J. Indones. Math. Soc.
Vol. 31, No. 02 (2025), pp. 1–13.

Mathematical Study for Proving Correctness of the Serial Graph-Validation Queue Scheme

Fitra Nuvus Salsabila[1,a*], **Fahren Bukhari**[1,b], **and Sri Nurdiati**[1,c]

Department of Mathematics, IPB University, Indonesia.
[1,a]**fitrasalsabila@apps.ipb.ac.id**, [1,b]**fahrenbu@apps.ipb.ac.id**,
[1,c]**nurdiati@apps.ipb.ac.id**

Abstract. Numerous studies have been conducted to develop concurency control schemes that can be applied to client-server systems, such as the Validation Queue (VQ) scheme, which uses object caching on the client side. This scheme has been modified into the Serial Graph-Validation Queue (SG-VQ) scheme, which employs validation algorithms based on queues on the client side and graphs on the server side. This study focuses on verifying the correctness of the SG-VQ scheme by using serializability as a mathematical tool. The results of this study demonstrate that the SG-VQ scheme can execute its operations correctly, in accordance with Theorem 4.16, which states that every history (H) of SG-VQ is serializable. Implementing a cycle-free transaction graph is a necessary and sufficient condition to achieve serializability. To prove Theorem 4.16, mathematical statements involving ten definitions, two propositions, and three lemmas have been formulated.

Key words and Phrases: client-server, concurrency control, correctness, serializability.

1. INTRODUCTION

According to Ali *et al.* [1], future generations will have cutting-edge technology to connect everyone wherever they are. This development has been felt for several years now. Remote work, commonly known as Work From Home (WFH) or Work From Anywhere (WFA), is increasingly accepted in various fields. The ability to perform information-based work remotely has significantly increased in the past decade. Collaboration tools can be a solution to overcome these challenges. Currently, the typical solution is applications that allow real-time collaboration, enabling multiple users to work together simultaneously, such as collaborating on

*Corresponding author
2020 Mathematics Subject Classification: 68Q85
Received: 05-12-2023, accepted: 10-01-2025.

editing a document through a real-time collaboration (RTC) [2]. The client-server system is an architecture suitable for applications that support real-time collaboration [3]. The client-server system is a distributed computing between two types of independent and autonomous entities known as the server and the client [4]. When multiple clients simultaneously access data in the same database, and one of the clients makes changes to the data, this can trigger inconsistent data. Therefore, in a client-server system, a mechanism is required to regulate access to shared resources by multiple clients simultaneously to ensure data consistency. This mechanism is called concurrency control [5]. However, in concurrency control, complexity can arise in completing transactions and sometimes increase the load on the server, affecting performance [6]. In recent years, caching has become an effective solution to reduce and balance the increasing traffic in communication networks [7].

Bukhari and Shrivastava [8] introduced a scheme in the client-server system that uses object caching on the client side. This scheme is called Validation Queue (VQ). Jauhari [6] modified the VQ scheme, particularly on the server side. The modified VQ scheme is called the Serial Graph-Validation Queue (SG-VQ) scheme. The modification focuses on the server side and is based on a graph.

After the modification, testing is required to prove the correctness of the modified scheme. Applying a cycle-free transaction graph is a necessary and sufficient condition to achieve serializability [9]. Therefore, this research focuses on testing the correctness of the SG-VQ scheme. The expected processing or execution is processing that is free from overlapping transactions (interleaving). Non-overlapping transaction execution is called serial execution. Serial execution can be achieved by processing transactions alternately or one by one. The advantage of serial execution is the guarantee of data consistency because there is no overlap between transactions. Each transaction views data in a consistent state and is not affected by changes made by other transactions. Therefore, serial execution is considered correct.

However, in a concurrent transaction environment, the expected processing is the system's ability to execute multiple transactions simultaneously or concurrently as if the transactions were executed sequentially (serially). Such execution is called serializable execution [9]. So, in an environment that supports concurrent data processing, serializable execution is the desired target because it allows for efficiency and high performance without sacrificing data consistency. Serializable execution has an equivalent effect to serial execution, so it is also considered correct [10]. Therefore, in this research a mathematical tool called serializability is used to prove the correctness of the SG-VQ scheme. Serializability is a crucial criterion to ensure correctness [11].

