BAREKENG: Journal of Mathematics and Its Applications
September 2024 Volume 18 Issue 3 Page 1899–1908
P-ISSN: 1978-7227 E-ISSN: 2615-3017
https://doi.org/10.30598/barekengvol18iss3pp1899-1908

EXTENDED SERIAL GRAPH-VALIDATION QUEUE SCHEME WITH LOCKING STRATEGY

Muhammad Fakhri Jauhari [1*], Fahren Bukhari[2], Sri Nurdiati[3]

[1,2,3]*Department of Mathematics, Faculty of Mathematics and Natural Sciences, IPB University*
Jl. Kampus Meranti, Babakan, Dramaga, Bogor, 16680, Indonesia

*Corresponding author's e-mail: ** ***fachrijrm@apps.ipb.ac.id***

ABSTRACT

Article History:

Received: 28th February 2024
Revised: 17th April 2024
Accepted: 11th July 2024
Published: 1st September 2024

Keywords:

Client-Server;
Concurrency Control;
Graph;
Locking.

In today's digital landscape, collaborative work in real-time is on the rise, allowing individuals to connect across different locations through applications facilitated by client-server architecture, enabling users to access and work on the same project simultaneously. However, clients' simultaneous access and modifications to the database can result in data inconsistencies, underscoring the importance of concurrency control. Managing concurrent transactions can introduce complexities and potentially adversely impact server performance. Object caching emerges as a viable solution as an alternative approach to handling transaction traffic. Extended Serial Graph-Validation Queue (Extended SG-VQ) is a control concurrency scheme that operates within the client-server architecture framework and incorporates object caching. The cache component implements a queue-based validation algorithm as part of its validation process. At the same time, the server-side employs a graph-based validation algorithm with locking strategies. Through a series of hypothetical transaction scenarios across three cases, this study validates the effectiveness of the Extended SG-VQ, demonstrating its ability to utilize serial graphs, resolve conflicts, and identify cyclic patterns.

How to cite this article:

M. F. Jauhari, F. Bukhari and S. Nurdiati., "EXTENDED SERIAL GRAPH-VALIDATION QUEUE SCHEME WITH LOCKING STRATEGY," *BAREKENG: J. Math. & App.*, vol. 18, iss. 3, pp. 1899-1908, September, 2024.

1. INTRODUCTION

In recent times, there has been an increasing need for applications that enhance synchronous collaboration, enabling multiple individuals to engage in shared projects concurrently, even when situated in separate locations [1]. This form of collaboration is known as real-time collaboration. It facilitates joint efforts on various tasks, such as co-authoring a document or utilizing an online platform for interactive discussions. Real-time collaboration transcends geographical barriers and simplifies collaboration. For instance, desktop sharing exemplifies this concept by enabling users to present their screens, facilitating simultaneous viewing of shared content among all participants for collaborative interactions [2]. Similarly, document sharing offers a collaborative platform where multiple individuals can access and collectively modify shared files [3]. The client-server model is the most suitable for projects supporting real-time collaboration [4].

The simultaneous processing of multiple transactions (concurrency) risks generating inconsistent data. Concurrent access to a shared database by multiple clients can lead to data inconsistency if one or more clients modify the data [5]. Hence, there arises a need for a mechanism to manage concurrent transactions effectively and mitigate the risk of inconsistencies, known as concurrency control. However, the execution of concurrent transactions can complicate the transaction completion process (commit) and impose an additional workload on the server, potentially decreasing its performance. Consequently, significant research efforts have been dedicated to enhancing server performance, such as exploring object caching strategies on the server side [6] and permitting transactions to access previous versions of objects [7].

One of the schemes developed to address concurrency control issues in client-server system environments is the Validation Queue (VQ) scheme, researched by Bukhari and Shrivastava [8]. This optimistic concurrency control scheme utilizes object caching on the client side. In optimistic concurrency control, transactions will be created of objects from the database server. Afterward, the server conducts a validation process to verify that other transactions have not changed the object. The transaction proceeds if the validation process is successful, and a copy of the modified object will be sent to the database server. Instead, the transaction is aborted if the validation process fails.

