

A Review of Locking Protocols for Concurrency Control in Parallel Databases

Donga Dhanushya¹, Manas Kumar Yogi^{2,*}

Abstract

In modern database systems, many transactions may run at the same time. When several users try to access the same data or update the data simultaneously, it creates problems such as inconsistent data, lost updates, or misinformation, and may lead to conflicts between transactions and deadlocks. To avoid these types of issues and maintain the correctness of the database, concurrency control protocols are mainly used. This paper reviews different locking-based concurrency control protocols that are used in parallel database systems. This paper explains how these protocols work and how they help in managing multiple transactions running at the same time. In the beginning, this study describes the main issues in concurrent transaction processing, such as ensuring data consistency and handling deadlocks. After that, it introduces some important locking protocols like Two-Phase Locking (2PL) and its different types. It also involves other advanced protocols like Distributed Two-Phase Locking (D2PL), High Priority Two-Phase Locking (HP2PL), and Two-Phase Locking–Lock Write All (2PL-LW). The review also analyses and compares these protocols based on different factors such as transaction throughput, response time, ability to handle conflicts, and complexity of implementation. The research also discusses different performance evaluation metrics and experimental methods that help researchers understand how effective these protocols are. These methods provide a framework to measure and analyze the performance of various concurrency control techniques. Finally, the paper discusses some future research directions such as decentralized timestamp methods like FairTID, hybrid protocol approaches, and AI-based adaptive concurrency control techniques that can improve the performance of modern parallel and distributed database systems.

Keywords: Locking protocols, two-phase locking, concurrency control, parallel databases, transactions, deadlocks

INTRODUCTION

Parallel database systems have gained significant importance over the last few years, as modern applications generate large amounts of data. In such systems, it is possible to have more than one processor and storage device cooperating and allowing numerous transactions to be run simultaneously [1]. This means that the processing of data is quicker, and it is excellent when it comes to large and complex data. Despite making numerous calls concurrently, hiccups may occur, and transactions may attempt to read or write the same data simultaneously, causing inconsistencies and erroneous outcomes.

To manage this, database management systems use concurrency control locking protocols. These protocols are used to control concurrent sessions of

*Author for Correspondence

Manas Kumar Yogi
E-mail: manas.yogi@gmail.com

¹Undergraduate Student, Department of Computer Science and Engineering, Pragati Engineering College (A), Surampalem, Andhra Pradesh, India

²Assistant Professor, Department of Computer Science and Engineering, Pragati Engineering College (A), Surampalem, Andhra Pradesh, India

Received Date: March 15, 2026

Accepted Date: March 30, 2026

Published Date: April 07, 2026

Citation: Donga Dhanushya, Manas Kumar Yogi. A Review of Locking Protocols for Concurrency Control in Parallel Databases. Journal of Advanced Database Management & Systems. 2026; 13(2): 1–9p.

transactions to prevent possible collisions. The priority is to make the ultimate result correct when the transactions are carried out sequentially. In summary, concurrency control maintains consistency in the database, even though several transactions can be executed simultaneously, thereby increasing performance.

One approach that is used to control concurrency is the locking-based approach. Under such a system, a transaction first obtains a lock on a piece of data, and only after that can it read or write that data. Rules of compatibility of locks determine the type of locks that can be used in combination to make transactions [2]. The primary purpose of these protocols is to achieve as much parallel processing as possible without issues such as dirty reads, lost updates, and unrepeatable reads.

There are also additional challenges in parallel database environments. For example, deadlocks may occur when transactions are held in contention with other transactions. It can also be difficult to manage locks that work across many processors and to coordinate lock requests between the nodes in a network [3]. Therefore, it is important to develop efficient locking protocols to ensure high system performance and data integrity.

This article reviews different locking-based concurrency control protocols in parallel database systems. It describes the manner in which they operate, their relative advantages and disadvantages, and discusses their functioning in various situations. The mechanisms to handle concurrent transactions provided by locking mechanisms and the applications of such protocols on multiprocessor and real-time database systems are also analyzed in this study. Through the analysis of earlier works and current practices, this study provides a better view of locking protocols and identifies potential opportunities for further research on the topic of concurrency control.