2. SERIAL GRAPH-VALIDATION QUEUE SCHEME

Below is a comparison table between the Validation Queue (VQ) scheme by Bukhari and Shrivastava [12] and Serial Graph-Validation Queue (SG-VQ) scheme by Jauhari [6]:

Table 1. SG-VQ and VQ Scheme Comparison

Features	VQ Scheme	SG-VQ Scheme
Architecture	Client-Server	Client-Server
Validation	Cache: Queue based	Cache: Queue based
	Server: Queue based	Server: Graph based

Based on the comparison table, it can be observed that modifications were only made on the server side. However, in this research, the proof of correctness for the SG-VQ scheme is not focused solely on the server side but on the scheme as a whole.

2.1. **Element.**

Element is a part of a transaction. A corresponding element is generated based on the request. An element consists of:

a. Transaction Identifier (TID);
b. Element Type:
    (1) Read Element: Includes a set of data to be read, known as a ”readset”;
    (2) Commit Element: May include a set of data to be written, known as a ”writeset”;
    (3) Update Propagation Element: Similar to an update from a remote transaction. This element includes the readset and writeset of the remote transaction.
    (4) Validated Element: Corresponds to a validated transaction. This element includes the readset and writeset of a transaction.
    (5) Local Validated Element: Represents a locally validated transaction;
    (6) Cache Element: Corresponds to a cache transaction consisting of a collection of client’ cached objects.
c. Object identifier fields containing a list of objects to be accessed with additional information;
d. Queue management link.

The elements and fields are depicted as in Figure 1 below,

TID	Element Type	Element Identifier	Links

Figure 1. Element Structure

2.2. **Cache Side Validation Algorithm.**

The cache-side validation algorithm is initiated by the local cache manager during the validation of local transactions. This algorithm is responsible for verifying the accuracy of transaction execution by examining the order in which transactions are executed, aiming to prevent the commit of inaccurate transaction executions. When an execution of a transaction is detected to interleave with another transaction, it is returned as failure. If not, it is returned as success.

The cache-side validation algorithm uses VQ to record the sequence of local executions. VQ consists of elements like Read, Commit, Validated, Local Validated, and Update Propagation. The Update Propagation element represents the process of executing remote update transactions. These transactions include the readset and writeset of the update and are sent to the local manager through an Update Propagation message from the server. When local transactions receives requests to read or commit, the Read or Commit elements are added to VQ.

Before a transaction is completed, it sends a request to the local cache manager to commit. Once the local cache manager receives this request, it creates a commit element and places it in VQ. The transaction is then validated.

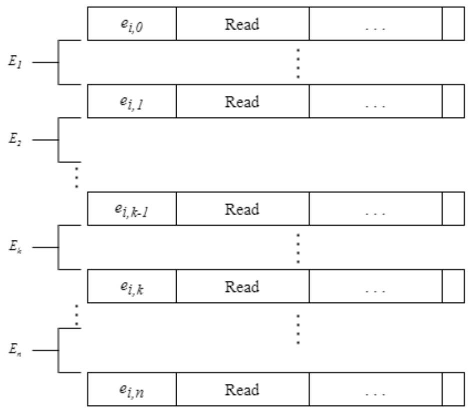

Figure 2. Queue Structure

Consider a transaction T_i that has n elements: $e_{i,0}$ through $e_{i,n}$. There are three sets of elements that connect these transactions: E_1 goes between $e_{i,0}$ and $e_{i,1}$, E_2 goes between $e_{i,1}$ and $e_{i,2}$, and E_j goes between $e_{i,j-1}$ and $e_{i,j}$ for $1 \leq j \leq n$. Suppose that E_k contains an element e' that divides E_k into two parts, P and Q. Here, we can represent E_k as $P; e'; Q$, where P and/or Q could be empty. For a transaction T_i to pass the validation process, it must meet one of these two conditions:

a. Condition 1. An element or a combination of 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 $j = 0, 1, \ldots n-1$.

b. Condition 2. 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+1}, for every $j = 0, 1, \ldots k-1$, and every element in P but the combined elements conflict with e', for $k = 1, 2, \ldots, n$. Then, 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 e' and any element in Q.