As the number of client-side transactions grows, the complexity of the VQ scheme escalates during the commit phase. Hence, this study suggests the Extended Serial Graph-Validation Queue (Extended SG-VQ) scheme as a potential advancement over the VQ scheme. This research aims to modify the Validation Queue (VQ) scheme, specifically on the server side, thereby introducing an alternative approach to manage the concurrent access of resources in client-server systems.

2. RESEARCH METHODS

The development of this scheme began with research on the Validation Queue (VQ) scheme by Bukhari and Shrivastava [8] based on the Read Order Concurrency Control (ROCC) scheme by Shi and Perrizo [9]. The Extended Serial Graph-Validation Queue scheme is a modification of the VQ scheme. Below are the assumptions used in the development of this scheme:

1. Single server system;
2. Clients issue transactions one by one;
3. No blind writes. Transactions must read object $\boldsymbol{x}$ before updating object $\boldsymbol{x}$;
4. Transactions work on their memory. When requesting access to an object, the object is first copied to its memory. Subsequently, the commit request and its updates are sent when the execution ends;
5. No network partition. A message is always sent to its destination, assuming it is received and processed by the client in the same order as sent from the server while maintaining message numbering and order.

2.1 System Architecture

This scheme employs a client-server architecture with object caching implemented on the client side, as shown in **Figure 1**. Therefore, the scheme is divided into the cache and server sides. The Validation Queue (VQ) scheme will encounter complexities in the commit process as the number of transactions from the client-side increases. Therefore, modifications are made to the Validation Queue scheme to simplify the commit process on the server side. The VQ scheme utilizes a queue-based validation algorithm both on the cache and server sides. The cache-side validation algorithm is called the Cache Validation Queue (CVQ). In contrast, the server-side validation algorithm is called the Server Validation Queue (SVQ). The modified scheme is called Extended Serial Graph-Validation Queue (Extended SG-VQ). The modifications only involve changing the validation algorithm on the server side, while the validation algorithm on the cache side continues to use the CVQ validation algorithm. After modification, the validation algorithms on the cache and server sides are sequentially referred to as Validation Queue (VQ) and Serial Graph (SG).

