Proof: Since, in the TO scheme, transactions are serialized in timestamp order, if T_i is serialized before T_j in S_k, $T_i, T_j \in \tau_k$, then T_i's timestamp must be smaller than T_j's timestamp. Thus, in S_k, T_i must have been assigned a timestamp before T_j is assigned one (assuming timestamps are assigned in an increasing order). $\Box$

Lemma 16: If site s_k follows a validation protocol, then any function that maps every transaction $T_i \in \tau_k$ to its operation that results in its validation is a serialization function for s_k.

Proof: Since validation protocols ensure that transactions are serialized in the order in which they are validated, if T_i is serialized before T_j in S_k, $T_i, T_j \in \tau_k$, then T_i must have been validated in S_k before T_j is validated. $\Box$

-Appendix E-

Central to the design of any of the schemes to ensure global serializability is the requirement that GTM$_1$ be able to determine operations in $ser(S)$. A serialization function for site s_k depends on the concurrency control protocol followed by s_k.

Lemma 14: If site s_k follows the 2PL protocol, then any function that maps every transaction $T_i \in \tau_k$ to one of its operations that executes between the time T_i obtains its last lock and the time it releases its first lock, is a serialization function for s_k.

Proof: Let ser be a function that maps every transaction $T_i \in \tau_k$ to one of its operations that executes between the time T_i obtains its last lock and the time it releases its first lock. We need to show that for any pair of transactions $T_i, T_j \in \tau_k$, if T_i is serialized before T_j in S_k, then $ser(T_i) \prec_{S_k} ser(T_j)$. Since T_i is serialized before T_j in S_k, there exist transactions, say, $T_1, T_2, \ldots, T_r$ in S_k such that T_i conflicts with T_1, T_1 conflicts with T_2, …, T_r conflicts with T_j. We show that, in S_k, T_i releases its first lock before T_1 releases its first lock. Since T_i conflicts with T_1 and is serialized before T_1, T_i releases its first lock before T_1 obtains all its locks. Since s_k follows the 2PL protocol, T_1 obtains all its locks before it releases its first lock. Thus, in S_k, T_i releases its first lock before T_1 releases its first lock.

Using a similar argument, it can be shown that, in S_k, T_1 releases its first lock before T_2 releases its first lock, and so on. Thus, it follows that T_i releases its first lock before T_r releases its first lock. Also, T_r releases its first lock before T_j obtains its last lock. Thus, T_i releases its first lock before T_j obtains its last lock, and as a result, $ser(T_i) \prec_{S_k} ser(T_j)$. □

Corollary 3: If site s_k follows the strict 2PL protocol, then any function that maps every transaction $T_i \in \tau_k$ to its commit operation is a serialization function for s_k.

Proof: Transaction T_i obtains all its locks before it commits and releases its first lock only after it commits. □

Thus, if site s_k follows the strict 2PL protocol, $ser_k(G_i)$, for a global transaction G_i, is c_{ik}, G_i's commit operation at site s_k, which can be easily identified by GTM$_1$. However, determining $ser_k(G_i)$ if site s_k follows a simple 2PL protocol (that is not strict 2PL) is more complicated, and in order to identify $ser_k(G_i)$ for a global transaction G_i, it is necessary for GTM$_1$ to exploit the nature of local DBMS interfaces, and the manner in which transactions obtain and release locks at site s_k. If the local DBMS interface at s_k provides for explicit lock and unlock operations, then GTM$_1$ can identify $ser_k(G_i)$ without any problem since it has explicit knowledge of when the last lock is obtained or the first lock is released by G_i. However, in most local DBMSs, transactions, when they execute, obtain and release locks internally, and as a result, GTM$_1$ may have no knowledge of when transactions obtain and release locks. In such cases, GTM$_1$ can use indirect means (e.g., knowledge of the execution of G_i's operations) in order to identify $ser_k(G_i)$. For example, if G_i obtains the lock for a data item at site s_k only when it first accesses the data item and not earlier, then $ser_k(G_i)$ could be chosen by GTM$_1$ to be G_i's operation that first accesses the last data item accessed by G_i at site s_k. If, however, G_i obtains locks on certain data items at site s_k before it accesses the data items, then if G_i's operation that first accesses the last data item accessed by it at s_k is to be treated as $ser_k(G_i)$, then G_i would need to perform the following additional steps: After the first access to the last data item accessed by G_i at s_k, G_i reaccesses all the data items accessed by it at s_k (reaccessing all the data items ensures that G_i is holding all its locks when it first accesses the last data item accessed by it at site s_k).

Lemma 15: If site s_k follows the TO protocol, then any function that maps every transaction $T_i \in \tau_k$ to its operation that results in it being assigned a timestamp is a serialization function for s_k.

execution of $act(ser_k(G_i))$, then $\widehat{G}_q \in S_1$ and $\widehat{G}_r \in S_2$ just before $act(ser_k(G_i))$ executes. Further, $\widehat{G}_q \notin set_k$ after the execution of $act(ser_k(G_i))$, since $\widehat{G}_i$ is deleted from set_k when $act(ser_k(G_i))$ executes, and no transaction in $ser\_bef(\widehat{G}_i)$ is in set_k just before $act(ser_k(G_i))$ executes (since then, $cond(ser_k(G_i))$ would not hold). Thus, $set_p \neq set_k$. However, the addition to $cond(ser_k(G_i))$ for Scheme 3 ensures that if $set_p \neq set_k$, then $init_q$ is processed before $init_r$.

- $act(ack(ser_k(G_i)))$: For all transactions $\widehat{G}_j$, $ser\_bef(\widehat{G}_j)$ is not modified by $act(ack(ser_k(G_i)))$. Also set_p is not modified by $act(ack(ser_k(G_i)))$. Thus the execution of $act(ser_k(G_i))$ preserves the lemma.
- $act(fin_i)$: For all transactions $\widehat{G}_j$, execution of $act(fin_i)$ results in $\widehat{G}_i$ being deleted from $ser\_bef(\widehat{G}_j)$. We show that $act(fin_i)$ preserves the lemma. If $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ after $act(fin_i)$ executes, then $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ before $act(fin_i)$ executes, since no new elements are added to $ser\_bef(\widehat{G}_r)$ during the execution of $act(fin_i)$. Further, since set_p is not modified by $act(fin_i)$, if $\widehat{G}_q, \widehat{G}_r \in set_p$ after the execution of $act(fin_i)$, then $\widehat{G}_q, \widehat{G}_r \in set_p$ before the execution of $act(fin_i)$. Since the lemma holds before execution of $act(fin_i)$, $init_q$ is processed before $init_r$. □

Proof of Theorem 12: We need to show that if $act(init_i)$ executes before $act(init_j)$, then no operation $ser_k(G_i)$ belonging to a transaction $\widehat{G}_i$ is delayed due to transaction $\widehat{G}_j$. Let us suppose that operation $ser_k(G_i)$ is delayed due to transaction $\widehat{G}_j$, or alternatively $cond(ser_k(G_i))$ does not hold due to transaction $\widehat{G}_j$. As a result, there must be a transaction $\widehat{G}_j \in set_k$, such that $\widehat{G}_j \in ser\_bef(\widehat{G}_i)$. By Lemma 13, $init_j$ is processed before $init_i$ is processed, which leads to a contradiction. □

Proof of Theorem 13: The sets set_k and $ser\_bef(\widehat{G}_i)$ are implemented as mentioned earlier in the complexity analysis for Scheme 3. Since there is an addition to $cond(ser_k(G_i))$, we first analyze the number of steps in $cond(ser_k(G_i))$. The number of steps in $cond(ser_k(G_i))$ in Scheme 3 without the addition is $O(n)$ ($cond(ser_k(G_i))$ requires the intersection of two sets of size $O(n)$ to be computed that in the worst case takes $O(n)$ steps). The addition results in the following additional steps. Sets S_1 and S_2 first need to be computed, where $S_1 = \{\widehat{G}_i\} \cup ser\_bef(\widehat{G}_i)$, and $S_2 = \{\widehat{G}_j : \widehat{G}_j \in (set_k - \widehat{G}_i) \vee (ser\_bef(\widehat{G}_j) \cap (set_k - \widehat{G}_i) \neq \emptyset)\}$. Computation of S_2 takes $O(n^2)$ steps (since for every transaction $\widehat{G}_j$, computation of $(ser\_bef(\widehat{G}_j) \cap set_k)$ takes $O(n)$ steps and there are at most n transactions).

Also, for every set set_p, the following are computed: $S'_p = S_1 \cap set_p$ and $S''_p = S_2 \cap set_p$. Since transactions in S'_p and S''_p are ordered in the order in which their $init_j$ operations are processed, $cond(ser_k(G_i))$ holds iff for every set set_p, the $init_j$ operation for the last transaction in S'_p is processed before the $init_j$ operation for the first transaction in S''_p is processed. The computation of S'_p and S''_p, for every set set_p, takes $O(n)$ steps (intersection of two sets of size $O(n)$). Thus, since there are m such sets, the number of steps in $cond(ser_k(G_i))$ is $O(mn + n^2)$. The number of steps in $act(o_j)$ and $cond(o_j)$, for the remaining operations, are as mentioned earlier in the complexity analysis for Scheme 3.

We now specify $wait(o_j)$ for each operation o_j. The sets $wait(init_i)$, $wait(ser_k(G_i)$ and $wait(fin_i)$ are as mentioned in the complexity analysis for Scheme 3. Further, since the execution of $act(ack(ser_k(G_i)))$ can result in $cond(ser_l(G_j))$ for any of the $ser_l(G_j)$ operations in WAIT to hold, $wait(ack(ser_k(G_i))) = \{ser_l(G_j) : ser_l(G_j) \in \text{WAIT}\}$. Thus, $wait(ack(ser_l(G_j)))$ has $O(nd_{av})$ operations in the worst case since there are at most n transactions with operations in WAIT, and each transaction has d_{av} operations. Thus, since the number of steps in $cond(ser_k(G_i))$ is $O(mn + n^2)$, the number of steps required to process $ack(ser_k(G_i))$ is $O((mn^2 + n^3)d_{av})$. The complexity of Scheme 3 is dominated by the number of steps required to process all the $ack(ser_k(G_i)$ operations belonging to a transaction. Since there are d_{av} operations per transaction, the complexity of Scheme 3 is $O((mn^2 + n^3)d_{av}^2)$. □

  - if $last_p = \widehat{G}_l$, then $act(ack(ser_p(G_l)))$ has not completed execution, or
  - for some transaction $\widehat{G}_l \in set_p$, $\widehat{G}_l \in ser\_bef(\widehat{G}_j)$.

  However, since execution of $act(ack(ser_k(G_i)))$ does not result in a modification of set_p, $wait(ack(ser_k(G_i)))$ is restricted to operations $ser_k(G_l)$, for transactions $\widehat{G}_l \in set_k$.

- $wait(fin_i)$: $\{fin_j : fin_j \in \text{WAIT}\}$.
  For any operation $ser_k(G_j) \in \text{WAIT}$, $cond(ser_k(G_j))$ cannot hold due to the execution of $act(fin_i)$ since $act(fin_i)$ only deletes transactions from $ser\_bef(\widehat{G}_l)$, for transactions $\widehat{G}_l$ such that $\widehat{G}_i \in ser\_bef(\widehat{G}_l)$.

Thus, the number of steps in $cond(o_l)$, for any operation $o_l \in wait(ack(ser_k(G_i)))$ is $O(n)$, and the number of steps in $cond(o_l)$ for any operation $o_l \in wait(fin_i)$, is $O(1)$. Further, in the worst case, the number of operations in both $wait(ack(ser_k(G_i)))$ and $wait(fin_i)$ is $O(n)$ (since size of set_k is $O(n)$, and the number of fin_j operations in WAIT never exceeds n).

Proof of Theorem 9: The complexity of Scheme 3 is dominated by the number of steps in $act(ser_k(G_i))$, which is $O(n^2)$. Thus, since each transaction has d_{av} operations, the complexity of Scheme 3 is $O(n^2 d_{av})$. □

Before we show that Scheme 3 with the addition is starvation-free, we prove the following lemma.

Lemma 13: At any point during the execution of Scheme 3 with the addition, the following holds:

- For every set set_p, for all pairs of transactions $\widehat{G}_q, \widehat{G}_r \in set_p$, if $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$, then $init_q$ is processed before $init_r$.

Proof: Trivially, the lemma holds initially since for every set set_p, $set_p = \emptyset$. In addition, we show that for all operations, o_j, $act(o_j)$ preserves the lemma.

- $act(init_i)$: Elements are added only to $ser\_bef(\widehat{G}_i)$ and only $\widehat{G}_i$ is added to sets set_p such that $s_p \in exec(G_i)$. Also, before the execution of $act(init_i)$, $ser\_bef(\widehat{G}_i) = \emptyset$, for all sets set_p, $\widehat{G}_i \notin set_p$, and for all $\widehat{G}_j$, $\widehat{G}_i \notin ser\_bef(\widehat{G}_j)$. If $\widehat{G}_r \neq \widehat{G}_i$, then since $ser\_bef(\widehat{G}_r)$ is not modified by $act(init_i)$, $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ before $act(init_i)$ executes. Also, $\widehat{G}_q \neq \widehat{G}_i$, since for all transactions $\widehat{G}_j$, $\widehat{G}_i \notin ser\_bef(\widehat{G}_j)$ before $act(init_i)$ executes. As a result, if $\widehat{G}_q, \widehat{G}_r \in set_p$ after $act(init_i)$ executes, then $\widehat{G}_q, \widehat{G}_r \in set_p$ before $act(init_i)$ executes since only $\widehat{G}_i$ is added to set_p, and $\widehat{G}_q \neq \widehat{G}_i$, $\widehat{G}_r \neq \widehat{G}_i$. Thus, since the lemma holds before $act(init_i)$ executes, $init_q$ is processed before $init_r$.