Read-only transactions are merged into Validated elements if they pass the validation process. When an update transaction is validated, all elements are merged into Local Validated elements, and the local cache manager sends a commit request to the server. The Local Validated elements are changed to Validated elements if the server's response is positive. Otherwise, they are discarded if the server responds with an abort. In the SG-VQ scheme's cache-side validation algorithm, read-only transactions are validated if they satisfy Condition 1 or 2. If not, they fail. Update transactions pass the validation process if they meet Condition 1. If not, they fail.

2.3. Server Side Validation Algorithm.

The validation algorithm on the server side is called the SG algorithm. The SG algorithm consists of two parts: commit request processing and validation processing. When a commit request message is sent to the server, it checks whether the message carries the latest cache version. If the cache version carried does not match the latest version, a message is sent to the original cache manager to verify the cache version of the message and update it first. Then, if the cache version is up to date, the validation process is carried out. A commit message will be sent to the object manager if it passes the validation process. If it fails, all transaction elements being committed are removed, and an abort message will be sent to the original cache manager. In this validation process, considering that T_{ij} is the transaction to be validated on the server side, and Σ is the set of transactions being validated on the server side, if there is a transaction T_{kl} with $\forall T_{kl} \in \Sigma$, then it is checked whether there is a conflict between T_{ij} and T_{kl}. If conflicts exist between these transactions, T_{ij} is aborted, and a failure is returned. However, if there are no conflicts, T_{ij} is inserted into the serial graph, meaning a node is created containing information about transaction T_{ij}, and the direction of the edge for transaction T_{ij}, indicating its execution order, is determined. If $wset(T_{ij}) \cap rset(T_{kl}) \neq \emptyset$, a serial graph is created, as shown in Figure 3. This serial graph shows that transaction T_{ij} will be executed after transaction T_{kl}.

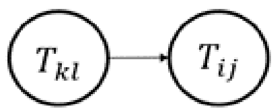

FIGURE 3. T_{kl} precedes T_{ij}

Furthermore, if $wset(T_{kl}) \cap rset(T_{ij}) \neq \emptyset$, a serial graph is formed, as shown in Figure 4. This serial graph indicates that transaction T_{ij} will be executed before transaction T_{kl}.

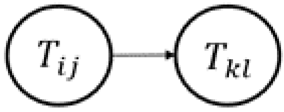

FIGURE 4. T_{ij} precedes T_{kl}

3. SERIALIZABILITY

In an application where multiple transactions are executed concurrently, it's essential to establish an order for carrying out operations because only one operation can be executed at a time. This sequence of transaction execution is known as a schedule. According to Connolly and Begg [13], a schedule is a sequence of operations by a set of concurrent transactions that maintains the operations' order within each transaction. Bernstein *et al.* [12] stated that the theory of serializability provides mathematical tools to verify the correctness of a scheduler. In the theory of serializability, "history" refers to a structure representing a set of transactions executed concurrently. An execution is considered serializable if it is equivalent to a serial execution of the same transactions. Two histories H and H^* are equivalent if:

a. both histories have the same sets of transactions and operations;
b. operations p_i belonging to transaction T_i conflicts with q_j belonging to transaction T_j are not present in H with $a_i, a_j \notin H$, where a represents an abort. If $p_i <_H q_j$, then $p_i <_{H^*} q_j$, where $<_H$ indicates the order in history H.

4. MAIN RESULTS

Definition 4.1. [10] *An element $e_{k\ell}$ represents the ℓ-th element in transaction T_k where:*

a. $e_{k\ell} \subseteq r_{k\ell}(x), w_{k\ell}(x) | x := object$;
b. $rset(e_{k\ell}) \cap wset(e_{k\ell}) = \emptyset | rset(e_{k\ell}) := readset$ *and* $wset(e_{k\ell}) := writeset$.