Lock-based concurrency control is a simplistic concept in that a transaction needs to obtain a lock on data to read or modify it before it can do so. These locks become control measures that prevent conflicting operations that may occur simultaneously to take place and ensure that the database is precise and consistent. The main concepts in this strategy are the various types of locks, lock compatibility regulations, and the steps that govern the purchase and release of locks.

Basic Lock Types and Compatibility

In a database system, many transactions attempt to use the same data simultaneously. In these situations, locking protocols are used to manage transactions. Locking helps control which transaction can access a data item at a particular moment. It acts as a permission that shows whether the data can be read or changed by other transactions.

There are mainly two types of locks used in database systems: shared locks (S-locks) and exclusive locks (X-locks). A shared lock is used when a transaction only needs to read the data [4]. Since reading does not change the value of the data, several transactions can read the same data at the same time. Because of this situation, multiple shared locks will exist on the same data. When a transaction attempts to change or update the data, it requires an exclusive lock.

This means that while one transaction is modifying the data, no other transaction can read or write that data at the same time. Other transactions must wait until the lock is released. This helps prevent conflicts between transactions and keeps the database more accurate and consistent. This is why both shared locks and exclusive locks are mainly important. These locks control how transactions access the data and ensure that the database remains structured and organized.

Two-Phase Locking (2PL) and Its Variations

Two-Phase Locking (2PL) is the method that is used in database systems to control transactions that run at the same time [4]. It helps to keep the database data consistent and reduces problems when many transactions try to access the same data. In this situation, the transaction occurs in two phases. They are

mainly the growing and shrinking phases. In the growing phase, a transaction requests and obtains locks on the data it requires. During this phase, it cannot release any locks. Subsequently, the transaction enters the shrinking phase. In this phase, the transaction starts releasing the locks it already has, but it cannot request new locks. Using these two phases, the database system controls how transactions use data and avoids conflicts between them. This ensures that the result is correct, as if the transactions were executed one after another. To improve performance and solve problems like deadlocks, different variations of the 2PL protocol are also used in database systems Table 1.

Strict Two-Phase Locking is a modified version of the basic 2PL protocol. In this method, a transaction retains all its locks until the transaction is completed by committing. The main advantage of this approach is that it prevents cascading aborts. Cascading aborts occur when the failure of one transaction causes other dependent transactions to roll back. Owing to this property, strict 2PL is commonly used in systems where many transactions are executed simultaneously. It helps to maintain consistency and avoids the chain reactions of rollbacks. As all the locks are in place until the completion of a transaction, other transactions must wait to access the same data.

Rigorous Two-Phase Locking is a stricter version of the Two-Phase Locking protocol. In this method, the shared and exclusive locks are held together until the transaction is completed. Owing to these situations, the database can maintain a high level of consistency with the data. The main advantage of this approach is that it enables an accurate and rapid recovery process. Because the locks are released only after the transaction ends, the database remains in a controlled and stable state. Lastly, one drawback is that concurrency is reduced because other transactions must wait until the locks are released. This is a major drawback if any transaction takes a long time to complete.

Conservative Two-Phase Locking works differently from other types of 2PL. In this method, a transaction must request all the locks it requires before it begins execution. The system checks and grants these locks beforehand. The main advantage of this approach is that it avoids deadlocks because the transaction does not request any new locks after it begins running. For one reason, this method is not always convenient because the transaction must know all the data items it will use before execution. In some situations, locks can also be held even when they are not needed immediately, which can reduce the level of concurrency in the database system.

Distributed Two-Phase Locking (D2PL)

When Two-Phase Locking is applied to distributed or parallel database systems, the process is more complicated because the information is stored in more than one node. Each node can handle various sections of the database, so they must have proper coordination to control the transactions of the entire system. A hierarchical lock management system is common in distributed environments.

Table 1. Comparison of two-phase locking protocol variants [5–7].

Protocol variant	Lock release timing	Commit dependencies	Concurrency level	Implementation complexity
Basic Two-Phase Locking (2PL)	Locks are released during the shrinking phase before the transaction commits.	Transactions are independent	Medium	Simple
Strict 2PL	All write locks are released only after commit or abort	Other transactions must wait for commit	Low to Medium	Moderate
Rigorous 2PL	Both read and write locks are released only after commit	Transactions strongly depend on commit	Low	Complex
Conservative (Static) 2PL	All locks are obtained before the transaction starts and released later	Transactions wait until all locks are available	Medium	Moderate

This system has Global Lock Managers (GLM), which coordinate the locks with the transactions in the various nodes, and Partition Lock Managers (PLM), which coordinate in-depth locks within separate data partitions [8]. This structure enables high consistency of the distributed transactions, and more than one operation can go on simultaneously through a very careful way of lock levels.