  If $\widehat{G}_r = \widehat{G}_i$, then for all transactions $\widehat{G}_j$ that are added to $ser\_bef(\widehat{G}_i)$ when $act(init_i)$ executes, $\widehat{G}_j$ is either $last_k$ or in $ser\_bef(last_k)$ for some site $s_k \in exec(G_i)$ just before $act(init_i)$ executes. Since $init_j$ is processed before a transaction $\widehat{G}_j$ is added to $ser\_bef(\widehat{G}_l)$, for some transaction $\widehat{G}_l$, for all transactions $\widehat{G}_j$ that are added to $ser\_bef(\widehat{G}_i)$ when $act(init_i)$ executes, $init_j$ has already been processed. Thus, $init_q$ is processed before $init_r$.

- $act(ser_k(G_i))$: Execution of $act(ser_k(G_i))$ does not result in any transactions being added to set_p. Thus, if $\widehat{G}_q, \widehat{G}_r \in set_p$ after execution of $act(ser_k(G_i))$, then $\widehat{G}_q, \widehat{G}_r \in set_p$ before execution of $act(ser_k(G_i))$.

  If $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ before execution of $act(ser_k(G_i))$, then since the lemma holds before execution of $act(ser_k(G_i))$, $init_q$ is processed before $init_r$.

  Let $S_1 = (\{\widehat{G}_i\} \cup ser\_bef(\widehat{G}_i))$ and $S_2 = \{\widehat{G}_l : (\widehat{G}_l \in (set_k - \widehat{G}_i)) \vee (ser\_bef(\widehat{G}_l) \cap (set_k - \widehat{G}_i) \neq \emptyset)\}$ just before $act(ser_k(G_i))$ executes. If $\widehat{G}_q \notin ser\_bef(\widehat{G}_r)$ before the

- Does $d \in S_1$? – $O(n)$.

Since the number of transactions $\widehat{G}_i$ such that $init_i$, but not fin_i, has been processed by Scheme 3, never exceeds n, the sizes of set_k and $ser\_bef(\widehat{G}_i)$ are $O(n)$.

The number of steps in $cond(o_j)$ and $act(o_j)$, for each operation o_j, are as follows.

- $cond(init_i)$: $O(1)$.
- $act(init_i)$: $O(nd_{av})$. In the worst case, $act(init_i)$ requires the union of d_{av} sets of size $O(n)$ to be computed and then assigned to $ser\_bef(\widehat{G}_i)$.
- $cond(ser_k(G_i))$: $O(n)$. $cond(ser_k(G_i))$ requires the intersection of two sets of size $O(n)$ to be computed that in the worst case takes $O(n)$ steps.
- $act(ser_k(G_i))$: $O(n^2)$. The cost of $act(ser_k(G_i))$ is dominated by the cost of updating $ser\_bef(\widehat{G}_j)$ for transactions $\widehat{G}_j$. For each transaction $\widehat{G}_j$ (such that $init_j$ has been processed, but fin_j has not been processed), first, checking if the condition $\widehat{G}_j \in set_k$ is true takes $O(n)$ steps, or if the condition $ser\_bef(\widehat{G}_j) \cap set_k \neq \emptyset$ takes $O(n)$ steps. Finally, if the condition is true, then the union of two sets of size $O(n)$ needs to be computed, that takes $O(n)$ steps. Since there are n such transactions in the worst case, the number of steps in $act(ser_k(G_i))$ is $O(n^2)$ in the worst case.
- $cond(ack(ser_k(G_i)))$: $O(1)$.
- $act(ack(ser_k(G_i)))$: $O(1)$.
- $cond(fin_i)$: $O(1)$.
- $act(fin_i)$: $O(n^2)$. For each transaction $\widehat{G}_j$, a check is made to determine if $\widehat{G}_i \in ser\_bef(\widehat{G}_j)$ that takes $O(n)$ steps per transaction. Further, if $\widehat{G}_i \in ser\_bef(\widehat{G}_j)$, then $\widehat{G}_i$ is deleted from $ser\_bef(\widehat{G}_j)$, that takes $O(n)$ steps (since size of $ser\_bef(\widehat{G}_j)$, in the worst case is $O(n)$). Since there are n transactions in the worst case, the number of steps in $act(fin_i)$ is $O(n^2)$.

Since $cond(init_i)$ and $cond(ack(ser_k(G_i)))$ are both *true*, the only operations in WAIT are either $ser_k(G_i)$ for some transaction $\widehat{G}_i$ and site $s_k \in exec(G_i)$, or fin_i for some transaction $\widehat{G}_i$. Also, execution of $act(o_j)$, for an operation o_j, can cause $cond(ser_k(G_i))$ to hold only if either execution of $act(o_j)$ results in the deletion of a transaction from set_k, or $o_j = ack(ser_k(G_l))$ for some transaction $\widehat{G}_l$. In addition, execution of $act(o_j)$, for some operation o_j, can cause $cond(fin_i)$ for some transaction $\widehat{G}_i$ to hold only if $act(o_j)$ deletes a transaction from $ser\_bef(\widehat{G}_i)$.

We now specify $wait(o_j)$ for each of the operations o_j.

- $wait(init_i)$: $\emptyset$. Execution of $act(init_i)$ does not result in transactions being deleted from any of the sets.
- $wait(ser_k(G_i))$: $\emptyset$. Even though $act(ser_k(G_i))$ results in the deletion of $\widehat{G}_i$ from set_k, for any operation $ser_k(G_l)$, $cond(ser_k(G_l))$ does not hold due to the execution of $act(ser_k(G_i))$ unless $act(ack(ser_k(G_i)))$ completes execution. Also, for an operation $ser_p(G_l)$, $s_p \neq s_k$, $cond(ser_p(G_l))$ cannot hold due to the execution of $act(ser_k(G_i))$ since $act(ser_k(G_i))$ deletes elments only from set_k and $set_k \neq set_p$. Further, execution of $act(ser_k(G_i))$ cannot cause $cond(fin_i)$ to hold since $act(ser_k(G_i))$ does not delete transactions from $ser\_bef(\widehat{G}_i)$.
- $wait(ack(ser_k(G_i)))$: $\{ser_k(G_j) : ser_k(G_j) \in (\text{WAIT} \cap set_k)\}$.
  For any operation $fin_j \in$ WAIT, execution of $act(ack(ser_k(G_i)))$ cannot cause $cond(fin_j)$ to hold since no elements are deleted from $ser\_bef(\widehat{G}_j)$ as a result of the execution of $act(ack(ser_k(G_i)))$.

  Further, execution of $act(ack(ser_k(G_i)))$ cannot cause $cond(ser_p(G_j))$ to hold for some operation $ser_p(G_j) \in$ WAIT, $s_p \neq s_k$, since if $cond(ser_p(G_j))$ did not hold prior to the execution of $act(ack(ser_k(G_i)))$, then either the transitive property holds, $\widehat{G}_{i_1} \in ser\_bef(\widehat{G}_{i_1})$. However, this leads to a contradiction since by Lemma 10**c**, $\widehat{G}_j \notin ser\_bef(\widehat{G}_j)$ for all transactions $\widehat{G}_j$ at all points during the execution of Scheme 3. $\Box$

Proof of Corollary 1: By Theorem 10, the total number of unprocessed operations decreases during the execution of Scheme 3 (since at any point during the execution of Scheme 3, it is possible to process an operation). Since every transaction has a finite number of operations and a finite number of transactions are initiated, the number of unprocessed operations eventually reduces to zero, that is, every transaction completes execution. $\Box$

Proof of Theorem 11: Let $ser_p(G_q)$ operations be inserted into QUEUE in a serializable order. We use induction to prove that for all $l \geq 0$, the first l $ser_p(G_q)$ operations inserted into QUEUE are processed by Scheme 3 when they are selected from QUEUE.
Basis $(l = 0)$**:** Trivial.
Induction: Assume that the first r $ser_p(G_q)$ operations inserted into QUEUE are processed when they are selected from QUEUE by Scheme 3. We need to show that the first $r+1$ $ser_p(G_q)$ operations are processed when they are selected from QUEUE by Scheme 3. Let $ser_k(G_i)$, for some transaction $\widehat{G}_i$ and $s_k \in exec(G_i)$ be the $r+1^{th}$ $ser_p(G_q)$ operation inserted into QUEUE. By the induction hypothesis, the first r $ser_p(G_q)$ operations inserted into QUEUE are processed by Scheme 3 when they are selected from QUEUE. We need to show that $ser_k(G_i)$ is processed by Scheme 3, or alternatively $cond(ser_k(G_i))$ holds, after the first r $ser_p(G_q)$ operations inserted into QUEUE have been processed, and $ser_k(G_i)$ is selected from QUEUE. Thus, we need to show that, when $ser_k(G_i)$ is selected from QUEUE, for all $\widehat{G}_l \in (set_k - \widehat{G}_i)$, $\widehat{G}_l \notin ser\_bef(\widehat{G}_i)$.

Suppose for some $\widehat{G}_l \in (set_k - \widehat{G}_i)$, $\widehat{G}_l \in ser\_bef(\widehat{G}_i)$. Thus, by Lemma 10**a**, $\widehat{G}_l$ is serialized before $\widehat{G}_i$ in $ser(S)$. Since the first r $ser_p(G_q)$ operations are processed by Scheme 3 when they are selected from QUEUE, if every $ser_p(G_q)$ operation in QUEUE is processed when it is selected from QUEUE, then $\widehat{G}_l$ would be serialized before $\widehat{G}_i$ in the resulting schedule. Further, since $\widehat{G}_l \in (set_k - \widehat{G}_i)$ when $ser_k(G_i)$ is selected from QUEUE, $ser_k(G_l)$ must have been inserted into QUEUE by GTM$_1$ after $ser_k(G_i)$ is inserted. Thus, if $ser_p(G_q)$ operations are processed when they are selected from QUEUE, then $ser_k(G_l)$ would be processed after $ser_k(G_i)$, and as a result, $\widehat{G}_i$ would be serialized before $\widehat{G}_l$ in the resulting schedule. However, this leads to a contradiction since operations are inserted into QUEUE by GTM$_1$ in a serializable order. Thus, for all $\widehat{G}_l \in (set_k - \widehat{G}_i)$, $\widehat{G}_l \notin ser\_bef(\widehat{G}_i)$ when $ser_k(G_i)$ is selected from QUEUE. As a result, since $ser\_bef(\widehat{G}_i) \cap (set_k - \widehat{G}_i) = \emptyset$, $cond(ser_k(G_i))$ holds and $ser_k(G_i)$ is processed by Scheme 3.
$\Box$

Complexity Analysis of Scheme 3:

We first describe additional data structures involved in the implementation of Scheme 3. We then analyze, for every operation o_j, the number of steps in $cond(o_j)$ and $act(o_j)$, and the characteristics of $wait(o_j)$ (the number of operations and their types).

Implementation: Every transaction $\widehat{G}_i$, when $init_i$ executes, is assigned a unique identifier (that increases with time) that defines a total order on the set of transactions. The sets of transactions set_k and $ser\_bef(\widehat{G}_i)$ are implemented as lists in which the transactions are stored in an increasing order of their identifiers. If S_1 and S_2 are two sets of size $O(n)$ that are implemented as lists of elements stored in an increasing order, and d is an element, then the complexity of various operations are as follows.

- $S_1 \cup S_2$ - $O(n)$.
- $S_1 \cap S_2$ - $O(n)$.
- $S_1 - \{d\}$ - $O(n)$.

- $act(fin_i)$ has not yet executed when $\widehat{G}_j$ is added to set_k due to the execution of $act(init_j)$,

$last_k = \widehat{G}_i$ when $act(init_j)$ executes. $\widehat{G}_i$ is thus added to $ser\_bef(\widehat{G}_j)$ when $act(init_j)$ executes. Further, since $\widehat{G}_i$ is deleted from $ser\_bef(\widehat{G}_j)$ only when $act(fin_i)$ executes, $act(ser_k(G_j))$ executes after both $act(ser_k(G_i))$ and $act(init_j)$ execute, and $act(fin_i)$ has not yet executed at p, $\widehat{G}_i \in ser\_bef(\widehat{G}_j)$ at p.

Induction: Assume the lemma is true for $num \leq r$, $r \geq 0$. We need to show that the lemma holds if the number of transactions $\widehat{G}_l$ such that $act(ser_k(G_l))$ executes in between $act(ser_k(G_i))$ and $act(ser_k(G_j))$ is $\leq r+1$. Let $\widehat{G}_q$ be a transaction such that $act(ser_k(G_q))$ executes in between $act(ser_k(G_i))$ and $act(ser_k(G_j))$. The number of transactions $\widehat{G}_l$ such that $act(ser_k(G_l))$ executes in between $act(ser_k(G_i))$ and $act(ser_k(G_q))$ is $\leq r$. Similarly, the number of transactions $\widehat{G}_l$ such that $act(ser_k(G_l))$ executes in between $act(ser_k(G_q))$ and $act(ser_k(G_j))$ is $\leq r$.