Definition 4.2. [10] *If element e_{km} is a compound element resulting from the merging of elements e_{kp} and e_{kq}, then $wset(e_{km}) = wset(e_{kp}) \cup wset(e_{kq})$ and $rset(e_{km}) = rset(e_{kp}) \cup rset(e_{kq})$.*

Definition 4.3. [10] *Element $e_{k\ell}$ and e_{mn} are conflicting if and only if $k \neq m$ and satisfy one of the following statement:*

a. $wset(e_{k\ell}) \cap wset(e_{mn}) \neq \emptyset$;
b. $wset(e_{k\ell}) \cap rset(e_{mn}) \neq \emptyset$;

c. $rset(e_{k\ell}) \cap wset(e_{mn}) \neq \emptyset$.

Definition 4.4. [10] *Transaction T_k and T_ℓ are conflicting if and only if their elements or compound elements are conflicting.*

Definition 4.5. [10] *A transaction T_k is considered as a partial order with ordering relation $<_k$ where:*

a. $T_k = e_{k1}, e_{k2}, \dots, e_{kn} \cup a_k, c_k | a_k := abort, c_k := commit;$
b. $a_k \in T_k$ *only if* $c_k \notin T_k$;
c. *if* t *is* a_k *or* c_k*, for any element* e_{kl} *in* $T_k, e_{kl} <_k t$;
d. *if* $r_{k\ell}(x) \in e_{k\ell}$ *and* $w_{mn}(x) \in e_{mn}$*, then* $e_{k\ell} <_i e_{mn}$.

Definition 4.6. [12] *A complete history H over transaction T is considered as a partial order with ordering relation $<_H$ where:*

a. $H = \bigcup_{k=1}^{n}$;
b. $<_H \supseteq \bigcup_{k=1}^{n} <_k$;
c. *for any two conflicting elements* $x, y \in H$*, either* $x <_H y$ *or* $y <_H x$.

Definition 4.7. [12] *The Serialization Graph (SG) for a complete history H over a set of transactions $T = T_1, ..., T_n$ is a directed graph denoted as $SG(H)$. The nodes represent the transactions in T, and the edges are all $T_k \rightarrow T_\ell (k \neq \ell)$ such that one of T_k's elements precedes and conflicts with one of T_ℓ's elements in H.*

Definition 4.8. [14] *Distributed serialization order: A global history H is considered serializable if there exists a total ordering of T such that for each pair of conflicting element $e_k \in T_k$ and $e_\ell \in T_\ell$ where $k \neq \ell$, e_k precedes e_ℓ in any $H_1, ..., H_n$ if and only if T_k precedes T_ℓ in the total ordering.*

Definition 4.9. [10] *Suppose H_i is a complete history at cache side i where $i = 1, 2, .., n$. H_i is a partial order over a set of transaction T from cache side i with ordering relation $<_{H_i}$ where:*

a. $H_i = T_{i,1} \cup T_{i,2} \cup ... \cup T_{i,n_i}$;
b. $<_{H,i} \supseteq <_1 \cup <_2 \cup ... \cup <_{ni}$;
c. *for any two conflicting elements* $x, y \in H_i$*, either* $x <_{H_i} y$ *or* $y <_{H_i} x$.

Definition 4.10. [10] *Suppose $T = T_1, ..., T_n$ is a set of transaction, H is a complete history generated by the SG-VQ algorithm, and the system has n cache sides. History H is considered as a partial order over transaction T with ordering relation $<_H$ where:*

a. $H = H_1 \cup H_2 \cup ... \cup H_n$ *where* H_n *is a complete history at cache side* n *and* H_n *is partial order over transaction* T;
b. $<_H \supseteq <_{H_1} \cup <_{H_2} \cup ... \cup <_{H_n}$;
c. *for any two conflicting elements* $x, y \in H$*, either* $x <_H y$ *or* $y <_H x$.

Proposition 4.11. *Suppose the cache side algorithm of the SG-VQ scheme generates a local history H_i at cache side i. If T_k is a transaction from cache side i, then the execution of T_k's elements at cache side i is equivalent to a single element, denoted as e_k.*

Proposition 4.12. *Let there be H_i as a local history at cache side i where $i = 1, 2, ..., n$, a set of transaction $T = T_1, T_2, ...$, and H is a global history. Suppose T_k and T_ℓ are from cache side i, if $e_k <_{H_i} e_\ell$, then $e_k <_H e_\ell$.*