Distributed two-phase locking (D2PL) performance studies in large cluster systems have demonstrated that the performance of parallel systems can be significantly better than that of single-node systems. Experiments on a 100-node cluster indicated that the system had an estimated throughput higher than the 2PL system on a single node. However, the latency increased by approximately 47% because of the extra communication and coordination required among the nodes. These findings highlight the trade-off of distributed locking systems, in which the benefits of parallel processing must be weighed against the additional communication and synchronization overhead required to coordinate locks among multi-nodes.

DEADLOCK MANAGEMENT AND PROTOCOL EXTENSIONS

Deadlock is a common problem that occurs in database systems; to control deadlocks, locking protocols are mainly used in transactions. Deadlocks mainly occur when two or more transactions wait for each other to release the locks on some data with the same requirements. This leads to creating a circular waiting situation, which is known as the present transaction waiting for another transaction to complete execution; it is like they both wait for each other's transactions to finish. For this reason, no transaction can be finished at the same time; they are locked in a circular way of situation. When a deadlock occurs, the transactions stop making progress, and the system performance becomes slower. Therefore, it is very important for database systems to handle all deadlocks properly using locking protocols so that transactions do not have to wait longer. To solve these problems, database systems use methods such as deadlock detection, prevention, and resolution. Some improved protocols are also used to increase system performance, mainly in parallel and distributed database systems [9].

Deadlock Detection Mechanisms

Deadlock detection is a process used in database systems to identify circular waiting situations between transactions and solve them using different methods. The most common method used for detecting deadlocks is the wait-for graph. A wait-for graph is a directed graph in which each node represents a transaction in the database system. If one transaction is waiting for a lock that is already held by another transaction, an edge is drawn between the transactions in the graph. When a cycle is formed in this graph, it indicates that a deadlock has occurred in the database, and the system should take some action to solve that issue. In centralized database systems, detecting deadlocks is much easier because all transaction information is available in one place. The system can easily create a wait-for graph and instantly check whether any cycle is present. However, in distributed database systems, deadlock detection becomes more difficult because transactions can run on different nodes, and the waiting information should be collected from many different locations. To detect deadlocks in distributed environments, different methods are used in the systems. In the centralized approach, all wait-for information from different nodes is given to a single coordinator node that checks the graph for cycles [10]. In the hierarchical approach, deadlock detection is done at different levels of the system. There is also a distributed approach, where multiple nodes work together at the same time to detect deadlocks without depending on a central coordinator. Each of these methods has some advantages and some drawbacks, especially in terms of communication cost, detection speed, and reliability. Therefore, many modern distributed database systems use hybrid approaches that combine local detection with global coordination to identify deadlocks more efficiently.

Deadlock Prevention Protocols

Deadlock prevention protocols are used in database systems to stop deadlocks before they occur in any transactions. These protocols mainly control the way transactions request locks on the data items. Instead of detecting and solving deadlocks, these protocols make sure that circular waiting situations

will not be created in the system. Two commonly used timestamp-based prevention methods are the wait-die and wound-wait protocols. In these methods, each transaction is assigned a unique timestamp when it starts its execution. Older transactions will have smaller timestamps, and newer transactions will have larger timestamps. In the wait-die protocol, if a younger transaction requests a lock that is held by an older transaction, the younger transaction is allowed to wait. However, if an older transaction requests a lock that is held by a younger transaction, the younger transaction is aborted and restarted later with the same timestamp. This method helps prevent circular waiting between transactions. In the wound-wait protocol, the logic is slightly different. If an older transaction requests a new lock held by a transaction, the older transaction forces the younger transaction to abort so that it can obtain the lock. However, if a younger transaction requests a lock held by an older transaction, the younger transaction must wait until the lock is released. Both wait-die and wound-wait protocols help prevent deadlocks in the database system, as shown in Table 2.