Since $act(fin_i)$ has not yet executed at p, and $act(ser_k(G_q))$ executes before p (since $act(ser_k(G_q))$ executes before $act(ser_k(G_j))$), $act(fin_i)$ has not yet executed when $act(ser_k(G_q))$ executes. Thus, by the induction hypothesis, $\widehat{G}_i \in ser\_bef(\widehat{G}_q)$ at any point after $act(ser_k(G_q))$ executes and before $act(fin_i)$ executes. Since $\widehat{G}_i \in ser\_bef(\widehat{G}_q)$ until $act(fin_i)$ executes, $cond(fin_q)$ does not hold unless $act(fin_i)$ executes. Thus, $act(fin_i)$ executes before $act(fin_q)$ executes, and at p $act(fin_q)$ has not yet executed. As a result, again, by the induction hypothesis, since $act(ser_k(G_q))$ executes before $act(ser_k(G_j))$ executes, $\widehat{G}_q \in ser\_bef(\widehat{G}_j)$ at any point after $act(ser_k(G_j))$ executes and before $act(fin_q)$ executes. Thus, at p, since $act(fin_i)$ and $act(fin_q)$ have not yet executed, $\widehat{G}_i \in ser\_bef(\widehat{G}_q)$ and $\widehat{G}_q \in ser\_bef(\widehat{G}_j)$. As a result, by Lemma10**b**, since the transitive property holds at all points during the execution of Scheme 3, $\widehat{G}_i \in ser\_bef(\widehat{G}_j)$ at p. □

Proof of Theorem 8: Suppose $ser(S)$ is not serializable. Thus, there exist distinct transactions, say, $\widehat{G}_1, \widehat{G}_2, \ldots, \widehat{G}_r$, $r > 1$, such that $ser_{i_1}(G_1)$ executes before $ser_{i_1}(G_2)$, $ser_{i_2}(G_2)$ executes before $ser_{i_2}(G_3)$, …, $ser_{i_r}(G_r)$ executes before $ser_{i_r}(G_1)$, and for all $j, k = 1, 2, \ldots, r$, $j \neq k$, $i_j \neq i_k$ (since for any site s_k, transaction $\widehat{G}_j$ has at most one operation $ser_k(G_j)$).

We claim that for all j, $j = 1, 2, \ldots, r$, none of $act(fin_j)$ can execute. To see this, observe that for all j, $j = 1, 2, \ldots, r$, $ser_{i_j}(G_j)$ executes before $ser_{i_j}(G_{(j \bmod r)+1})$. Thus, by Lemma 11 and Lemma 12, if $act(fin_j)$ has not executed when Scheme 3 attempts to execute $act(fin_{(j \bmod r)+1})$, $\widehat{G}_j \in ser\_bef(\widehat{G}_{(j \bmod r)+1})$ and thus, $ser\_bef(\widehat{G}_{(j \bmod r)+1}) \neq \emptyset$. (since Scheme 3 attempts to execute $act(fin_{(j \bmod r)+1})$ after it executes $act(ser_{i_j}(G_{(j \bmod r)+1}))$). As a result, since $cond(fin_{(j \bmod r)+1})$ does not hold unless $ser\_bef(\widehat{G}_{(j \bmod r)+1}) = \emptyset$, $act(fin_{(j \bmod r)+1})$ cannot execute unless $act(fin_j)$ has executed. Thus, $act(fin_j)$ must execute before $act(fin_{(j \bmod r)+1})$ executes. Now suppose $act(fin_k)$ executes for some $k = 1, 2, \ldots, r$. From the above arguments, if follows that $act(fin_k)$ executes before $act(fin_k)$ executes, which is not possible. Thus, none of $act(fin_j)$ can execute, for all j, $j = 1, 2, \ldots, r$.

Consider a point p during the execution of Scheme 3 when all of $act(ser_{i_j}(G_j))$, $act(ser_{i_j}(G_{(j \bmod r)+1}))$, $j = 1, 2, \ldots, r$ have been executed. Since for all j, $j = 1, 2, \ldots, r$, $act(fin_j)$ does not execute, by Lemma 12, $\widehat{G}_j \in ser\_bef(\widehat{G}_{(j \bmod r)+1})$ at p. By Lemma 10**b**, since the transitive property holds, $\widehat{G}_1 \in ser\_bef(\widehat{G}_1)$ at p. However, this leads to a contradiction since by Lemma 10**c**, $\widehat{G}_j \notin ser\_bef(\widehat{G}_j)$, for all $\widehat{G}_j$ and at all points during the execution of Scheme 3. Thus, $ser(S)$ is serializable. □

Proof of Theorem 10: Suppose that for all $\widehat{G}_p \in set_k$, $act(ser_k(G_p))$ cannot be executed. Thus, for every $\widehat{G}_p \in set_k$, there exists a $\widehat{G}_q \in set_k$ such that $\widehat{G}_q \in ser\_bef(\widehat{G}_p)$. Since set_k has a finite number of elements, there must be transactions in set_k $\widehat{G}_{i_1}, \widehat{G}_{i_2}, \ldots, \widehat{G}_{i_r}$ such that $\widehat{G}_{i_1} \in ser\_bef(\widehat{G}_{i_2})$, $\widehat{G}_{i_2} \in ser\_bef(\widehat{G}_{i_3})$, …, $\widehat{G}_{i_r} \in ser\_bef(\widehat{G}_{i_1})$. Thus, by Lemma 10**b**, since if $\widehat{G}_p \in ser\_bef(\widehat{G}_p)$ after the execution of $act(ser_k(G_i))$, then just before $act(ser_k(G_i))$ executes, $\widehat{G}_p \in (\{\widehat{G}_i\} \cup ser\_bef(\widehat{G}_i))$ and $\widehat{G}_p \in (\{\widehat{G}_j\} \cup \{\widehat{G}_l : \widehat{G}_j \in ser\_bef(\widehat{G}_l)\}$ for some $\widehat{G}_j \in (set_k - \widehat{G}_i)$ just before $act(ser_k(G_i))$ executes. We consider the following cases just before $act(ser_k(G_i))$ executes.

1. $\widehat{G}_p = \widehat{G}_i$ and $\widehat{G}_p = \widehat{G}_j$: This is not possible since $\widehat{G}_j \in (set_k - \widehat{G}_i)$ and thus, $\widehat{G}_j \neq \widehat{G}_i$.
2. $\widehat{G}_p = \widehat{G}_i$ and $\widehat{G}_p \neq \widehat{G}_j$: Thus, since $\widehat{G}_j \in ser\_bef(\widehat{G}_p)$, $\widehat{G}_j \in ser\_bef(\widehat{G}_i)$ and thus $ser\_bef(\widehat{G}_i) \cap (set_k - \widehat{G}_i) \neq \emptyset$ just before $act(ser_k(G_i))$ executes. As a result, $cond(ser_k(G_i))$ does not hold, and thus $act(ser_k(G_i))$ cannot be executed.
3. $\widehat{G}_p \neq \widehat{G}_i$ and $\widehat{G}_p = \widehat{G}_j$: In this case $\widehat{G}_p \in ser\_bef(\widehat{G}_i)$ and thus $\widehat{G}_j \in ser\_bef(\widehat{G}_i)$ just before $act(ser_k(G_i))$ executes. For reasons similar to above, $act(ser_k(G_i))$ cannot be executed.
4. $\widehat{G}_p \neq \widehat{G}_i$ and $\widehat{G}_p \neq \widehat{G}_j$: As a result, $\widehat{G}_p \in ser\_bef(\widehat{G}_i)$ and $\widehat{G}_j \in ser\_bef(\widehat{G}_p)$ just before $act(ser_k(G_i))$ executes. Thus, since the transitive property holds before $act(ser_k(G_i))$ executes, $\widehat{G}_j \in ser\_bef(\widehat{G}_i)$ just before $act(ser_k(G_i))$ executes, and $act(ser_k(G_i))$ cannot execute.

- $act(ack(ser_k(G_i)))$: For all transactions $\widehat{G}_j$, $ser\_bef(\widehat{G}_j)$ is not modified by $act(ack(ser_k(G_i)))$. Thus **a**, **b** and **c** are preserved.
- $act(fin_i)$: For all transactions $\widehat{G}_j$, execution of $act(fin_i)$ results in $\widehat{G}_i$ being deleted from $ser\_bef(\widehat{G}_j)$. We show that $act(fin_i)$ preserves a, b and c.

  a: If $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ after $act(fin_i)$ executes, then $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ before $act(fin_i)$ executes, since no new elements are added to $ser\_bef(\widehat{G}_q)$ during the execution of $act(fin_i)$. Since **a** holds before execution of $act(fin_i)$, $\widehat{G}_p$ is serialized before $\widehat{G}_q$ in $ser(S)$.

  b: Since for all transactions $\widehat{G}_j$, execution of $act(fin_i)$ results in $\widehat{G}_i$ being deleted from $ser\_bef(\widehat{G}_j)$, if $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ after execution of $act(fin_i)$, then $\widehat{G}_p \neq \widehat{G}_i$, and $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$, $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ before $act(fin_i)$ executes. As a result, since **b** holds before $act(fin_i)$ executes, $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$ before $act(fin_i)$ executes. Further, since $\widehat{G}_p \neq \widehat{G}_i$, $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$ after $act(fin_i)$ executes.

  c: Since **c** holds before execution of $act(fin_i)$ and no new elements are added to $ser\_bef(\widehat{G}_p)$ when $act(fin_i)$ executes, $\widehat{G}_p \notin ser\_bef(\widehat{G}_p)$ after the execution of $act(fin_i)$. □

Lemma 11: For all sites s_k, transactions $\widehat{G}_i, \widehat{G}_j$, if $ser_k(G_i)$ executes before $ser_k(G_j)$, then $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$.

Proof: Let us assume that $act(ser_k(G_j))$ executes before $act(ser_k(G_i))$. Since before $act(ack(ser_k(G_j)))$ executes and after $act(ser_k(G_j))$ executes, $last_k = \widehat{G}_j$, $act(ser_k(G_i))$ cannot execute before $act(ack(ser_k(G_j)))$ executes. As a result, $ser_k(G_j)$ executes before $ser_k(G_i)$ which leads to a contradiction. Thus, $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$. □

Lemma 12: For all sites s_k, transactions $\widehat{G}_i, \widehat{G}_j$, if $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$ executes, then at any point p during the execution of Scheme 3 after the execution of $act(ser_k(G_j))$, but before the execution of $act(fin_i)$, the following is true: $\widehat{G}_i \in ser\_bef(\widehat{G}_j)$.

Proof: We prove the lemma by induction on num, the number of transactions $\widehat{G}_l$ such that $act(ser_k(G_l))$ executes in between $act(ser_k(G_i))$ and $act(ser_k(G_j))$.

Basis ($num = 0$): If $\widehat{G}_j \in set_k$ when $act(ser_k(G_i))$ executes, then $\widehat{G}_i$ is added to $ser\_bef(\widehat{G}_j)$ when $act(ser_k(G_i))$ executes. If $\widehat{G}_j \notin set_k$ when $act(ser_k(G_i))$ executes, then since

- $last_k$ is set to $\widehat{G}_i$ when $act(ser_k(G_i))$ executes,
- for all transactions $\widehat{G}_l$, $act(ser_k(G_l))$ does not execute in between $act(ser_k(G_i))$ and $act(ser_k(G_j))$, and

- $act(init_i)$: Elements are added only to $ser\_bef(\widehat{G}_i)$. Also, before $init_i$ is processed, $ser\_bef(\widehat{G}_i) = \emptyset$ and $\widehat{G}_i \notin ser\_bef(\widehat{G}_j)$, for all $\widehat{G}_j$.

  a: If $\widehat{G}_q \neq \widehat{G}_i$, then since $ser\_bef(\widehat{G}_q)$ is not modified by $act(init_i)$, $\widehat{G}_p \in \widehat{G}_q$ before $act(init_i)$ executes. Thus, since **a** holds before $act(init_i)$ executes, $\widehat{G}_p$ is serialized before $\widehat{G}_q$ in $ser(S)$.