Lemma 4.13. *Let $T = T_1, T_2, ...$ be a set of transactions, and there are n clients in the system. Based on the SG-VQ algorithm, each client executes a serial local history, $H_1, H_2, ..., H_n$. The SG-VQ scheme's global history H is defined over T. If $e_k <_H e_\ell$, then $e_k <_{H_i} e_\ell$ for client i that generates both transactions, where $i = 1, \ldots, n$.*

Proof. Suppose clients k and ℓ each create transactions T_k and T_ℓ. If $e_k <_{H_i} e_\ell$, then e_k conflicts with e_ℓ. According to Definition 4.3, three cases demonstrate how e_k conflicts with e_ℓ. Here is an overview of the conflict occurrence process:

(1) $rset(e_k) \cap wset(e_\ell) \neq \emptyset$
In this case, e_k reads an object on client k, which is then updated by e_ℓ. Since e_k is a read-only transaction, its validation is only local. As a result, there is no $e_k <_{H_\ell} e_\ell$ on client ℓ. However, globally, $e_k <_H e_\ell$ still holds.

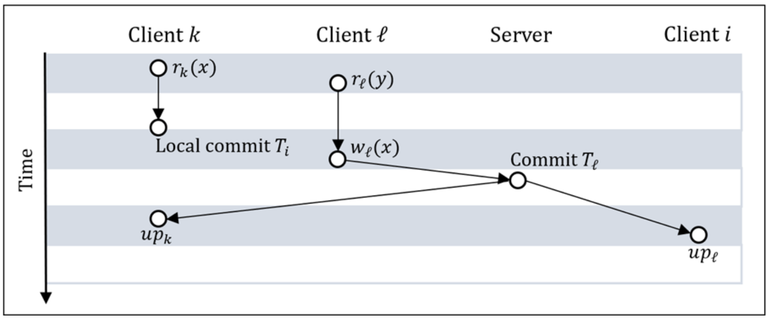

FIGURE 5. Case $rset(e_k) \cap wset(e_l) \neq \emptyset$

(2) $wset(e_k) \cap rset(e_\ell) \neq \emptyset$
In this case, e_k updates an object that is later read by e_ℓ, resulting in a write-read conflict. Since $e_k <_{H_\ell} e_\ell$, the commit of T_k precedes T_ℓ on the server, and therefore, the cache manager on client ℓ executes update propagation from T_k before conflicting object is read by T_ℓ. Update transactions are not allowed to read stale objects. As a result, the commit of T_k precedes T_ℓ on the server. If the update transaction by T_ℓ reads a stale object that T_k has updated, then the cache version carried by T_ℓ becomes outdated, and T_ℓ is rolled back to its original cache for revalidation. Hence, $e_k <_{H_\ell} e_\ell$ and $e_k <_{H_i} e_\ell$ holds for every client $i = 1, ..., n$ that generates both transactions.

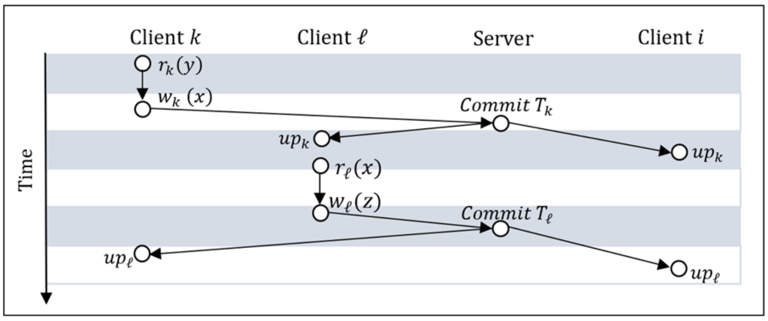

FIGURE 6. Case $wset(e_k) \cap rset(e_\ell) \neq \emptyset$

(3) $wset(e_k) \cap wset(e_\ell) \neq \emptyset$
In this case, e_k and e_ℓ update the same object, resulting in a write-write conflict. Since $e_k <_{H_\ell} e_\ell, T_k$ is committed to the server first. If the update propagation from T_k is received by client ℓ after T_ℓ has already been locally committed, then when T_ℓ is validated on the server, the cache version it carries is no longer valid and is rolled back for revalidation. As a result, the commit of T_k precedes T_ℓ on the server, and therefore, $e_k <_{H_\ell} e_\ell$ and $e_k <_{H_i} e_\ell$ hold for every client $i = 1, \ldots, n$ that generates both transactions.