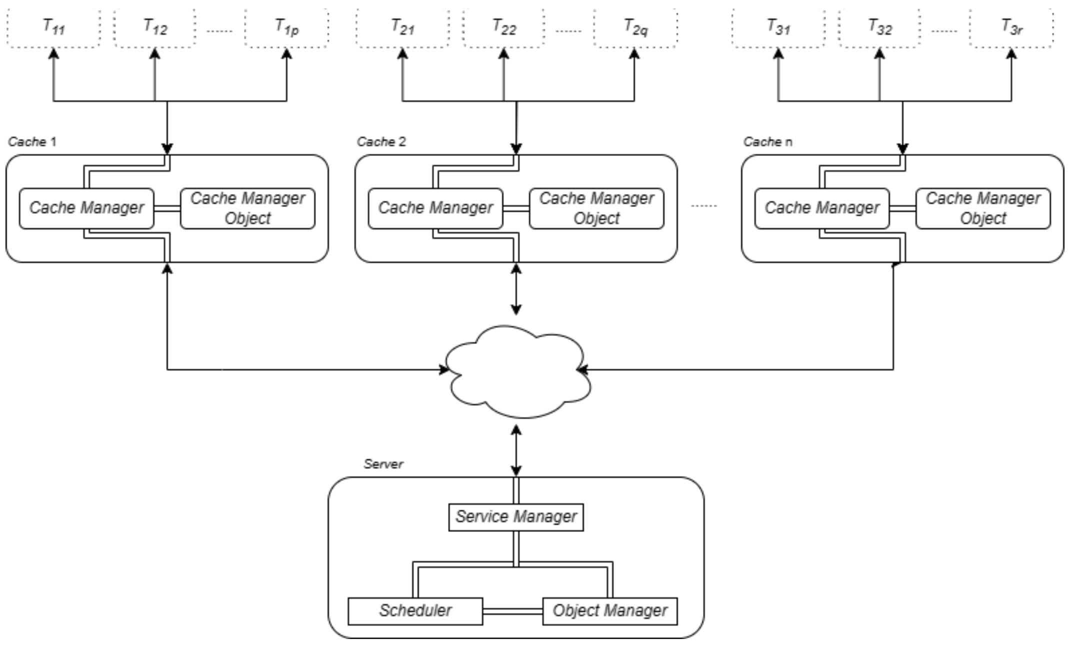


Figure 1. System architecture of the extended SG-VQ scheme

The client-side consists of the application, Cache Manager, and Cache Manager Object. These components are independent modules that communicate explicitly with each other through messages. In this research, the Cache Manager and the Cache Manager Object refer to a cache (local cache). The server side comprises the Service Manager, Scheduler, and Object Manager. The Service Manager manages incoming and outgoing messages on the server. Each request for database access is directed to the Service Manager, who forwards it to the Scheduler. The Scheduler's responsibility is to coordinate database access synchronization. To execute these accesses, the Scheduler sends them to the Object Manager.

It is necessary to understand the concept of elements and transactions. In this research, elements and transactions are defined as the following,

Definition 1. [10] *An element $\boldsymbol{e_{pq}}$ is the $\boldsymbol{q}$-th element of transaction $\boldsymbol{T_p}$, where:*

1. $\boldsymbol{e_{pq}} \subseteq \{\boldsymbol{r_{pq}(x), w_{pq}(x)} | \boldsymbol{x} \coloneqq object\}$
2. $\boldsymbol{Rset(e_{pq})} \cap \boldsymbol{Wset(e_{pq})} = \emptyset$, $\boldsymbol{Rset} \coloneqq readset$, $\boldsymbol{Wset} \coloneqq writeset$

Several requests the system receives can be regarded as a collection of accesses. Without loss of generality, the collection of accesses in the study is considered as elements. The q-th element of transaction T_p is denoted as element e_{pq}. According to **Definition 1** (1), the type of operation within an element can be a read operation and/or a write operation, with x being the object. **Definition 1** (2) explains that read and write operations on the same object cannot be in the same element. The readset is a set of objects to be read, and the writeset is a set of objects to be written/updated.

Definition 2. [10] *Transaction $\boldsymbol{T_p}$ is partial order with ordering relation $<_{\boldsymbol{p}}$, where:*

1. $\boldsymbol{T_p} = \{\boldsymbol{e_{p1}}, \boldsymbol{e_{p2}}, \dots, \boldsymbol{e_{p3}}\} \cup \{\boldsymbol{a_p}, \boldsymbol{c_p} | \boldsymbol{a_p} := abort, \boldsymbol{c_p} := commit\}$;
2. $\boldsymbol{a_p} \in \boldsymbol{T_p}$ *only if* $\boldsymbol{c_p} \notin \boldsymbol{T_p}$;
3. *if* $\boldsymbol{t}$ *is* $\boldsymbol{c_p}$ *or* $\boldsymbol{a_p}$, *for any* $\boldsymbol{e_{pq}}$ *in* $\boldsymbol{T_p}$, *then* $\boldsymbol{e_{pq}} <_p \boldsymbol{t}$;
4. *if* $\boldsymbol{r_{pq}}(\boldsymbol{x}) \in \boldsymbol{e_{pq}}$ *and* $\boldsymbol{w_{pr}}(\boldsymbol{x}) \in \boldsymbol{e_{pr}}$, *then* $\boldsymbol{e_{pq}} <_p \boldsymbol{e_{pr}}$.

Definition 2 (1) indicates that the transaction contains elements and abort or commit operations. Definition 2 (2) explains that if a transaction aborts, it will not commit, and vice versa. Definition 2 (3) states that if operation t is aborted or commited, the ordering relationship includes all elements preceding operation t in the transaction execution. Definition 2 (4) explains that if read and write are operations executed on the same object, the ordering relationship defines the sequence of execution of corresponding elements.

2.2 Graph

As previously explained, the scheme modification focuses on the server-side validation algorithm, which initially used a queue but now utilizes a graph. The server-side validation algorithm is based on a directed graph, also known as a digraph. A directed graph or digraph has a definition similar to an undirected graph, but its edges have directions from one vertex to another, meaning $(v_1, v_2) \neq (v_2, v_1)$ [11]. A path in a digraph $G = (V, E)$ is a sequence $v_1, v_2, \dots, v_k$ with $[v_i, v_{i+1}] \in E \; for \; 1 \leq i < k$ [12]. This path originates from v_1 and goes to v_k. In a digraph, if there is a cycle such that the first and last vertices are the same, this condition is called cyclic [13]. A directed acyclic graph is a digraph that does not contain cycles.