  If $\widehat{G}_q = \widehat{G}_i$, then just before $act(init_i)$ executes, either $\widehat{G}_p = last_k$ or $\widehat{G}_p \in ser\_bef(last_k)$ for some $s_k \in exec(G_i)$. Since **a** holds before $act(init_i)$ executes, transactions in $ser\_bef(last_k)$ are serialized before $last_k$ in $ser(S)$. Since $act(ser_k(G_i))$ executes after the acknowledgement of the completion of $last_k$'s operation, $last_k$ is serialized before $\widehat{G}_i$ in $ser(S)$. By transitivity of the serialized before relationship, transactions in $ser\_bef(last_k)$ are serialized before $\widehat{G}_i$ in $ser(S)$. Thus, $\widehat{G}_p$ is serialized before $\widehat{G}_i$ in $ser(S)$.

  b: We now use Lemma 9 to show that **b** is preserved. Just before $act(init_i)$ executes, let $S_1 = ser\_bef(last_k) \cup \{last_k\}$, for some site $s_k \in exec(G_i)$, and let $S_2 = \{\widehat{G}_l : \widehat{G}_i \in ser\_bef(\widehat{G}_l)\} \cup \{\widehat{G}_i\}$. Since, before $act(init_i)$ executes, $\widehat{G}_i \notin ser\_bef(\widehat{G}_j)$, for all $\widehat{G}_j$, $S_2 = \{\widehat{G}_i\}$. Thus, by Lemma 9, executing $ser\_bef(\widehat{G}_i) := ser\_bef(\widehat{G}_i) \cup S_1$ preserves **b**.

  c: If $\widehat{G}_p \neq \widehat{G}_i$, then since $ser\_bef(\widehat{G}_p)$ is not modified by $act(init_i)$, and since **c** holds before $act(init_i)$ executes, $\widehat{G}_p \notin ser\_bef(\widehat{G}_p)$ after execution of $act(init_i)$.

  If $\widehat{G}_p = \widehat{G}_i$, then for all $s_k \in exec(G_i)$, before $act(init_i)$ executes, $last_k \neq \widehat{G}_i$ since $act(ser_k(G_i))$ has not yet executed. Also, before $act(init_i)$ executes, since $\widehat{G}_i \notin ser\_bef(\widehat{G}_j)$ for all $\widehat{G}_j$, executing $ser\_bef(\widehat{G}_i) := ser\_bef(last_k) \cup \{last_k\}$ cannot result in $\widehat{G}_i \in ser\_bef(\widehat{G}_i)$.

- $act(ser_k(G_i))$:

  a: If $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ before execution of $act(ser_k(G_i))$, then since **a** holds before execution of $act(ser_k(G_i))$, $\widehat{G}_p$ is serialized before $\widehat{G}_q$ in $ser(S)$.

  Just before $act(ser_k(G_i))$ executes, let $S_1 = (\{\widehat{G}_i\} \cup ser\_bef(\widehat{G}_i))$ and $S_2 = \{\widehat{G}_l : (\widehat{G}_l \in (set_k - \widehat{G}_i)) \vee (ser\_bef(\widehat{G}_l) \cap (set_k - \widehat{G}_i) \neq \emptyset)\}$. If $\widehat{G}_p \notin ser\_bef(\widehat{G}_q)$ before the execution of $act(ser_k(G_i))$, then just before $act(ser_k(G_i))$ executes, $\widehat{G}_p \in S_1$ and $\widehat{G}_q \in S_2$. We show that every transaction in S_1 is serialized before every transaction in S_2 in $ser(S)$, and thus $\widehat{G}_p$ is serialized before $\widehat{G}_q$ in $ser(S)$. Since for all $\widehat{G}_l \in (set_k - \widehat{G}_i)$ just before $act(ser_k(G_i))$ executes, $act(ser_k(G_l))$ has not executed, $act(ser_k(G_i))$ executes before $act(ser_k(G_l))$ and thus $ser_k(G_i)$ executes before $ser_k(G_l)$ executes. As a result, $\widehat{G}_i$ is serialized before $\widehat{G}_l$ in $ser(S)$. Since **a** holds before the execution of $act(ser_k(G_i))$, every transaction in $ser\_bef(\widehat{G}_i)$, just before the execution of $act(ser_k(G_i))$, is serialized before $\widehat{G}_i$ in $ser(S)$. By the transitivity of the serialized before relation, for all $\widehat{G}_l \in (set_k - \widehat{G}_i)$ since $\widehat{G}_i$ is serialized before $\widehat{G}_l$, every transaction in S_1 is serialized before $\widehat{G}_l$ in $ser(S)$. Also, if for some transaction $\widehat{G}_j$, $ser\_bef(\widehat{G}_j) \cap (set_k - \widehat{G}_i) \neq \emptyset$ just before $act(ser_k(G_i))$ executes, then there exists a transaction $\widehat{G}_l \in (set_k - \widehat{G}_i)$ such that $\widehat{G}_l \in ser\_bef(\widehat{G}_j)$ just before $act(ser_k(G_i))$ executes. Since **a** holds before the execution of $act(ser_k(G_i))$, $\widehat{G}_l$ is serialized before $\widehat{G}_j$ in $ser(S)$. Thus, by transitivity of the serialized before relation, since every transaction in S_1 is serialized before every transaction in $(set_k - \widehat{G}_i)$, every transaction in S_1 is serialized before every transaction in S_2 in $ser(S)$.

  b: Just before $act(ser_k(G_i))$ executes, let $S_1 = (\{\widehat{G}_i\} \cup ser\_bef(\widehat{G}_i))$ and $S_2 = \{\widehat{G}_j\} \cup \{\widehat{G}_l : \widehat{G}_j \in ser\_bef(\widehat{G}_l)\}$, where $\widehat{G}_j \in (set_k - \widehat{G}_i)$ just before $act(ser_k(G_i))$ executes. By Lemma 9, since executing $ser\_bef(\widehat{G}_l) := ser\_bef(\widehat{G}_l) \cup S_1$ for all $\widehat{G}_l \in S_2$ preserves **b**, execution of $act(ser_k(G_i))$ preserves **b**.

  c: We show that if for transaction $\widehat{G}_p$, execution of $act(ser_k(G_i))$ results in $\widehat{G}_p \in ser\_bef(\widehat{G}_p)$, then $cond(ser_k(G_i))$ could not have held just before $act(ser_k(G_i))$ executed, and thus $act(ser_k(G_i))$ could not have executed. Since **c** holds before execution of $act(ser_k(G_i))$,

-Appendix D-

In order to show that Scheme 3 ensures the serializability of $ser(S)$, we first need to prove properties of $ser\_bef(\widehat{G}_i)$ for all transactions $\widehat{G}_i$. At any point during the execution of Scheme 3, the *transitive property* is said to hold if for any transactions $\widehat{G}_p, \widehat{G}_q, \widehat{G}_r$ such that $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$, the following is true: $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$.

Lemma 9: The following action A preserves the transitive property.

$$A: \text{ For all } \widehat{G}_k \in S_2, ser\_bef(\widehat{G}_k) := ser\_bef(\widehat{G}_k) \cup S_1$$

where $S_1 = ser\_bef(\widehat{G}_i) \cup \{\widehat{G}_i\}$ and $S_2 = \{\widehat{G}_k : \widehat{G}_j \in ser\_bef(\widehat{G}_k)\} \cup \{\widehat{G}_j\}$ just before A executes, and $\widehat{G}_i, \widehat{G}_j$ are transactions.

Proof: We show that if the transitive property holds before A executes, then it holds after A executes. Thus, we show that, after A executes, for any $\widehat{G}_p, \widehat{G}_q, \widehat{G}_r$, if $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$, then $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$. We consider the following cases:

- $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ before A executes and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ before A executes: If $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ before A executes, then since the transitive property holds before A executes, $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$ after A executes.
- $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ before A executes and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ only after A executes: Thus, before A executes, $\widehat{G}_q \in S_1$ and $\widehat{G}_r \in S_2$. We show that before A executes, $\widehat{G}_p \in ser\_bef(\widehat{G}_i)$ and thus $\widehat{G}_p \in S_1$. Since $\widehat{G}_q \in S_1$ before A executes, either $\widehat{G}_q = \widehat{G}_i$ or $\widehat{G}_q \in ser\_bef(\widehat{G}_i)$ before A executes. If $\widehat{G}_q = \widehat{G}_i$, then trivially $\widehat{G}_p \in ser\_bef(\widehat{G}_i)$ before A executes. If $\widehat{G}_q \in ser\_bef(\widehat{G}_i)$, then since the transitive property holds before A executes, $\widehat{G}_p \in ser\_bef(\widehat{G}_i)$ before A executes. Thus, before A executes, since $\widehat{G}_p \in S_1$ and $\widehat{G}_r \in S_2$, $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$ after A executes.
- $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ only after A executes and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ before A executes: Thus, $\widehat{G}_p \in S_1$ and $\widehat{G}_q \in S_2$ before A executes. We show that before A executes, $\widehat{G}_j \in ser\_bef(\widehat{G}_r)$ and thus $\widehat{G}_r \in S_2$. Since $\widehat{G}_q \in S_2$, either $\widehat{G}_q = \widehat{G}_j$ or $\widehat{G}_j \in ser\_bef(\widehat{G}_q)$ before A executes. If $\widehat{G}_q = \widehat{G}_j$, then trivially $\widehat{G}_j \in ser\_bef(\widehat{G}_r)$ before A executes. Else if, $\widehat{G}_j \in ser\_bef(\widehat{G}_q)$, then since the transitive property holds before A executes, $\widehat{G}_j \in ser\_bef(\widehat{G}_r)$ before A executes. Thus, since before A executes, $\widehat{G}_p \in S_1$ and $\widehat{G}_r \in S_2$, $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$ after A executes.
- $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ only after A executes and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$ only after A executes: Thus, before A executes, $\widehat{G}_p \in S_1$ and $\widehat{G}_r \in S_2$. As a result, $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$ after A executes. □

Lemma 10: At any point during the execution of Scheme 3, for all transactions $\widehat{G}_p, \widehat{G}_q, \widehat{G}_r$, the following hold:

a: If $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$, then $\widehat{G}_p$ is serialized before $\widehat{G}_q$ in $ser(S)$.

b: If $\widehat{G}_p \in ser\_bef(\widehat{G}_q)$ and $\widehat{G}_q \in ser\_bef(\widehat{G}_r)$, then $\widehat{G}_p \in ser\_bef(\widehat{G}_r)$ (the transitive property).

c: $\widehat{G}_p \notin ser\_bef(\widehat{G}_p)$.

Proof: Trivially, **a**,**b** and **c** hold initially since for all $\widehat{G}_i$, $ser\_bef(\widehat{G}_i) = \emptyset$. In addition, we show that for all operations, o_j, $act(o_j)$ preserves **a**, **b** and **c** (since $ser\_bef(\widehat{G}_i)$ is only modified when $act(o_j)$ executes).

is not minimal with respect to $\widehat{G}_i$ and (V, E, D) first calls $S((V, E, D), \widehat{G}_i)$. If the set of dependencies Δ returned by S is non-empty, then the algorithm responds “yes” (since if $\Delta' = \emptyset$ is minimal with respect to $\widehat{G}_i$ and (V, E, D), then a non-empty Δ cannot be minimal with respect to $\widehat{G}_i$ and (V, E, D), and S would return $\emptyset$). If, on the other hand, the set of dependencies Δ returned by S is $\emptyset$, then the algorithm responds “no” (since $\Delta' = \emptyset$ is minimal with respect to (V, E, D) and $\widehat{G}_i$). $\Box$

- $(x_{i+1}, emp'_i), (emp'_i, x_i)$, if $|neg_i| = 0$.

This can be shown formally using an induction argument. We shall, however, resort to a less formal approach in our arguments. The path has to contain edges $(D_2, s_2), (s_1, x_{p+1})$. Furthermore, for any node x_{i+1} in the path, the only edges in a continuation of the path from x_{i+1} are edges

- $(x_{i+1}, pos'_i(|pos_i|)), (pos'_i(|pos_i|), P'_{i,|pos_i|})$, if $|pos_i| > 0$,
- $(x_{i+1}, emp_i), (emp_i, x_i)$, if $|pos_i| = 0$,

or edges

- $(x_{i+1}, neg'_i(|neg_i|)), (neg'_i(|neg_i|), N'_{i,|neg_i|})$, if $|neg_i| > 0$,
- $(x_{i+1}, emp'_i), (emp'_i, x_i)$, if $|neg_i| = 0$.

We show that if edges $(x_{i+1}, pos'_i(|pos_i|)), (pos'_i(|pos_i|), P'_{i,|pos_i|})$ are in the path, then all the edges $(x_{i+1}, pos'_i(|pos_i|)), (pos'_i(|pos_i|), P'_{i,|pos_i|}), \ldots, (pos_i(1), P_{i,1}), (P_{i,1}, emp_i), (emp_i, x_i)$ are also in the path (the argument for showing that if edges $(x_{i+1}, neg'_i(|neg_i|)), (neg'_i(|neg_i|), N'_{i,|neg_i|})$ are in the path, then all the edges $(x_{i+1}, neg'_i(|neg_i|)), (neg'_i(|neg_i|), N'_{i,|neg_i|}), \ldots, (neg_i(1), N_{i,1}), (N_{i,1}, emp'_i), (emp'_i, x_i)$ are also in the path is similar). Let us assume that for some $k = 1, 2, \ldots, |pos_i|$, edge $(pos'_i(k), P'_{i,k})$ is in the path. We show that the following edges are also in the path

- $(P'_{i,k}, pos_i(k)), (pos_i(k), P_{i,k}), (P_{i,k}, pos'_i(k-1)), (pos'_i(k-1), P'_{i,k-1})$, if $k > 1$,
- $(P'_{i,k}, pos_i(k)), (pos_i(k), P_{i,k}), (P_{i,k}, emp_i), (emp_i, x_i)$, if $k = 1$.