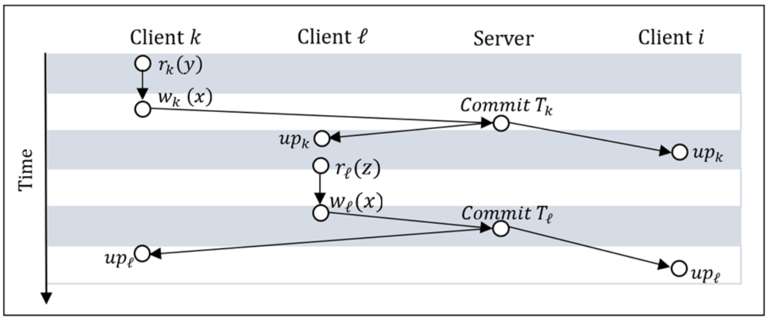

FIGURE 7. Case $wset(e_k) \cap wset(e_\ell) \neq \emptyset$

□

Lemma 4.14. *Let $T = T_1, T_2, \ldots$. The SG-VQ algorithm produces a complete history of H over T. The serialization graph SG is defined over H. If $T_k \to T_\ell$ exists in $SG(H)$, then the validated element e_k of T_k conflicts with the validated element e_ℓ of T_ℓ in H, thus $e_k <_H e_\ell$.*

Proof. If $T_k \to T_\ell$ exists in $SG(H)$, then, according to Definition 4.8, there exists e_k conflicting with e_ℓ, and e_k precedes e_ℓ. Therefore, $e_k <_H e_\ell$. □

Lemma 4.15. *The SG-VQ algorithm generates a complete history H. Suppose there is a path $T_1 \to T_2 \to ... \to T_n$ in $SG(H)$, where $n > 1$, then e_1 precedes e_n in H, $e_1 <_H e_n$.*

Proof. Induction will be used to prove the statement. Let $n = 2$ as the induction base. In accordance with Lemma 4.14, we can identify a path $T_1 \to T_2$ in $SG(H)$ where an edge of $e_1 \in T_1$conflicts with an edge of $e_2 \in T_2$ This implies that $e_1 <_H e_2$. Hence, Lemma 3 holds true for $n = 2$.

Assuming Lemma 4.15 is true for $n = k$ where $k \geq 2$ and k is a positive integer, we can show that Lemma 4.15 also holds true for $n = k + 1$. Consider the path $T_1 \to T_2 \to ... \to T_k \to T_{k+1}$ in $SG(H)$. We will prove that e_1 precedes e_{k+1} in the total order H, $e_1 <_H e_{k+1}$.

Based on the assumption that Lemma 4.15 is true for $n = k$, the following can be derived:

(1) In the total order H, e_1 precedes e_k because there is a path $T_1 \to T_2 \to ... \to T_k$ in $SG(H)$. According to Lemma 4.14, e_1 conflicts with e_k, and e_1 precedes e_k, or it can be denoted as $e_1 <_H e_k$.
(2) In the total order H, e_k precedes e_{k+1} because there is a path $T_k \to T_{k+1}$ in SG(H). According to Lemma 4.14, e_k conflicts with e_{k+1}, and e_k precedes e_{k+1}, or it can be denoted as $e_k <_H e_{k+1}$.

Since e_1 precedes e_k and e_k precedes e_{k+1}, based on the transitive property of the total order, it can be concluded that e_1 precedes e_{k+1} in the total order H, which can be denoted as $e_1 <_H e_{k+1}$. By proving this inductively, it has been shown that if Lemma 4.15 is true for $n = k$, then it is also true for $n = k+1$. Thus, it can be concluded that Lemma 4.15 is true for all $n > 1$. □

Theorem 4.16. *Every history H from SG-VQ is serializable.*

Proof. To prove this, we will use contradiction. Suppose there is a cycle in $SG(H)$, denoted by $T_1 \to T_2 \to ... \to T_n \to T_1$, with $n > 1$. According to Lemma 4.15, one element from T_1 conflicts with another element from T_1 in history H. This condition contradicts Proposition 4.11, which explains the singularity of elements. Hence, $SG(H)$ does not have cycles, and H is serializable. □

5. IMPLEMENTATIONS

In collaborative editing systems, users work simultaneously on the same document. SG-VQ can be used to ensure that changes made by users on different devices remain consistent, even when performed concurrently. This approach prevents data conflicts when multiple users modify the same data object.

