Performance Computing Conference*, September 2005. https://doi.org/10.1007/11557654_99.

[68] Abdouli, M. A. (2005). Scheduling distributed real-time nested transactions. *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. IEEE Computer Society. https://doi.org/10.1109/ISORC.2005.49.

[69] Huang, J., & Ramamritham, K. (2000). The prompt real-time commit protocol. *IEEE Transactions on Parallel and Distributed Systems*, 11(2), 160–181. https://doi.org/10.1109/71.841752.

[70] Resende, T. H. A. G., Furtado, J. L. R., & Resende, F. (1997). Detection arcs for deadlock management in nested transactions and their performance. In *Advances in Databases, BNCOD* (Lecture Notes in Computer Science). Berlin, Heidelberg: Springer.

[71] Haque, W. U. (1993). *Transaction processing in real-time database systems*.

[72] Haerder, T., & Reuter, K. R. (1987). Concepts for transaction recovery in nested transactions. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/38714.38741.

[73] Awoyelu, I., & Osinuga, P. (2012). Mathlab implementation of quantum computation in searching an unstructured database. *Informatica (Slovenia)*, 36, 249–254.