Due to dependencies $(P'_{i,k}, pos_i(k)) \rightarrow (pos_i(k), C_i)$ and $(P'_{i,k}, pos_i(k)) \rightarrow (pos_i(k), C_{i+1})$, the only choice of edges from $P'_{i,k}$ in the path is $(P'_{i,k}, pos_i(k)), (pos_i(k), P_{i,k})$. From $(P_{i,k})$, the only choice of edges is $(P_{i,k}, pos'_i(k-1)), (pos'_i(k-1), P'_{i,k-1})$, if $k > 1$, and $(P_{i,k}, emp_i), (emp_i, x_i)$, if $k = 1$.

Thus, the path must contain edges $(x_1, s_0), (s_0, C_{p+1})$. Further, we claim that for all $i = 1, 2, \ldots, p$, edges $(C_{i+1}, l_{i,j}), (l_{i,j}, C_i)$ are in the path, for some $j = 1, 2, 3$. This follows from the fact that there are dependencies $(C_r, l_{r,s}) \rightarrow (l_{r,s}, C_{r+1})$, for all $r = 1, 2, \ldots, p$, for all $s = 1, 2, 3$. Also, edges $(C_{i+1}, l_{i,j}), (l_{i,j}, v)$, where $v \notin \{C_1, C_2, \ldots, C_p\}$, cannot be in the path, since as shown earlier, the path would then end at v.

We now show that there exists an assignment of truth values to x_k for all $k = 1, 2, \ldots, q$, such that for all $i = 1, 2, \ldots, p$, for some $j = 1, 2, 3$, $val(l_{i,j})$ is *true*, and thus C is satisfiable. For all $i = 1, 2, \ldots, p$, for all $j = 1, 2, 3$, $val(l_{i,j})$ is assigned *true* iff $(C_{i+1}, l_{i,j}), (l_{i,j}, C_i)$ are in the path. This assignment causes C to be true since as shown earlier, for all $i = 1, 2, \ldots, p$, for some $j = 1, 2, 3$, edges $(C_{i+1}, l_{i,j}), (l_{i,j}, C_i)$ are in the path.

Further, it is not possible that for some $k = 1, 2, \ldots, q$, x_k and $\bar{x_k}$ are both assigned *true*. If x_k and $\bar{x_k}$ are both assigned *true*, then there must exist symbols $l_{i,j}$ and $l_{r,s}$ such that edges $(C_{i+1}, l_{i,j}), (l_{i,j}, C_i)$, $(C_{r+1}, l_{r,s}), (l_{r,s}, C_r)$ are in the path, and $val(l_{i,j}) = x_k$, $val(l_{r,s}) = \bar{x_k}$. Thus, $|neg_k|)0$, $|pos_k|)0$, $l_{i,j} = pos_k(u)$, for some u, $u = 1, 2, \ldots, |pos_k|$, and $l_{r,s} = neg_k(v)$, for some v, $v = 1, 2, \ldots, |neg_k|$. However, this is not possible, since one of $l_{i,j}$ and $l_{r,s}$ is in the path between x_{k+1} and x_k, and a path cannot contain a site node more than once. □

We now show that the problem of computing a set of dependencies, Δ, that is minimal with respect to (V, E, D) and $\widehat{G}_i$, is NP-hard.

Proof of Theorem 7: We show that the NP-complete problem of determining if $\Delta' = \emptyset$ is not minimal with respect to $\widehat{G}_i$ and (V, E, D) can be Turing-reduced to the problem of computing a Δ that is minimal with respect to $\widehat{G}_i$ and (V, E, D).

Consider a subroutine $S((V, E, D), \widehat{G}_i)$ that returns a set of dependencies Δ that is minimal with respect to $\widehat{G}_i$ and (V, E, D). An algorithm for solving the problem of determining if $\Delta' = \emptyset$

a cycle in (V', E', D') such that all the transaction nodes in the cycle are in S_1 (since there are dependencies $(C_i, l_{i,j}) \rightarrow (l_{i,j}, C_{i+1})$, for all $i = 1, 2, \ldots, p$, for all $j = 1, 2, 3$, a path from C_r to C_s is possible only if $r > s$). Similarly, there can be no cycle in (V', E', D') such that all the transaction nodes in the cycle are in S_2. In addition, there is no cycle in (V', E', D') consisting of transaction nodes from both S_1 and S_2 since such a cycle must have edges $(v_1, l_{i,j})$, $(l_{i,j}, v_2)$, for some site node $l_{i,j}$ and $v_1 \in S_1$ and $v_2 \in S_2$ (s_0 and $l_{i,j}$ are the only site nodes that have edges to transaction nodes in both S_1 and S_2). Let $l_{i,j} = pos_r(k)$ (the argument for $l_{i,j} = neg_r(k)$ is similar). Due to dependencies $(C_i, l_{i,j}) \rightarrow (l_{i,j}, P_{r,k})$, $(C_{i+1}, l_{i,j}) \rightarrow (l_{i,j}, P_{r,k})$, $v_1 = C_i$ or C_{i+1}, and $v_2 = P'_{r,k}$. The only other edge incident on $P'_{r,k}$ is $(P'_{r,k}, l'_{i,j})$. However, due to the dependency $(P'_{r,k}, l'_{i,j}) \rightarrow (l'_{i,j}, P_{r,k+1})$, if $k < |pos_r|$ or $(P'_{r,k}, l'_{i,j}) \rightarrow (l'_{i,j}, x_{r+1})$, if $k = |pos_r|$, the path ends at $P'_{r,k}$ and cannot be part of a cycle. Thus, there can be no cycle in (V', E', D') consisting of transaction nodes from both S_1 and S_2, and (V', E', D') is acyclic.

We now show that (V, E, D) contains a cycle involving D_2 iff there is a path from D_2 to C_1 through node s_1 in (V, E, D). If (V, E, D) contains a cycle involving D_2, due to the dependency $(x_{p+1}, s_1) \rightarrow (s_1, D_2)$, there cannot be a path from C_1 to D_2 through s_1. Thus, there must be a path from D_2 to C_1 through s_1 that results in the cycle. If in (V, E, D), there is no cycle involving D_2, then if there was a path from D_2 to C_1 through node s_1, then there would be a cycle due to the edges $(C_1, s_2), (s_2, D_2)$. Thus, we need to show that C is satisfiable iff there is a path from D_2 to C_1 through s_1.

If C is satisfiable, we show that there is a path from D_2 to C_1 through s_1 by specifying the edges in the path. Since C is satisfiable, there exists an assignment of truth values to x_k for all $k = 1, 2, \ldots, q$, such that for all $i = 1, 2, \ldots, p$, for some $j = 1, 2, 3$, $val(l_{i,j})$ is *true*. We now specify the edges in the path. Edges $(D_2, s_1), (s_1, x_{m+1})$ are in the path. For all $i = 1, 2, \ldots, q$, if x_i is *false* in the assignment, then the following edges are in the path:

- $(x_{i+1}, pos'_i(|pos_i|)), (pos'_i(|pos_i|), P'_{i,|pos_i|}), \ldots, (pos_i(1), P_{i,1}), (P_{i,1}, emp_i), (emp_i, x_i)$, if $|pos_i| > 0$,
- $(x_{i+1}, emp_i), (emp_i, x_i)$, if $|pos_i| = 0$,

else if x_i is *true* in the assignment, the path contains the edges:

- $(x_{i+1}, neg'_i(|neg_i|)), (neg'_i(|neg_i|), N'_{i,|neg_i|}), \ldots, (neg_i(1), N_{i,1}), (N_{i,1}, emp'_i), (emp'_i, x_i)$, if $|neg_i| > 0$,
- $(x_{i+1}, emp'_i), (emp'_i, x_i)$, if $|neg_i| = 0$.

Edges $(x_1, s_0), (s_0, C_{p+1})$ are also in the path. For all $i = 1, 2, \ldots, p$, edges $(C_{i+1}, l_{i,j}), (l_{i,j}, C_i)$ are in the path, for some $j = 1, 2, 3$ such that $val(l_{i,j})$ is *true* in the assignment.

In the above choice of edges, we show that no node appears more than once in the path. Nodes other than $l_{i,j}$, trivially, appear only once. For any node $l_{i,j}$, it is in the path between nodes C_{i+1} and C_i only if $val(l_{i,j})$ is *true* in the assignment. If $l_{i,j} = pos_r(k)$, then $val(l_{i,j}) = x_r$, and since x_r is *true* in the assignment, $l_{i,j}$ is not among the nodes in the path between x_{r+1} and x_r. Similarly, if $l_{i,j} = neg_r(k)$, then $val(l_{i,j}) = \bar{x}_r$, and since x_r is *false* in the assignment, $l_{i,j}$ is not among the nodes in the path between x_{r+1} and x_r. Thus, the above edges constitute a path from D_2 to C_1 through s_1.

On the other hand, if there is a path from D_2 to C_1 through s_1, then we show that for all $i = 1, 2, \ldots, q$, the path contains either edges

- $(x_{i+1}, pos'_i(|pos_i|)), (pos'_i(|pos_i|), P'_{i,|pos_i|}), \ldots, (pos_i(1), P_{i,1}), (P_{i,1}, emp_i), (emp_i, x_i)$, if $|pos_i| > 0$,
- $(x_{i+1}, emp_i), (emp_i, x_i)$, if $|pos_i| = 0$,

or edges

- $(x_{i+1}, neg'_i(|neg_i|)), (neg'_i(|neg_i|), N'_{i,|neg_i|}), \ldots, (neg_i(1), N_{i,1}), (N_{i,1}, emp'_i), (emp'_i, x_i)$, if $|neg_i| > 0$,

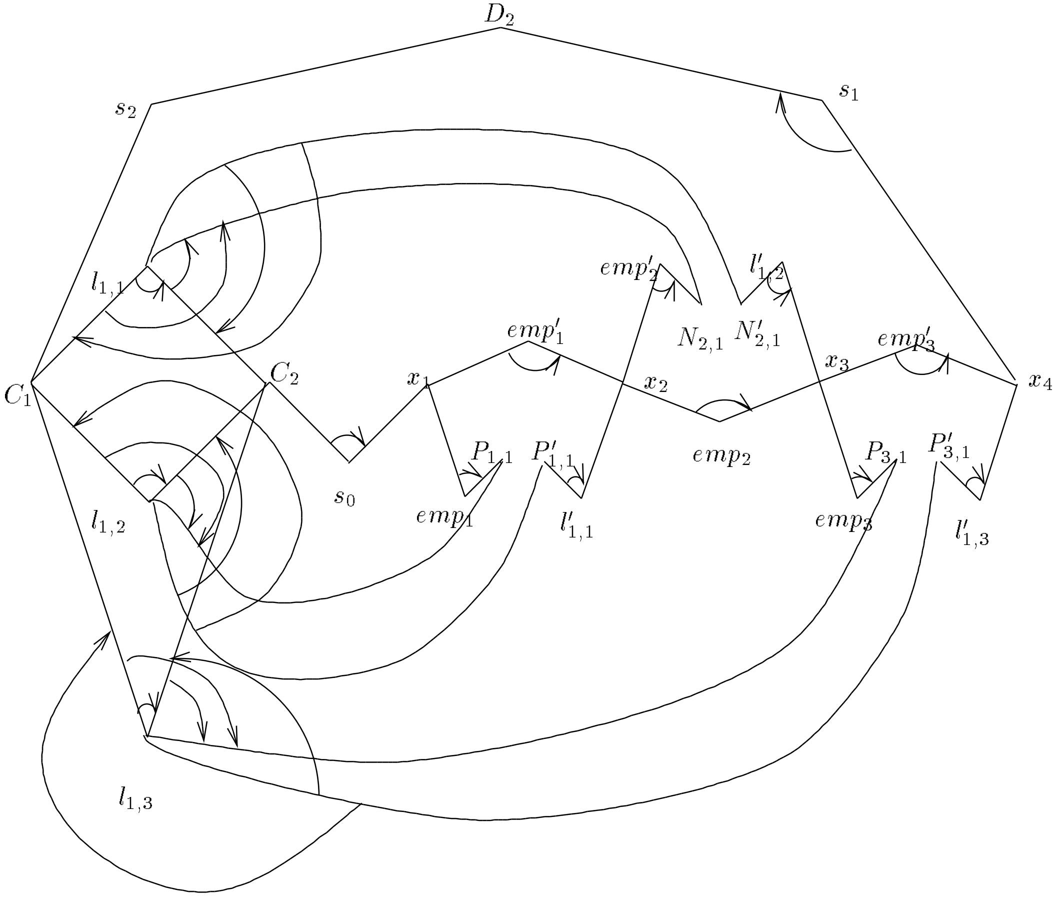

Figure 6: TSGD

  - $(x_i, emp_i), (emp_i, P_{i,1}), (P_{i,1}, pos_i(1)), (pos_i(1), P'_{i,1}), (P'_{i,1}, pos'_i(1)), (pos'_i(1), P_{i,2}),$ $(P_{i,2}, pos_i(2)), \ldots, (P'_{i,|pos_i|}, pos'_i(|pos_i|)), (pos'_i(|pos_i|), x_{i+1})$, if $|pos_i| > 0$,
  - $(x_i, emp_i), (emp_i, x_{i+1})$, if $|pos_i| = 0$,
  - $(x_i, emp'_i), (emp'_i, N_{i,1}), (N_{i,1}, neg_i(1)), (neg_i(1), N'_{i,1}), (N'_{i,1}, neg'_i(1)), (neg'_i(1), N_{i,2}),$ $(N_{i,2}, neg_i(2)), \ldots, (N'_{i,|neg_i|}, neg'_i(|neg_i|)), (neg'_i(|neg_i|), x_{i+1})$, if $|neg_i| > 0$,
  - $(x_i, emp'_i), (emp'_i, x_{i+1})$, if $|neg_i| = 0$,
- $(x_{p+1}, s_1), (s_1, D_2), (D_2, s_2), (s_2, C_1)$.