Jauhari [6] conducted hypothetical transaction executions involving three validation cases on the server. The first case describes a transaction validated and added to a serial graph containing the set of ongoing transactions. The second case identifies conflicts arising from the intersection of the writeset between the new transaction and the ongoing transactions, leading to the rejection (abort) of the new transaction. The last case explains a cycle in the serial graph.

The application of SG-VQ is not limited to editing systems. However, this section briefly explains these three cases within an editing system as an example of SG-VQ implementation. A document editing system implementing RTC (Real-Time Collaboration) allows multiple users to edit a document simultaneously. Each modification is treated as a transaction T that the server must validate to avoid conflicts.

5.1. **Case 1.**

Two transactions are in progress. Transaction $T_{11} = \{w_{11}(x)\}$ represents a user editing part x of the document, for example, the first paragraph. Transaction $T_{21} = \{r_{21}(x), w_{21}(y)\}$ represents another user reading the first paragraph (x) and editing the second paragraph (y). The server forms an initial serial graph where $T_{21} \to T_{11}$. A new transaction then arrives: $T_{31} = \{r_{31}(y), w_{31}(z)\}$, where the third user reads the second paragraph (y) and edits the third paragraph (z). The server accepts the request from T_{31} and begins the validation process. The validation process starts by ensuring that the cache version of T_{31} is the latest, meaning that the editing system verifies that the third user is working on the most up-to-date version of the document—next, the system checks for conflicts. If no user is editing the same part of the document, the document remains safe. If a conflict is detected, the editing system will arrange the transactions in the correct order. In this case, the conflict check shows that $T_{31} \cap T_{11} = 0$, meaning there is no conflict between T_{31} and T_{11}. However, $T_{31} \cap T_{21} \neq 0$ because $rset_{31} \cap wset_{21} \neq 0$; transaction T_{31} reads part y, which is being written by T_{21}. Since there is no cycle in the serial graph, T_{31} is added to the execution order, resulting in a new serial graph order of $T_{31} \to T_{21} \to T_{11}$. Thus, the order of operations in this case is as follows: the third user reads the second paragraph and writes to the third paragraph, the second user completes editing the first and second paragraphs, and the first user finishes editing the first paragraph.

5.2. **Case 2.**

Case 2 is the continuation of Case 1. Three users edit the document simultaneously, each making changes to different sections. At this point, the serial graph has already established the order $T_{31} \to T_{21} \to T_{11}$. User 4 T_{41} arrives and makes changes to the section being worked on by T_{31}, specifically the third paragraph (z), creating $T_{41} = \{w_{41}(z)\}$, and submits a validation request to check whether their changes can be accepted without interfering with others. The editing system will check if any changes conflict with previous edits. As in the previous case, the validation starts by checking the version of the document edited by T_{41}. Since T_{31} is writing to the third paragraph, the same section that T_{41} wants to modify, the system detects a write-write conflict, $wset_{31} \cap wset_{41} \neq 0$. Because there is a conflict between the changes made by user 3 and user 4, the changes from user 4 will be aborted. Therefore, the conflicting edit will not be accepted.

5.3. **Case 3.**

We encounter a scenario where three users are editing the document simultaneously. Initially, there are two users. User 1 generates transaction $T_{11} = r_{11}(y), w_{11}(x)$, which involves reading the second paragraph and editing the first paragraph (x), while user 2 generates transaction $T_{21} = r_{21}(x), w_{21}(y)$, which involves reading the first paragraph (x) and editing the second paragraph (y). These changes do not cause conflicts, thus the execution order is $T_1 1 \rightarrow T_2 1$. Then, user 3 joins and attempts to edit the document. User 3 makes transaction $T_{31} = r_{31}(z), w_{31}(x)$, which involves reading the third paragraph (z) and attempting to modify the first paragraph (x). When the system checks the changes for T_{31}, it shows that $rset_{31} \cap wset_{11} \neq 0$, thus $T_{31} \rightarrow T_{11}$ and $rset_{21} \cap wset_{31} \neq 0$ thus $T_{21} \rightarrow T_{31}$. Thefore a cycle is created: $T_{31} \rightarrow T_{11} \rightarrow T_{11} \rightarrow T_{31}$. To maintain document consistency, the editing system rejects the changes made by T_{31} and the system removes these changes from the list of modifications to be applied.