Based on graph theory, the following definition can be formulated for this research:

Definition 3. *Vertex is a collection of non-empty sets* $\boldsymbol{V} \neq \{\}$ *containing readset and/or writeset in a transaction, such that* $\boldsymbol{V} = \{\boldsymbol{T_{i,j}} \mid \boldsymbol{T_{i,j}} = \{\boldsymbol{Rset_{i,j}}, \boldsymbol{Wset_{i,j}}\}$.

Vertex $V(T)$ is represented in the form of nodes as illustrated in Figure 2 below,

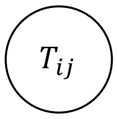


Figure 2. Vertex $\boldsymbol{V}(\boldsymbol{T})$

Definition 4. *Edge is a connector between two vertices indicating the sequence of transaction execution.*

Edge E is represented in the form of directed lines as shown in Figure 3 below,

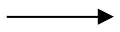


Figure 3. Edge $\boldsymbol{E}$

Definition 5. *A serial graph is a collection of sets of vertices and edges that have been inserted and represent a queue of transactions, thus ensuring serial execution.*

A simple serial graph can be depicted as Figure 4 below,

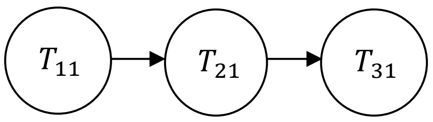


Figure 4. Example of serial graph

Figure 4 illustrates the sequence of transactions occurring as follows: transaction $T_{11} \to T_{21} \to T_{31}$, indicating that T_{11} precedes T_{21} and T_{31}, and subsequently, T_{21} precedes T_{31}.

2.3 Lock-Based Protocols

The lock mechanism facilitates concurrent access to a data item. Access to a data item is only permitted if a lock is currently held on that item. Data items can be locked in two modes, either exclusive (X) mode or shared mode (S) [14]. An exclusive-mode lock is assigned for transactions requiring reading and writing access to the data item. Conversely, a shared-mode lock is assigned for transactions that only need to read from an item but not write to it. Transaction execution can only proceed after the requested lock is obtained [15]. A transaction may be granted a lock on an item if the requested lock is compatible with the locks already held on the item by other transactions. Multiple transactions can hold shared locks (S) on an item simultaneously. However, if any transaction holds an exclusive lock (X) on the item, no other transaction may hold any lock. In such a scenario, a lock cannot be granted, and the requesting transaction must wait until all incompatible locks held by other transactions are released before the lock can be granted [16].

3. RESULTS AND DISCUSSION

3.1 Cache-Side Validation Algorithm

In the Extended SG-VQ scheme, the cache-side algorithm is referred to as the Validation Queue (VQ). VQ serves as a tool for recording execution requests of elements. VQ consists of Read element, Commit element, Validated element, Local Validated element, and Update Propagation element. Each Update Propagation element represents the execution of an update transaction, containing $Rset$ and $Wset$ information within the element. When the Local Manager receives an Update Propagation message from the server, the Read or Commit element is added to VQ, enabling the Local Manager to receive Read or Commit requests from the local transaction.

Each transaction will be executed by sending a commit request to the Local Cache Manager. Upon receiving the commit request, the Local Cache Manager will create a Commit element displayed in the VQ, and then the transaction will be validated. If validation is successful and the transaction is read-only, all transaction elements are merged into a Validated element. If the transaction is an update transaction, all transaction elements are merged into a Local Validated element, after which the Local Cache Manager sends a commit request message to the server. The Local Validated elements will become Validated if the server response is positive; otherwise, the local elements will be discarded.

To validate a transaction, the Local Cache Manager utilizes a validation algorithm. Two elements, namely element e_{ip} from transaction T_i and element e_{jq} from transaction T_j, where $i \neq j$, are said to conflict if they satisfy at least one of the following conditions [10]:

1. $Wset(e_{ip}) \cap Wset(e_{jq}) \neq \emptyset$;
2. $Wset(e_{ip}) \cap Rset(e_{jq}) \neq \emptyset$;
3. $Rset(e_{ip}) \cap Wset(e_{jq}) \neq \emptyset$.