Note that there are two edges incident on each of the symbols emp_i, emp'_i, $l'_{i,j}$, $P_{i,j}$, $P'_{i,j}$, $N_{i,j}$ and $N'_{i,j}$. In addition, there are four edges incident on every symbol $l_{i,j}$.

- If $l_{i,j} = pos_r(k)$, there are edges $(C_i, l_{i,j})$, $(l_{i,j}, C_{i+1})$, $(P_{r,k}, l_{i,j})$ and $(l_{i,j}, P'_{r,k})$ in the TSGD.
- If $l_{i,j} = neg_r(k)$, there are edges $(C_i, l_{i,j})$, $(l_{i,j}, C_{i+1})$, $(N_{r,k}, l_{i,j})$ and $(l_{i,j}, N'_{r,k})$ in the TSGD.

The set of dependencies D consist of

- $(C_i, l_{i,j}) \rightarrow (l_{i,j}, C_{i+1})$, for all $i = 1, 2, \ldots, p$, for all $j = 1, 2, 3$,
- $(C_{p+1}, s_0) \rightarrow (s_0, x_1)$,
- for $i = 1, 2, \ldots, q$,
  - $(x_i, emp_i) \rightarrow (emp_i, P_{i,1}), (P'_{i,1}, pos'_i(1)) \rightarrow (pos'_i(1), P_{i,2}), (P'_{i,2}, pos'_i(2)) \rightarrow (pos'_i(2), P_{i,3}),$ $\ldots, (P'_{i,|pos_i|}, pos'_i(|pos_i|)) \rightarrow (pos'_i(|pos_i|), x_{i+1})$, if $|pos_i| > 0$,
  - $(x_i, emp_i) \rightarrow (emp_i, x_{i+1})$, if $|pos_i| = 0$,
  - $(x_i, emp'_i) \rightarrow (emp'_i, N_{i,1}), (N'_{i,1}, neg'_i(1)) \rightarrow (neg'_i(1), N_{i,2}), (N'_{i,2}, neg'_i(2)) \rightarrow (neg'_i(2), N_{i,3}),$ $\ldots, (N'_{i,|neg_i|}, neg'_i(|neg_i|)) \rightarrow (neg'_i(|neg_i|), x_{i+1})$, if $|neg_i| > 0$,
  - $(x_i, emp'_i) \rightarrow (emp'_i, x_{i+1})$, if $|neg_i| = 0$,
- for each symbol $l_{i,j}$,
  - if $l_{i,j} = pos_r(k)$, there are edges $(C_i, l_{i,j})$, $(l_{i,j}, C_{i+1})$, $(P_{r,k}, l_{i,j})$ and $(l_{i,j}, P'_{r,k})$ in the TSGD. The following dependencies are in D.
    $(C_i, l_{i,j}) \rightarrow (l_{i,j}, P_{r,k})$, $(C_{i+1}, l_{i,j}) \rightarrow (l_{i,j}, P_{r,k})$,
    $(P'_{r,k}, l_{i,j}) \rightarrow (l_{i,j}, C_i)$, $(P'_{r,k}, l_{i,j}) \rightarrow (l_{i,j}, C_{i+1})$.

  - if $l_{i,j} = neg_r(k)$, there are edges $(C_i, l_{i,j})$, $(l_{i,j}, C_{i+1})$, $(N_{r,k}, l_{i,j})$ and $(l_{i,j}, N'_{r,k})$ in the TSGD. The following dependencies are in D.
    $(C_i, l_{i,j}) \rightarrow (l_{i,j}, N_{r,k})$, $(C_{i+1}, l_{i,j}) \rightarrow (l_{i,j}, N_{r,k})$,
    $(N'_{r,k}, l_{i,j}) \rightarrow (l_{i,j}, C_i)$, $(N'_{r,k}, l_{i,j}) \rightarrow (l_{i,j}, C_{i+1})$.
- $(x_{q+1}, s_1) \rightarrow (s_1, D_2)$,

It is easy to see that the number of steps required to construct the TSGD (V, E, D) is $O(p+q)$. If $C = x_1 \vee \bar{x}_2 \vee x_3$, then the constructed TSGD is as shown in figure 6.

Our goal is to show that C is satisfiable iff (V, E, D) does not contain any cycles involving D_2. We begin by showing that the TSGD (V, E, D) satisfies the conditions. In D, the only dependency on any of D_2's edges is $(x_{m+1}, s_1) \rightarrow (s_1, D_2)$. Thus, in D, there are only dependencies into D_2's edges. Also, the set of dependencies, D, is legal. Further, we show that the TSGD (V', E', D') is acyclic, where
$V' = V - D_2$,
$E' = E - \{(D_2, s_1), (D_2, s_2)\}$, and
$D' = D - \{(x_{q+1}, s_1) \rightarrow (s_1, D_2)\}$.
Let $S_1 = \{C_1, C_2, \ldots, C_{p+1}\}$, and $S_2 = \{x_1, x_2, \ldots, x_{q+1}\} \cup \{N_{r,k}, N'_{r,k} : r = 1, 2, \ldots, q, k = 1, 2, \ldots, |neg_r|\} \cup \{P_{r,k}, P'_{r,k} : r = 1, 2, \ldots, q, k = 1, 2, \ldots, |pos_r|\}$. Note that there cannot exist

Theorem 7 is a consequence of the following NP-completeness result.

Theorem 17: The following problem is NP-complete.
Given a TSGD (V, E, D), and a transaction node $\widehat{G}_i \in V$ in the TSGD such that for all transactions $\widehat{G}_j \in V$, for all sites s_k, dependency $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k) \notin D$. Also, TSGD (V', E', D') resulting from the deletion of $\widehat{G}_i$, its edges and dependencies from (V, E, D), is acyclic. Is $\Delta = \emptyset$ not minimal with respect to the TSGD and transaction $\widehat{G}_i$?

Proof: We begin by showing that $\Delta = \emptyset$ is not minimal with respect to $\widehat{G}_i$ and (V, E, D) iff (V, E, D) contains a cycle involving transaction $\widehat{G}_i$. Since $\Delta = \emptyset$, and universal quantification over $\emptyset$ is always *true*, by the definition of minimality Δ is minimal with respect to $\widehat{G}_i$ and (V, E, D) iff (V, E, D) does not contain any cycles involving $\widehat{G}_i$. As a result, it suffices to show that the following problem is NP-complete.
Does (V, E, D) contain a cycle involving $\widehat{G}_i$?

The above problem is in NP since a non-deterministic algorithm only needs to guess a sequence of at most $m + n$ nodes (since in a path no node can appear more than once and the TSGD has at most $m + n$ nodes) and then check in polynomial time if the edges between the nodes result in a path from $\widehat{G}_i$ to $\widehat{G}_i$ in the TSGD (V, E, D).

We show a polynomial transformation from 3-SAT. Consider a formula in *Conjunctive Normal Form (CNF)* $C = C_1 \wedge C_2 \wedge \cdots \wedge C_p$ that is defined over literals $x_1, x_2, \ldots, x_q$. Let $l_{i,j}$, $l'_{i,j}$, $i = 1, 2, \ldots, p$, $j = 1, 2, 3$, be new symbols for the j^{th} literal in clause C_i. Each symbol $l_{i,j}$ has a value, $val(l_{i,j})$, that is either x_k or $\bar{x}_k$, $k = 1, 2, \ldots, q$. Note that for any two symbols $l_{i,j}$ and $l_{i,k}$, $j \neq k$, $val(l_{i,j}) \neq val(l_{i,k})$. In addition, for every literal x_i, there are new symbols emp_i and emp'_i. For $r = 1, 2, \ldots, q$, pos_r denotes the sequence of symbols $l_{i,j}$ in the order of increasing i, such that $val(l_{i,j}) = x_r$, and pos'_r, the corresponding sequence of symbols $l'_{i,j}$ in the order of increasing i, such that $val(l_{i,j}) = x_r$. For $r = 1, 2, \ldots, q$, neg_r denotes the sequence of symbols $l_{i,j}$ in the order of increasing i, such that $val(l_{i,j}) = \bar{x}_r$, and neg'_r, the corresponding sequence of symbols $l'_{i,j}$ in the order of increasing i, such that $val(l_{i,j}) = \bar{x}_r$. Also $|pos_r|$ denotes the number of elements in the sequence pos_r and for $k = 1, 2, \ldots, |pos_r|$, $pos_r(k)$ denotes the k^{th} element in the sequence pos_r. $|pos'_r|$, $pos'_r(k)$, $|neg_r|$, $neg_r(k)$, $|neg'_r|$ and $neg'_r(k)$ are similarly defined. We introduce new symbols $P_{r,k}$, $P'_{r,k}$ for each $pos_r(k)$, $r = 1, 2, \ldots, q$, $k = 1, 2, \ldots, |pos_r|$ and new symbols $N_{r,k}, N'_{r,k}$ for each $neg_r(k)$, $r = 1, 2, \ldots, q$, $k = 1, 2, \ldots, |neg_r|$. We illustrate the notation by means of the following example.

Example: Let $C = (x_1 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_1 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_4 \vee x_1)$.
$val(l_{1,1}) = x_1$, $val(l_{2,2}) = \bar{x}_1$, $val(l_{3,2}) = \bar{x}_4$.
$pos_1 = l_{1,1} \circ l_{3,3}$, $neg_1 = l_{2,2}$, $pos_2 = null$.
$pos'_1 = l'_{1,1} \circ l'_{3,3}$, $neg'_2 = l'_{2,1} \circ l'_{3,1}$.
Also, $|pos_1| = 2$, $|pos_2| = 0$, $|neg'_2| = 2$.
$pos_1(1) = l_{1,1}$, $pos_1(2) = l_{3,3}$, $neg'_1(1) = l'_{2,2}$, $neg'_2(2) = l'_{3,1}$. □

We now construct the TSGD as follows. The set of nodes V consist of transaction and site nodes. The transaction nodes in the TSGD consists of $C_1, C_2, \ldots, C_p, C_{p+1}$, $x_1, x_2, \ldots, x_q, x_{q+1}$, D_2 (C_{p+1}, x_{q+1} and D_2 are new symbols) in addition to $P_{r,k}$, $P'_{r,k}$ for all $r = 1, 2, \ldots, q$, $k = 1, 2, \ldots, |pos_r|$ and $N_{r,k}$, $N'_{r,k}$ for all $r = 1, 2, \ldots, q$, $k = 1, 2, \ldots, |neg_r|$. Site nodes consist of $l_{i,j}$, $l'_{i,j}$, $i = 1, 2, \ldots, p$, $j = 1, 2, 3$, in addition to new symbols s_0, s_1, s_2 and for all i, $i = 1, 2, \ldots, q$, emp_i, emp'_i.

The set of edges E consist of

- $(C_i, l_{i,j})$ and $(l_{i,j}, C_{i+1})$, for all $i = 1, 2, \ldots, p$, for all $j = 1, 2, 3$,
- (C_{p+1}, s_0), (s_0, x_1),
- for $i = 1, 2, \ldots, q$,

- $act(ser_k(G_i))$: $O(n)$. At most n dependencies are added to D as a result of the execution of $act(ser_k(G_i))$ since every transaction $\widehat{G}_j$ has at most one operation $ser_k(G_j)$ and there are at most n transactions in the TSGD. If a dependency $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k)$ is added to D, then both $tot\_count(\widehat{G}_j, s_k)$ and $act\_count(\widehat{G}_j, s_k)$ are incremented by 1.
- $cond(ack(ser_k(G_i)))$: $O(1)$.
- $act(ack(ser_k(G_i)))$: $O(n)$. For every transaction $\widehat{G}_j$ such that a dependency $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k) \in D$, $act\_count(\widehat{G}_j, s_k)$ is decremented by 1. Since every transaction $\widehat{G}_j$ has at most one operation $ser_k(G_j)$ and there are at most n transactions in the TSGD, D contains at most n such dependencies when $act(ack(ser_k(G_i)))$ executes.
- $cond(fin_i)$: $O(d_{av})$. $cond(fin_i)$ holds only if for every site $s_k \in exec(G_i)$, $tot\_count(\widehat{G}_i, s_k) = 0$.
- $act(fin_i)$: $O(nd_{av})$. For every transaction $\widehat{G}_j$ such that a dependency $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k) \in D$, where $s_k \in exec(G_i)$, $tot\_count(\widehat{G}_j, s_k)$ is decremented by 1, and the dependency deleted from D. Since D contains at most nd_{av} such dependencies (a transaction has d_{av} operations and there are at most n transactions in the TSGD), the number of steps in $act(fin_i)$ is $O(nd_{av})$.