High Priority Two-Phase Locking (HP2PL)

In real-time database systems, transactions must be completed within a fixed time limit. Because of this timing requirement, the normal deadlock handling methods cannot always work properly. To manage this situation, a method called high-priority two-phase locking (HP2PL) is used. This locking protocol mainly considers the priority of transactions when any lock conflicts occur. In HP2PL, every transaction is assigned a priority level by the system. When two transactions request the same lock simultaneously, the system compares their priorities. Based on priorities, a transaction with a high priority will get the lock first, and a transaction with a low priority will have to wait until the lock is released. This method helps the system to complete more important transactions faster based on their given priorities in real-time.

Research studies comparing HP2PL with optimistic concurrency control techniques in parallel database systems have shown that HP2PL performs better when the number of incoming transactions is moderate or high [11]. Because higher-priority transactions are given preference, the chances of completing important transactions within their deadlines increase. However, this approach can cause lower-priority transactions to be restarted more frequently. Although this increases the number of restarts, it is acceptable in real-time systems because completing important transactions on time is more important than delaying or restarting less important transactions.

Two-Phase Locking-Lock Write All (2PL-LW)

Studies comparing lock-based and optimistic concurrency control protocols in multiprocessor real-time databases have shown that both approaches have certain limitations when applied to parallel systems. Optimistic concurrency control methods usually perform well in disk-based systems; however, their performance decreases when there is high contention because transactions may need to be restarted multiple times. However, traditional locking protocols can cause excessive blocking and may also lead to priority inversion, where high-priority transactions are forced to wait for lower-priority ones.

Table 2. Deadlock prevention protocol characteristics [12–14].

Protocol	Transaction Behavior Pattern	Restart Characteristics	Starvation Risk
Wait-die	Older transaction waits, younger transaction aborts	Aborted younger transactions restart with the same timestamp	Low
Wound wait	Older transaction aborts younger transaction.	Aborted younger transactions restart later.	Very Low
Timeout-based	Transaction waits for a limited time for a lock	Transaction restarts after timeout	Medium
No-WAIT	Transaction aborts immediately if lock is unavailable	Frequent restarts may occur	High

To overcome these problems, a hybrid protocol called Two-Phase Locking – Lock Write All (2PL-LW) was introduced. This protocol modifies the traditional locking approach by requiring transactions to acquire write locks for all data items that might be updated. Simultaneously, read locks continue to follow the usual rules of the Two-Phase Locking (2PL) protocol [15]. This approach helps reduce conflicts between transactions that attempt to update the same data. Simultaneously, it allows many transactions to read the data simultaneously. Therefore, the method works well in memory-based multiprocessor systems, where data can be accessed very quickly, but conflicts between transactions may still occur.

PERFORMANCE ANALYSIS AND COMPARATIVE EVALUATION

It is important to understand how the locking protocols behave under different conditions when selecting the most suitable protocol for a database system. This helps the system to work more efficiently and manage transactions accurately. Performance analysis helps the researchers and developers to evaluate how efficiently a protocol handles all the transactions in different situations. This section mainly shows the results from experimental studies and compares them with different concurrency control protocols based on important performance factors [16].

Throughput and Scalability

Throughput is one of the main performance metrics used to evaluate parallel database systems. It is usually measured as the number of transactions processed per second (TPS). Studies comparing Distributed Two-Phase Locking, Distributed Optimistic Concurrency Control, and Distributed Multi-Version Concurrency Control show that their performance can vary depending on the type of workload. In workloads where most operations are read operations and there is low data contention, optimistic concurrency control protocols generally perform better than locking-based ones. Because optimistic protocols do not require locks for most operations, they avoid the overhead of acquiring and releasing locks, which can lead to approximately 41% higher throughput in such cases.

However, when the workload includes many write operations and there is a high level of locking based protocols usually perform better. In concurrency control methods, transactions must restart many times when any issues happen. These repeated restarts can suddenly reduce the overall system throughput. In some situations, the system throughput can decrease by 63% [17]. To solve this problem, a method Multi-Version Concurrency Control (MVCC) is used, which combines the advantages of the different approaches. In this method, multiple versions of the same data items are stored in the system. For this reason, read operations can access older versions of the data without waiting for the write operations to be completed. This improves reading performance, while writing operations are managed using timestamp ordering.

However, maintaining multiple versions of the data also creates extra work for the system. Old versions must be removed regularly to meet their requirements. This process needs at least 15% extra work for the additional requirements Table 3.

Table 3. Performance comparison of major concurrency control protocols [18, 19].