Observe the following queue structure as shown on Figure 5,

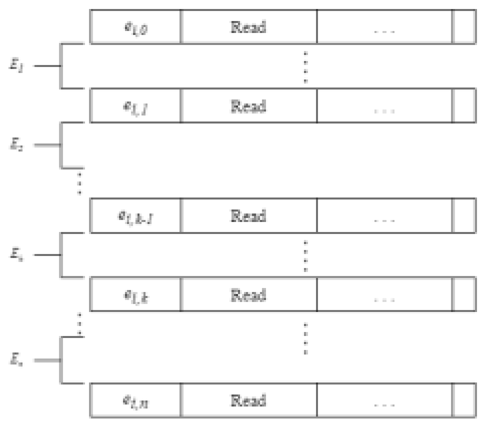


Figure 5. Queue structure

E_1 is a collection of elements from other transactions between the elements $e_{i,0}$ and $e_{i,1}$. Based on Figure 5, E_1 may be empty. Furthermore, E_2 is a collection of elements from other transactions between the elements $e_{i,1}$ and $e_{i,2}$, and E_j between the elements $e_{i,j-1}$ and $e_{i,j}$ of transaction T_i where $1 \leq j \leq n$. E_k is divided into two parts, M and N, by the element e^*, such that $E_k = M; e^*;\ N$, where M and/or N can be empty sequences. Two conditions apply during validation:

1. Condition 1: An element or the combined elements $e_{i,0} \cup e_{i,1} \cup \ldots \cup e_{i,j}$ does not conflict with any element in the sequence E_{j+1}, for every $j = 0, 1, \ldots, n-1$.
2. Condition 2: The combination of elements $e_{i,0} \cup e_{i,1} \cup \ldots \cup e_{i,j}$ do not conflict with any element in the sequence E_{j+1}, for every $j = 0{,}1, \ldots k-1$, and every element in M but the combined elements conflict with the element e^*, for $k = 1, 2, \ldots, n$. Then, the element $e_{i,n}$ or the combined elements $e_{i,0} \cup e_{i,1} \cup \ldots \cup e_{i,j}$ do not conflict with any element in the sequence E_j, for every $j = n, n-1, \ldots k+1$, and the combined elements $e_{i,n} \cup e_{i,n-1} \cup \ldots \cup e_{i,k-1} \cup e_{i,k}$ do not conflict with the element e^* and every element in N.

In the Extended SG-VQ scheme, the cache-side validation algorithm is as follows:

1. The validation process is considered successful in a read-only transaction successful if the transaction satisfies either condition 1 or condition 2.
2. The validation process is considered successful in an update transaction if the transaction satisfies only condition 1.

3.2 Server-Side Validation Algorithm

Before performing the validation process, the server will receive a commit request message from the cache manager. Each object has a unique sequential number called a cache version. The cache version is constantly updated whenever an object is updated. When the server receives the commit request message, it first checks the cache version carried by the transaction. Suppose the cache version carried matches the latest version stored by the server. In that case, the process will proceed to the validation stage.

The server maintains a list containing the objects and the transactions holding locks on each object. In this scheme, the object status is divided into LOCK and UNLOCK. If the object status is LOCK, a transaction locks it, and other transactions cannot access it to update the object. If the object status is UNLOCK, any transaction does not currently lock the object and can be accessed by anyone. The validation begins with checking the object status to prevent multiple transactions from updating the same object simultaneously. Suppose the object status is LOCK, and another transaction attempts to update the same object. In that case, the last arriving transaction will be returned to its originated cache.

The next step is to insert transactions into the serial graph. The process of inserting a transaction into the serial graph is adjusted based on the following conditions:

1. If $Rset_{ij} \cap Wset_{kl} \neq \{\}$, then transaction T_{ij} is inserted into the serial graph with the sequence $T_{ij} \rightarrow T_{kl}$. An edge exits from T_{ij} to T_{kl}, indicating that $T_{k,l}$ is processed after T_{ij};
2. If $Wset_{ij} \cap Rset_{kl} \neq \{\}$, then transaction T_{ij} is inserted into the serial graph with the sequence $T_{ij} \leftarrow T_{kl}$. An edge enters from T_{kl} to T_{ij}, indicating that T_{ij} is processed after T_{kl}.