Since $cond(init_i)$ and $cond(ack(ser_k(G_i)))$ are both *true*, the only operations in WAIT are either $ser_k(G_i)$ for some transaction $\widehat{G}_i$ and site $s_k \in exec(G_i)$, or fin_i for some transaction $\widehat{G}_i$. Also, execution of $act(o_j)$, for an operation o_j, can cause $cond(ser_k(G_i))$ to hold only if execution of $act(o_j)$ causes $act\_count(\widehat{G}_i, s_k)$ to be decremented. In addition, execution of $act(o_j)$, for some operation o_j, can cause $cond(fin_i)$ for some transaction $\widehat{G}_i$ to hold only if $act(o_j)$ decrements $tot\_count(\widehat{G}_i, s_k)$, for some site $s_k \in exec(G_i)$.

We now specify $wait(o_j)$ for each of the operations o_j.

- $wait(init_i)$: $\emptyset$. Execution of $act(init_i)$ does not result in any counters being decremented.
- $wait(ser_k(G_i))$: $\emptyset$. Execution of $act(ser_k(G_i))$ does not result in any counters being decremented.
- $wait(ack(ser_k(G_i)))$: $\{ser_k(G_j) : (ser_k(G_j) \in \text{WAIT})\}$.

  For any operation $fin_j \in$ WAIT, execution of $act(ack(ser_k(G_i)))$ cannot cause $cond(fin_j)$ to hold since only $act\_count(\widehat{G}_l, s_k)$, $s_k \in exec(G_l)$, is decremented due to the execution of $act(ack(ser_k(G_i)))$.

  Further, execution of $act(ack(ser_k(G_i)))$ cannot cause $cond(ser_p(G_j))$ to hold for some operation $ser_p(G_j) \in$ WAIT, $s_p \neq s_k$, since execution of $act(ack(ser_k(G_i)))$ results in only $act\_count(\widehat{G}_l, s_k)$, $s_k \in exec(G_l)$, being decremented and $s_k \neq s_p$. Thus, $wait(ack(ser_k(G_i)))$ is restricted to operations $ser_k(G_l)$, for transactions $\widehat{G}_l \in V$ such that $s_k \in exec(G_l)$.
- $wait(fin_i)$: $\{fin_j : fin_j \in \text{WAIT}\}$.

  For any operation $ser_k(G_j) \in$ WAIT, $cond(ser_k(G_j))$ cannot hold due to the execution of $act(fin_i)$ since only $tot\_count(\widehat{G}_j, s_k)$, for some transaction $\widehat{G}_j$ and some site $s_k \in (exec(G_i) \cap exec(G_j))$, is decremented due to the execution of $act(fin_i)$.

Thus, the number of steps in $cond(o_l)$, for any operation $o_l \in wait(ack(ser_k(G_i)))$ is $O(1)$, and the number of steps in $cond(o_l)$ for any operation $o_l \in wait(fin_i)$, is $O(d_{av})$. Further, in the worst case, the number of operations in both $wait(ack(ser_k(G_i)))$ and $wait(fin_i)$ is $O(n)$ (since there are at most n transactions in the TSGD).

Proof of Theorem 6: The complexity of Scheme 2 is dominated by the number of steps in $act(init_i)$ which is $O(n^2 d_{av})$. Thus, the complexity of Scheme 2 is $O(n^2 d_{av})$. □

dependency $(\widehat{G}_l, s_k) \rightarrow (\widehat{G}_{(j \bmod r)+1}, s_k)$, for all $\widehat{G}_l, s_k$ such that $s_k \in (exec(G_{(j \bmod r)+1}) \cap exec(G_l))$, $act(fin_{(j \bmod r)+1})$ cannot be executed unless $act(fin_j)$ has executed. Thus, $act(fin_j)$ must execute before $act(fin_{(j \bmod r)+1})$ executes. Now suppose $act(fin_k)$ executes for some $k = 1, 2, \ldots, r$. From the above arguments, if follows that $act(fin_k)$ executes before $act(fin_k)$ executes, which is not possible. Thus, none of $act(fin_j)$ can execute, for all j, $j = 1, 2, \ldots, r$.

Consider a point p during the execution of Scheme 2 when all of $act(ser_{i_j}(G_j))$, $act(ser_{i_j}(G_{(j \bmod r)+1}))$, $j = 1, 2, \ldots, r$ have been executed. Since for all j, $j = 1, 2, \ldots, r$, $act(fin_j)$ does not execute and $act(init_j)$ executes before $act(ser_{i_j}(G_j))$, by Lemma 4, $(\widehat{G}_j, s_{i_j}) \rightarrow (\widehat{G}_{(j \bmod r)+1}, s_{i_j}) \in D$ at p. By Lemma 3 and Lemma 8, since the TSGD is legal at p, there is no dependency $(\widehat{G}_{(j \bmod r)+1}, s_{i_j}) \rightarrow (\widehat{G}_j, s_{i_j})$ in D, for all j, $j = 1, 2, \ldots, r$. Thus, the edges $(\widehat{G}_1, s_{i_r}), (s_{i_r}, \widehat{G}_r), (\widehat{G}_r, s_{i_{r-1}}), \ldots, (\widehat{G}_2, s_{i_1}), (s_{i_1}, \widehat{G}_1)$ in the TSGD form a cycle. However, this leads to a contradiction since by Lemma 2 and Lemma 8, the TSGD does not contain cycles at any point during the execution of Scheme 2. Thus, $ser(S)$ is serializable. □

Before performing the complexity analysis for Scheme 2, we first analyze the number of steps required by Eliminate_Cycles.

Theorem 16: Eliminate_Cycles terminates in $O(n^2 d_{av})$ steps.

Proof: Since there are at most n transactions in the TSGD, and each transaction has d_{av} edges, the TSGD has at most nd_{av} edges. The number of edges marked by the algorithm is $O(nd_{av})$ since each of $\widehat{G}_i$'s edges are marked at most n times and every other edge in the TSGD is marked at most once. Also, since every time a state transition is made, an edge is marked, the number of state transitions $st_j \rightarrow st_k$ possible is bounded above by $O(nd_{av})$. Thus, the number of reverse transitions made are also $O(nd_{av})$. Further, at every transaction node, in the worst case, there are nd_{av} choices of pairs of edges that can be made. Each of these must be examined in the worst case and since there are at most n transaction nodes, Eliminate_Cycles terminates in $O(n^2 d_{av})$ steps. □

Complexity Analysis of Scheme 2:

We first describe additional data structures involved in the implementation of Scheme 2. We then analyze, for every operation o_j, the number of steps in $cond(o_j)$ and $act(o_j)$, and the characteristics of $wait(o_j)$ (the number of operations and their types).

Implementation: Associated with every edge $(\widehat{G}_i, s_k)$ in the TSGD are associated two counters: $tot\_count(\widehat{G}_i, s_k)$ and $act\_count(\widehat{G}_i, s_k)$. The counter $tot\_count(\widehat{G}_i, s_k)$ maintains a count of the total number of dependencies into an edge, while $act\_count(\widehat{G}_i, s_k)$ keeps a count of dependencies $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$ such that $act(ack(ser_k(G_j)))$ has not yet completed execution.

The number of steps in $cond(o_j)$ and $act(o_j)$, for each operation o_j, are as follows.

- $cond(init_i)$: $O(1)$.
- $act(init_i)$: $O(n^2 d_{av})$. Since there are at most n transactions in the TSGD and every transaction has d_{av} operations, in the worst case, $act(init_i)$ results in the addition of $O(nd_{av})$ dependencies of the form $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$ to D, where $s_k \in exec(G_i)$. Addition of each of these dependencies, say, $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$ for some $s_k \in exec(G_i)$, results in updates to $tot\_count(\widehat{G}_i, s_k)$, and in certain cases to $act\_count(\widehat{G}_i, s_k)$ depending on whether or not $act(ack(ser_k(G_j)))$ has completed execution. Further, by Theorem 16, Eliminate_Cycles terminates in $O(n^2 d_{av})$ steps. Thus, the number of steps required in order to update D when $act(init_i)$ executes is $O(n^2 d_{av})$.
- $cond(ser_k(G_i))$: $O(1)$. $cond(ser_k(G_i))$ holds only if $act\_count(\widehat{G}_i, s_k) = 0$. This check takes $O(1)$ steps.

marks edge (v_{2r+1}, v_{2r+2}) "used" if it has not been marked "used" already. If $v_{2r+2} = \widehat{G}_i$, then a dependency $(v_{2r}, v_{2r+1}) \rightarrow (v_{2r+1}, v_{2r+2})$ is added to Δ if it has not already been added to Δ (irrespective of whether or not (v_{2r+1}, v_{2r+2}) is marked "used"). □

Theorem 15: The procedure Eliminate_Cycles ensures that there are no cycles involving $\widehat{G}_i$ in $(V, E, D \cup \Delta)$.

Proof: Let us suppose that the set of edges $(\widehat{G}_i, v_1)$, (v_1, v_2), …, (v_{2k-2}, v_{2k-1}), $(v_{2k-1}, \widehat{G}_i)$, $k > 1$, form a cycle in the TSGD $(V, E, D \cup \Delta)$. Thus, at least one of the following cases must be true.

1. There is a path $(\widehat{G}_i, v_1)(v_1, v_2) \cdots (v_{2k-2}, v_{2k-1})(v_{2k-1}, \widehat{G}_i)$, $k > 1$.
2. There is a path $(\widehat{G}_i, v_{2k-1})(v_{2k-1}, v_{2k-2}) \cdots (v_2, v_1)(v_1, \widehat{G}_i)$, $k > 1$.

By Lemma 7, for Case 1, Eliminate_Cycles ensures that a dependency $(v_{2k-2}, v_{2k-1}) \rightarrow (v_{2k-1}, \widehat{G}_i)$ is added to Δ. Thus, TSGD $(V, E, D \cup \Delta)$ cannot contain the path $(\widehat{G}_i, v_1)(v_1, v_2) \cdots (v_{2k-1}, \widehat{G}_i)$ and Case 1 is not true. By a similar argument, it can be shown that Case 2 is not true since Eliminate_Cycles ensures that a dependency $(v_2, v_1) \rightarrow (v_1, \widehat{G}_i)$ is added to Δ. Thus, the TSGD $(V, E, D \cup \Delta)$ contains no cycles involving $\widehat{G}_i$. □

Lemma 8: For all transactions $\widehat{G}_i$, $act(init_i)$ preserves the acyclicity of TSGD (V, E, D) and the legality of D.

Proof: Let (V_1, E_1, D_1) denote the TSGD before the execution of $act(init_i)$, and (V_2, E_2, D_2) denote the TSGD after execution of $act(init_i)$. It is given that (V_1, E_1, D_1) is acyclic, and D_1 is legal. Also, since $act(init_i)$ results in the addition of $\widehat{G}_i$'s edges to (V_1, E_1, D_1) and no dependencies in D_1 are deleted, $V_1 \subset V_2$, $E_1 \subset E_2$ and $D_1 \subseteq D_2$. More precisely,
$V_2 := V_1 \cup \{\widehat{G}_i\}$.
$E_2 := E_1 \cup \{(\widehat{G}_i, s_k) : s_k \in exec(G_i)\}$.

We first show that D_2 is legal. Since, in the set of dependencies returned by Eliminate_Cycles, all the dependencies are of the form $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$ for some transaction $\widehat{G}_j \in V$ and some site s_k, in $D_2 - D_1$, all the dependencies are of the form $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$. Thus, since there is no dependency of the form $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k)$ in D_1 before the execution of $act(init_i)$, and D_1 is legal, D_2 is legal.

By Theorem 15, procedure eliminate_Cycles ensures that in (V_2, E_2, D_2), there are no cycles involving $\widehat{G}_i$. Since (V_1, E_1, D_1) is acyclic, $D_1 \subseteq D_2$ and $E_2 := E_1 \cup \{(\widehat{G}_i, s_k) : s_k \in exec(G_i)\}$, (V_2, E_2, D_2) does not contain any cycles not involving $\widehat{G}_i$. As a result, (V_2, E_2, D_2) does not contain any cycles and is acyclic. □

In the following theorem, we show that scheme 2 ensures the serializability of $ser(S)$.

Proof of Theorem 5: Suppose $ser(S)$ is not serializable. Thus, there exist distinct transactions, say, $\widehat{G}_1, \widehat{G}_2, \ldots, \widehat{G}_r$, $r > 1$, such that $ser_{i_1}(G_1)$ executes before $ser_{i_1}(G_2)$, $ser_{i_2}(G_2)$ executes before $ser_{i_2}(G_3)$, …, $ser_{i_r}(G_r)$ executes before $ser_{i_r}(G_1)$, and for all $j, k = 1, 2, \ldots, r$, $j \neq k$, $i_j \neq i_k$ (since for any site s_k, transaction $\widehat{G}_j$ has at most one operation $ser_k(G_j)$).