6. CONCLUDING REMARKS

In proving the correctness of the transaction execution produced by the SG-VQ scheme, ten definitions, two propositions, and three lemmas have been elaborated to establish Theorem 4.16. Based on the proof of this theorem, it can be concluded that every history H produced by the SG-VQ scheme is serializable. Therefore, it has been theoretically proven that the Serial Graph-Validation Queue (SG-VQ) scheme can execute transactions correctly.

Acknowledgement. The author would like to express gratitude to M. Fakhri J. as a fellow researcher who contributed to developing the Serial Graph-Validation Queue scheme.

REFERENCES

[1] S. Ali, R. Alauldeen, and R. A. Khamees, "What is client-server system: Architecture, issues and challenge of client-server system (review)," *Recent Trends in Cloud Computing and Web Engineering*, vol. 2, no. 1, pp. 1–6, 2020. https://doi.org/10.5281/zenodo.3673071.

[2] Q.-V. Dang and C.-L. Ignat, "Performance of real-time collaborative editors at large scale: User perspective," in *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pp. 548–553, IEEE, May 2016. https://doi.org/10.1109/IFIPNetworking.2016.7497258.

[3] M. Hartwig and S. Gartz, "Mobile modeling with real-time collaboration support," *J. Object Technol.*, vol. 21, no. 3, pp. 1–15, 2022. https://doi.org/10.5381/jot.2022.21.3.a2.

[4] S. Kumar, "A review on client-server based applications and research opportunity," *Int. J. Recent Sci. Res.*, vol. 10, no. 7, pp. 33857–33862, 2019. https://doi.org/10.24327/ijrsr.2019.1007.3768.

[5] M. Kaur and H. Kaur, "Concurrency control in distributed database system," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 7, pp. 1443–1447, 2013. https://api.semanticscholar.org/CorpusID:7899973.

[6] M. F. Jauhari, "Skema serial graph-validation queue pada sistem klien-server," *Magister Theses, IPB University*, 2024. https://repository.ipb.ac.id/handle/123456789/152689.

[7] F. Rezaei and B. H. Khalaj, "Stability, rate, and delay analysis of single bottleneck caching networks," *IEEE Trans Commun.*, vol. 64, no. 1, pp. 300–313, 2016. https://doi.org/10.1109/TCOMM.2015.2498177.

[8] F. Bukhari and S. Shrivastava, "An efficient distributed concurrency control scheme for transactional systems with client-side caching," in *Proc. 14th International Conference on High Performance Computing and 9th International Conference on Embedded Software and Systems*, 2012. https://doi.org/10.1109/HPCC.2012.157.

[9] T. Wang, R. Johnson, A. Fekete, and I. Pandis, "Efficiently making (almost) any concurrency control mechanism serializable," *VLDB J.*, vol. 26, no. 4, pp. 537–562, 2017. https://doi.org/10.1007/s00778-017-0463-8.

[10] F. Bukhari, "Maintaining consistency in client-server database systems with client-side caching," *Doctoral dissertation, Newcastle University.* http://theses.ncl.ac.uk/jspui/handle/10443/1789.

[11] A. Mhatre and R. Shedge, "Comparative study of concurrency control techniques in distributed databases," in *Proc. 4th International Conference on Communication Systems and Network Technologies*, 2014. https://doi.org/10.1109/CSNT.2014.81.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database System.* Addison-Wesley Longman, 1987. https://www.sigmod.org/publications/dblp/db/books/dbtext/bernstein87.html.

[13] T. Connolly and C. Begg, *Database Systems: A Practical Approach in Design, Implementation, and Management Database, Sixth Edition.* Pearson Education, 2015.

[14] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981. https://doi.org/10.1145/356842.356846.