After transactions are inserted into the serial graph, a check for cycles in the sequence of the serial graph is performed. The last arriving transaction is removed from the serial graph and aborted if a cycle is detected. However, the transaction validation process is declared successful if no cycle is detected. Then, the transactions are executed based on the sequence in the serial graph.

Upon passing validation, the values of objects in the object manager are updated, and the transaction is removed from the serial graph, releasing the LOCK from the object. Subsequently, the server propagates an update to all caches, storing the objects to update their values.

3.3 Hypothetical Transaction Execution on the Server Side

In this section, the hypothetical execution of transactions on the server side will be explained. The initial condition in the serial graph is when transactions within the serial graph are being processed for execution. When a transaction is executed on the server side, the system requires time to update the object values in the database until the server sends an update propagation to every existing client cache. Once this process is completed, the transaction is removed from the serial graph, and the commit process is finished.

1. Case 1

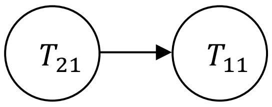


Figure 6. Initial state in the serial graph for case 1

Figure 6 shows initial state of the serial graph, where:

$$T_{11} = \{W_{11}(x)\}$$
$$T_{21} = \{R_{21}(x), W_{21}(y)\}$$

The server is currently validating transactions T_{11} and T_{21}, which are in the serial graph. Then, a validation request is sent by $T_{31} = \{R_{31}\,(y), W_{31}\,(z)\}$. Initially, the cache version brought by T_{31} will be checked by the Server. Assuming the cache version brought by T_{31} is valid. Then, the server checks the locking status of objects in transactions T_{11} and T_{21}, which are already in the serial graph. The object accessed by transaction T_{31} is currently in an UNLOCK mode because the object in T_{11} or T_{21} differs from T_{31}. Therefore, transaction T_{31} is inserted into the serial graph, as shown in **Figure 7**.

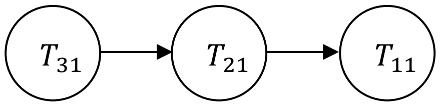


Figure 7. Final state in the serial graph for case 1

The execution sequence in the serial graph becomes $T_{31} \rightarrow T_{21} \rightarrow T_{11}$. Since there are no cycles in the serial graph, the validation process is successful.

2. Case 2

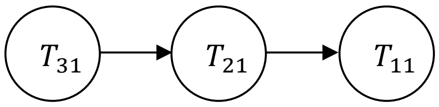


Figure 8. Initial State in The Serial Graph for Case 2

Figure 8 shows initial state of the serial graph, where:

$$T_{11} = \{W_{11}(x)\}$$
$$T_{21} = \{R_{21}(x). W_{21}(y)\}$$
$$T_{31} = \{R_{31}(y), W_{31}(z)\}$$

Then, a validation request is sent by transaction $T_{41} = \{W_{41}\,(z)\}$. The cache version of T_{41} will be checked by the server first. After ensuring its validity, the server checks the object locking. The object accessed by T_{41} is in a LOCK mode because its lock is held by T_{31}, causing T_{41} to fail the validation process and be aborted.

3. Case 3

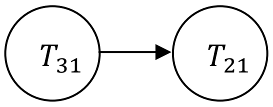


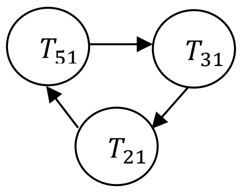
Figure 9. Initial State in The Serial Graph for Case 3

Figure 9 shows initial state of the serial graph, where:

$$T_{31} = \{R_{31}(y), W_{31}(z)\}$$
$$T_{21} = \{R_{21}(x), W_{21}(y)\}$$

Then, a validation request is sent by transaction $T_{51} = \{R_{51}\ (z), W_{51}\ (x)\}$. The Server will check the cache version of T_{51}. After ensuring its validity, the server checks the object locking. The object status in transaction T_{51} is in an UNLOCK mode. Subsequently, transaction T_{51} is inserted into the serial graph with the sequence $T_{51} \rightarrow T_{31}$ because $Rset(T_{51}) \cap Wset(T_{31}) \neq \{\ \}$, and $T_{21} \rightarrow T_{51}$ because $Rset(T_{21}) \cap Wset(T_{51}) \neq \{\ \}$, resulting in the serial graph condition as shown in Figure 10.