Protocol	Throughput	Latency	Conflict handling capability	Optimal workload type
Two-Phase Locking (2PL)	Moderate	Moderate to High	Handles conflicts using locks but may cause blocking	Workloads with moderate contention and balanced read/write operations
Optimistic Concurrency Control (OCC)	High when contention is low	Low	Detects conflicts at commit time, may cause transaction restarts	Read-heavy workloads with low contention
Multi-Version Concurrency Control (MVCC)	High	Low to Moderate	Allows multiple versions to reduce read-write conflicts	Read-intensive workloads with mixed operations
Timestamp Ordering	Moderate	Low	Resolves conflicts based on transaction timestamps	Systems requiring strict ordering and consistency

Scalability Characteristics

Scalability analysis is used to study the performance of different concurrency control protocols when system resources, such as the number of nodes, increase. In Distributed Two-Phase Locking (D2PL), the system performance increases almost linearly when the number of nodes grows up to about 50. Subsequently, the coordination required for managing locks between different nodes becomes more complex. This extra coordination creates a workload and reduces the performance improvement gained from adding more nodes. In addition, depending on the global lock, managers can limit the scalability of the system.

Optimistic Concurrency Control (OCC) protocols usually scale better in situations where most operations are read operations, and there is very little data contention. Because these protocols require less coordination between nodes, they can scale efficiently in such situations [17]. However, when contention becomes high, the number of transaction restarts increases rapidly as the number of nodes increases. This can significantly reduce the system's performance, making OCC suitable mainly for workloads with predictable and low contention levels.

Multi-Version Concurrency Control (MVCC) protocols generally provide more stable scalability across different workload conditions. These protocols allow read operations to access different data versions without blocking write operations. As a result, they reduce the need for coordination between transactions. Studies show that MVCC systems can maintain good performance even when the system grows to more than 100 nodes [18]. Although managing multiple versions introduces additional overhead, this overhead increases gradually with the number of transactions, allowing the system to maintain consistent performance as it scales.

Real-Time Performance Considerations

Real-time database systems have an additional requirement that transactions must be completed within a specified time limit or a deadline. Therefore, the overall system throughput is not always the most important factor. Instead of this, the main goal is to make sure that there are more transactions as possible and are finished before their deadlines. This is usually measured using the deadline satisfaction ratio.

Studies comparing high-priority two-phase locking (HP2PL), optimistic timestamp-based protocols, and validation-based concurrency control methods have shown that no single protocol performs best in every situation. When the transaction arrival rate is low, optimistic protocols usually achieve higher deadline satisfaction because they have less overhead and transactions can be executed quickly. However, when the arrival rate becomes high, optimistic protocols may suffer from frequent transaction restarts owing to conflicts [19]. These restarts can cause several transactions to miss their deadlines. In such situations, locking-based protocols tend to perform more reliably because they control conflicts more strictly than other protocols.

This behavior suggests that hybrid approaches, which can switch between different protocols depending on the system load, may provide better performance in real-time systems. Another approach used in parallel real-time databases is the validation concurrency control protocol. In this method, the transactions are validated before entering their critical sections to detect conflicts quickly. If a transaction is not ready to be completed before its deadline, it can be aborted quickly, allowing the system to focus its resources on finishing the transactions that still have a chance to meet their deadlines [20].

CONCLUSION AND FUTURE DIRECTIONS

Various locking mechanisms applicable to parallel database systems in concurrency control are discussed in this review. The two-phase locking (2PL) protocol and its variants offer high consistency and efficient management of transactions, which is why they are highly employed in various database

applications. However, with the expansion of distributed and parallel database architectures, standard locking techniques pose problems, including increased overhead of coordination and latency. These issues have prompted scholars to come up with better and composite protocols. It was revealed that there is no single locking protocol suitable for every situation. High-priority two-phase locking enhances the performance of real-time systems because it prioritizes critical transactions that must be completed in time. Similarly, the Two-Phase Locking-Lock Write All protocol is a hybrid protocol that enhances performance in multiprocessor systems that are characterized by high transaction contention.

Finally, integrating blockchain technology with database concurrency control opens new research directions and presents some difficulties. In such environments, protocols may use optimistic style execution together with deterministic validation when transactions occur between different organizations that may not fully trust each other. These protocols must still maintain strong consistency and support Byzantine fault tolerance. Some early studies have shown that this approach can achieve good throughput while preserving trust between different parties.