Figure 10. Final State in The Serial Graph for Case 3



The execution sequence of transactions in the serial graph encounters an issue because there is a cycle, resulting in transaction T_{51} failing the validation process. Consequently, transaction T_{51} is removed from the serial graph and aborted.

4. CONCLUSIONS

Based on the research, a new scheme has been developed through modifications to the Validation Queue (VQ) scheme. These modifications offer an alternative scheme for further development from the VQ scheme. This scheme is called the Extended Serial Graph-Validation Queue (SG-VQ) scheme. The Extended SG-VQ scheme utilizes a serial graph with a locking strategy as the validation algorithm on the server side. It employs a validation queue as the validation algorithm on the cache side. Through a series of hypothetical transaction scenarios across three cases, this study validates the effectiveness of the Extended SG-VQ, demonstrating its capability to utilize serial graphs, resolve conflicts, and identify cyclic patterns. This research can be advanced to the simulation stage to analyze the optimal environment for implementing the proposed scheme. Furthermore, the scheme can be further developed by incorporating attributes that more accurately reflect real-world conditions, such as implementating of prioritization mechanisms.

REFERENCES

[1] Q.-V. Dang and C.-L. Ignat, “Performance of real-time collaborative editors at large scale: user perspective,” in 2016 IFIP Networking Conference and Workshops, pp. 548–553, May 17-19, 2016.
[2] X. Liu and J. Shen, “Research on the desktop sharing mechanism of online teaching system,” in 3rd International Conference on Mechatronics and Industrial Informatics, pp. 790–793, Oct. 30-31, 2015.
[3] B. Ashadevi and P. Muthamil Selvi, “Google docs: an effective collaborative tool for graduates to perform academic activities in the cloud,” *IJDR*, vol. 7, no. 8, pp. 14626–14633, August 2017.
[4] M. Härtwig and S. Götz, “Mobile modeling with real-time collaboration support,” Journal of Object Technology, vol. 21, no. 3, pp. 1-15, July 2022.
[5] T. Connolly and C. Begg, Database Systems: A Practical Approach in Design, Implementation, and Management Database, 6th ed. Essex: Pearson Education, 2015, 99-100.
[6] M. Alkhazaleh, S. A. Aljunid, and N. Sabri, “A review of caching strategies and its categorizations in information centric network,” *JATIT*, vol. 97, no. 19, pp. 4996–5011, October 2019.
[7] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, “Relaxed currency serializability for middle-tier caching and replication,” in Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data pp. 599–610, Jun. 27-29, 2006.
[8] F. Bukhari and S. Shrivastava, “An efficient distributed concurrency control scheme for transactional systems with client-side caching,” in Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications, HPCC-2012 - 9th IEEE International Conference on Embedded Software and Systems, pp. 1074–1081, Jun. 25-27, 2012.
[9] V. T. S. Shi and W. Perrizo, “A new method for concurrency control in centralized database systems,” in Proceedings of the ISCA 17th International Conference Computers and Their Applications, pp. 184–187, Apr. 4-6, 2002.
[10] F. Bukhari, Maintaining Consistency in Client-Server Database Systems with Client-Side Caching. Doctoral [Dissertation]. Newcastle upon Tyne: Newcastle Univ, 2012. [Online]. Available: Newcastle University Theses.
[11] Y. Heryadi and I. Sonata, Dasar-Dasar Graph Machine Learning dan Implementasinya Menggunakan Bahasa Python. Yogyakarta: GAVA MEDIA, 2022, 1-4.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database System. Addison-Wesley, 1987, 1-34.
[13] R. Diestel, *Graph Theory*, 5th ed. Berlin: Springer, 2017.
[14] Fathansyah, *Basis Data*, 3rd ed. Bandung: Informatika Bandung, 2018, 312-318.
[15] S. B. Gupta and A. Mittal, *Introduction to Database Management System*, 2nd ed. New Delhi: Laxmi Publications, 2017, 353.
[16] S. kanungo and morena rustom. D, “Analysis and comparison of concurrency control techniques,” IJARCCE, vol. 4, no. 3, pp. 245–251, Mar. 2015.