REFERENCES

1. Shams MY, Abolaban AS, Abohany AA. A review on concurrency control techniques in database management systems. *Kafrelsheikh J Inf Sci.* 2022;3(1).
2. Cellary W, Morzy T, Gelenbe E. *Concurrency Control in Distributed Database Systems.* Amsterdam: Elsevier; 1988.
3. Nguyen T, Kawashima H. Fairly decentralizing a hybrid concurrency control protocol for real-time database systems. *Concurr Comput Pract Exp.* 2025;37(4-5):e70018. doi:10.1002/cpe.70018.
4. Chiu A, Kao B, Lam KY. An analysis of lock-based and optimistic concurrency control protocols in multiprocessor real-time databases. *J Syst Softw.* 1998;42(3):273-286. doi:10.1016/S0164-1212(98)10015-8.
5. Vadapalli P. Lock based protocol in DBMS: types, working, and real-life examples [Internet]. upGrad; 2024 Jul 30. Available from: <https://www.upgrad.com/tutorials/software-engineering/dbms-tutorial/lock-based-protocol-in-dbms/>
6. Lei XD, Yuan XL. Validation concurrency control protocol in parallel real-time database systems. *J Cent South Univ Technol.* 2002;9(3):197-201. doi:10.1007/s11771-002-0026-y.
7. Agrawal R, Carey MJ, Livny M. Models for studying concurrency control performance: alternatives and implications. *ACM SIGMOD Rec.* 1985;14(4):108-121. doi:10.1145/971699.318909.
8. Bernstein PA, Goodman N. Concurrency control in distributed database systems. *ACM Comput Surv.* 1981;13(2):185-221. doi:10.1145/356842.356846.
9. Bernstein PA, Hadzilacos V, Goodman N. *Concurrency Control and Recovery in Database Systems.* Reading (MA): Addison-Wesley; 1987.
10. Eswaran KP, Gray JN, Lorie RA, Traiger IL. The notions of consistency and predicate locks in a database system. *Commun ACM.* 1976;19(11):624-633. doi:10.1145/360363.360369.
11. Bernstein PA, Goodman N. Concurrency control in distributed database systems. *ACM Comput Surv.* 1981;13(2):185-221. doi:10.1145/356842.356846.
12. Hsiao DK. Data Base Computers. In: *Advances in Computers.* Vol. 19. New York (NY): Academic Press; 1980. p. 1-64. doi:10.1016/S0065-2458(08)60032-7.
13. Haritsa JR, Carey MJ, Livny M. On being optimistic about real-time constraints. In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems;* 1990 Apr 2-4; Nashville, TN. New York (NY): ACM; 1990. p. 331-343. doi:10.1145/298514.298585.
14. Haritsa JR, Carey MJ, Livny M. Dynamic real-time optimistic concurrency control. In: *Proceedings of the 11th IEEE Real-Time Systems Symposium;* 1990 Dec; Lake Buena Vista, FL. Los Alamitos (CA): IEEE Computer Society; 1990. p. 94-103.
15. Kanellakis PC, Papadimitriou CH. The complexity of distributed concurrency control. In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science;* 1981; Nashville, TN. Washington (DC): IEEE Computer Society; 1981. p. 185-197. doi:10.1109/SFCS.1981.45.

16. Silberschatz A, Korth HF, Sudarshan S. Database System Concepts. 4th ed. New York (NY): McGraw-Hill; 2002.
17. Silberschatz A. A case for non-two-phase locking protocols that ensure atomicity. IEEE Trans Softw Eng. 1983;9(4):535-538. doi:10.1109/TSE.1983.235107.
18. Hellerstein JM, Stonebraker M, editors. Readings in Database Systems. 4th ed. Cambridge (MA): MIT Press; 2005.
19. Wolfson O, Yannakakis M. Deadlock-freedom (and safety) of transactions in a distributed database. In: Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems; 1985 Mar; Portland, OR. New York (NY): Association for Computing Machinery; 1985. p. 105-112. doi:10.1145/325405.325418.
20. Wolfson O. The overhead of locking (and commit) protocols in distributed databases. ACM Trans Database Syst. 1987;12(3):453-471. doi:10.1145/27629.28053.