We claim for all j, $j = 1, 2, \ldots, r$, none of $act(fin_j)$ can execute, and thus none of $\widehat{G}_j$'s edges can be deleted from the TSGD. To see this, observe that for all j, $j = 1, 2, \ldots, r$, $ser_{i_j}(G_j)$ executes before $ser_{i_j}(G_{(j \bmod r)+1})$. Thus, by Lemma 4 and Lemma 5, if $act(fin_j)$ has not executed when Scheme 2 attempts to execute $act(fin_{(j \bmod r)+1})$, then $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_{(j \bmod r)+1}, s_k) \in D$ (since Scheme 2 attempts to execute $act(fin_{(j \bmod r)+1})$ after it executes $act(ser_{i_j}(G_{(j \bmod r)+1}))$, and $act(ser_{i_j}(G_{(j \bmod r)+1}))$ executes after $act(ser_{i_j}(G_j))$ as well as $act(init_{(j \bmod r)+1})$). As a result, since $cond(fin_{(j \bmod r)+1})$ requires there to be no

$w = \widehat{G}_i$, the choice is eliminated in one step by adding a dependency $(v,u) \rightarrow (u,w)$. If $w \neq \widehat{G}_i$, then Eliminate_Cycles makes a state transition $st_k \rightarrow st_l$ due to Step 3, where $st_l.v = w$, $st_l.t\_par(st_l.v) = (st_k.v) \circ (st_k.t\_par(st_l.v))$, $st_l.s\_par(st_l.v) = u \circ st_k.s\_par(st_l.v)$, and edge (u,w) is marked. As a result, this choice is eliminated and the number of unmarked edges in the TSGD just after the transition $st_k \rightarrow st_l$ is made is r. Thus, by the induction hypothesis, Eliminate_Cycles makes a reverse transition $st_l \rightarrow st_k$ in a finite number of steps. Since there are a finite number of choices of pairs of edges in state st_k, each choice is eliminated when a transition $st_k \rightarrow st_l$ is made, and no further state transitions (due to Step 3) can be made once all choices have been eliminated, Eliminate_Cycles makes the reverse transition $st_k \rightarrow st_j$ (due to Step 4) in a finite number of steps. □

Corollary 2: Eliminate_Cycles terminates in a finite number of steps.

Proof: The initial state $st_0 = (\widehat{G}_i, null, null)$. Every time a transition $st_0 \rightarrow st_j$ is made, by Lemma 6, a reverse transition $st_j \rightarrow st_0$ is made in a finite number of steps. Since there are a finite number of choices in state st_0, each choice is eliminated when a transition $st_0 \rightarrow st_j$ is made, and no further transitions can be made once all choices have been eliminated, Eliminate_Cycles terminates in a finite number of steps. □

In order to prove that Eliminate_Cycles detects all cycles in the TSGD, we first define the notion of a *path* in the TSGD.

Definition 6: In a TSGD (V, E, D), $(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$, $k > 0$, is a path from v_0 to v_k iff

- for all i, $i = 0, 1, 2, \ldots, k-1$, $(v_i, v_{i+1}) \in E$,
- for all i, $i = 0, 1, 2, \ldots, k-2$, dependency $(v_i, v_{i+1}) \rightarrow (v_{i+1}, v_{i+2}) \notin D$,
- for all pairs (i,j), such that $i, j = 0, 1, 2, \ldots, k$, $i < j$, and $(i,j) \neq (0,k)$, the following is true: $v_i \neq v_j$, and
- if $k \leq 2$, then $v_0 \neq v_k$.

If, in addition, $v_0 = v_k$, and $k > 2$, then the set of edges (v_0, v_1), (v_1, v_2), …, (v_{k-1}, v_k) form a *cycle*. □

Lemma 7: If there is a path from $\widehat{G}_i$ to a transaction node v_{2k}, $k > 0$, $(\widehat{G}_i, v_1)(v_1, v_2) \cdots (v_{2k-1}, v_{2k})$ in the TSGD (V, E, D), then

- if $v_{2k} \neq \widehat{G}_i$, then (v_{2k-1}, v_{2k}) is marked "used" during the execution of Eliminate_Cycles.
- if $v_{2k} = \widehat{G}_i$, $k > 1$, then a dependency $(v_{2k-2}, v_{2k-1}) \rightarrow (v_{2k-1}, v_{2k})$ is added to Δ.

Proof: We prove the above lemma by induction on k.

Basis (k=1): The lemma holds for $k = 1$ since by Corollary 2, Eliminate_Cycles terminates. Before termination, Eliminate_Cycles ensures that (v_1, v_2) is marked "used" if it already has not been marked "used" (since, by the definition of path, $v_2 \neq \widehat{G}_i$, and there is no dependency $(\widehat{G}_i, v_1) \rightarrow (v_1, v_2)$ in $D \cup \Delta$).

Induction: Assume the lemma is true for $k = r$, $r > 0$. We need to show that the lemma is true for $k = r+1$. Let the path be $(\widehat{G}_i, v_1)(v_1, v_2) \cdots (v_{2r-1}, v_{2r})(v_{2r}, v_{2r+1})(v_{2r+1}, v_{2r+2})$. By the induction hypothesis, edge (v_{2r-1}, v_{2r}) is marked "used". By the definition of path, $v_{2r} \neq \widehat{G}_i$. Thus, when (v_{2r-1}, v_{2r}) is marked, a transition $st_j \rightarrow st_l$ is made for some states st_j, st_l, where $st_l.v = v_{2r}$, $st_l.t\_par(v_{2r}) = (st_j.v) \circ (st_j.t\_par(v_{2r}))$, and $st_l.s\_par(v_{2r}) = v_{2r-1} \circ (st_j.s\_par(v_{2r}))$. By the definition of path, $v_{2r-1} \neq v_{2r+1}$, and thus, $head(st_l.s\_par(v_{2r})) \neq v_{2r+1}$. By Lemma 6, Eliminate_Cycles makes a reverse transition $st_l \rightarrow st_j$ in a finite number of steps. If $v_{2r+2} \neq \widehat{G}_i$, then by the definition of path, there is no dependency $(v_{2r}, v_{2r+1}) \rightarrow (v_{2r+1}, v_{2r+2})$ in $D \cup \Delta$, and before making the reverse transition $st_l \rightarrow st_j$, Eliminate_Cycles

- dependencies are deleted from D only when $act(fin_l)$ for some transaction $\widehat{G}_l$ executes,
- for all transactions $\widehat{G}_l$, $\widehat{G}_l \neq \widehat{G}_i$, $\widehat{G}_l \neq \widehat{G}_j$, $act(fin_l)$, results in the deletion of only $\widehat{G}_l$'s edges from the TSGD, and
- $act(fin_j)$ cannot be executed since $cond(fin_j)$ does not hold if $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k) \in D$, and thus $act(fin_j)$ executes only after $act(fin_i)$ executes. □

Lemma 5: For all sites s_k, for all transactions $\widehat{G}_i$, $\widehat{G}_j$, if $ser_k(G_i)$ executes before $ser_k(G_j)$, then $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$.

Proof: Let us assume that $act(ser_k(G_j))$ executes before $act(ser_k(G_i))$. If $act(ack(ser_k(G_j)))$ executes before $act(ser_k(G_i))$ executes, then $ser_k(G_j)$ executes before $ser_k(G_i)$ which leads to a contradiction. In addition, we show that $act(ser_k(G_i))$ cannot execute before $act(ack(ser_k(G_j)))$ completes execution. By Lemma 4, when $act(ser_k(G_i))$ executes, dependency $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k) \in D$ (since $act(ser_k(G_i))$ executes only after $act(init_i)$ executes, and $act(fin_j)$ executes only after $act(ack(ser_k(G_j)))$ executes). Thus, $cond(ser_k(G_i))$ does not hold, and $act(ser_k(G_i))$ cannot execute before $act(ack(ser_k(G_j)))$ executes. Thus, $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$ executes. □

We now prove that for all transactions $\widehat{G}_i$, $act(init_i)$ preserves the acyclicity of the TSGD. In order to prove the above, we need to first show that the set of dependencies Δ returned by Eliminate_Cycles, is such that $(V, E, D \cup \Delta)$ does not contain any cycles involving $\widehat{G}_i$. For this purpose, we introduce the notion of a *state* of Eliminate_Cycles. A state of Eliminate_Cycles, st_j, is a tuple $(v, t\_par(\widehat{G}_1), t\_par(\widehat{G}_2), \ldots, t\_par(\widehat{G}_q), s\_par(\widehat{G}_1), s\_par(\widehat{G}_2), \ldots, s\_par(\widehat{G}_q))$, where v is a transaction node, $\widehat{G}_1, \widehat{G}_2, \ldots, \widehat{G}_q$ are the transaction nodes in the TSGD, and for all transaction nodes $\widehat{G}_l \in V$, $t\_par(\widehat{G}_l)$ is a list of transaction nodes and $s\_par(\widehat{G}_l)$ is a list of site nodes. We denote the values of $v, t\_par(\widehat{G}_l)$ and $s\_par(\widehat{G}_l)$ in state st_j , where transaction node $\widehat{G}_l \in V$, by $st_j.v$, $st_j.t\_par(\widehat{G}_l)$ and $st_j.s\_par(\widehat{G}_l)$ respectively.

Eliminate_Cycles is said to be in state st_j at a point in between the execution any two of its steps, if at that point, $v = st_j.v$, and for all transaction nodes $\widehat{G}_l$, $t\_par(\widehat{G}_l) = st_j.t\_par(\widehat{G}_l)$ and $s\_par(\widehat{G}_l) = st_j.s\_par(\widehat{G}_l)$. Certain steps in Eliminate_Cycles cause it to move from one state to another. When a step causes Eliminate_Cycles to move from state st_j to state st_k, Eliminate_Cycles is said to make a *state transition* $st_j \rightarrow st_k$. Note that only steps 3 and 4 cause state transitions (we assume that the state of Eliminate_Cycles is undefined before the execution of Step 1). Also, if step 3 causes a state transaction $st_j \rightarrow st_k$, then $|st_j.t\_par(st_k.v)| < |st_k.t\_par(st_k.v)|$[5]. Similarly, if step 4 causes a state transaction $st_j \rightarrow st_k$, then $|st_j.t\_par(st_j.v)| > |st_k.t\_par(st_j.v)|$.

Lemma 6: If Eliminate_Cycles makes a state transition $st_j \rightarrow st_k$ due to Step 3, then after the execution of a finite number of steps, Eliminate_Cycles also makes the reverse transition $st_k \rightarrow st_j$ (due to Step 4).

Proof: We prove the above lemma by induction on num, where num is the number of unmarked edges in the TSGD just after the transition $st_j \rightarrow st_k$ is made.

Basis ($num = 0$): In this case, the only choices for pairs of edges $((v, u), (u, w))$ available in state st_k are those in which $w = \widehat{G}_i$ (if $w \neq \widehat{G}_i$, then since (u, w) is marked, the pair of edges cannot be chosen). Since a finite number of such choices exist, and each choice becomes unavailable once made (since a dependency $(v, u) \rightarrow (u, \widehat{G}_i)$ is added to Δ), Eliminate_Cycles makes the reverse transition $st_k \rightarrow st_j$ in a finite number of steps.

Induction: Assume the lemma is true for $num = r$. We show that the lemma is true for $num = r + 1$. Thus, just after the transition $st_j \rightarrow st_k$ is made, the number of unmarked edges in the TSGD is $r + 1$. For every choice of pairs of edges $((v, u), (u, w))$ while in state st_k, if

---

[5] For a list L, we denote the number of elements in the list by $|L|$.

-Appendix C-

In order to prove that Scheme 2 ensures the serializability of $ser(S)$, we need to first prove a series of lemmas. In the following lemma, we state the conditions under which Scheme 2 preserves the acyclicity of the TSGD.

Lemma 2: If, for all transactions $\widehat{G}_i$, $act(init_i)$ preserves acyclicity of the TSGD, then Scheme 2 ensures that at any point during its execution, the TSGD does not contain cycles.

Proof: Initially $V = E = D = \emptyset$. Thus, trivially, the TSGD (V, E, D) does not contain any cycles initially. Since only $act(o_j)$ modifies the TSGD, we show that it preserves the acyclicity property of the TSGD for all operations o_j.
$act(init_i)$: Assumed to preserve the acyclicity of the TSGD.
$act(ser_k(G_i))$: Since the addition of dependencies to an acyclic TSGD preserves its acyclicity, and $act(ser_k(G_i))$ only causes dependencies to be added to D, the TSGD stays acyclic.
$act(ack(ser_k(G_i)))$: The TSGD is not modified, and thus acyclicity of the TSGD is preserved.
$act(fin_i)$: The deletion of $\widehat{G}_i$'s edges from the TSGD when $act(fin_i)$ executes does not add any cycles to the TSGD. Thus, $act(fin_i)$ preserves the acyclicity of the TSGD. □

In addition to requiring that the TSGD be acyclic, we also require that the set of dependencies be *legal*, which is defined below.

Definition 5: A set of dependencies D is legal, if for all transactions $\widehat{G}_i, \widehat{G}_j$ and for all sites s_k, if $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k) \in D$, then $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k) \notin D$. □

In the following lemma, we state conditions under which Scheme 2 ensures the legality of the set of dependencies.

Lemma 3: If, for all transactions $\widehat{G}_i$, $act(init_i)$ preserves legality of D, then Scheme 2 ensures that at any point during its execution, D is legal.

Proof: Initially $D = \emptyset$. Thus, trivially, D is legal. Since only $act(o_j)$ modifies the TSGD, we show that it preserves the legality of D for all operations o_j.
$act(init_i)$: Assumed to preserve the legality of D.
$act(ser_k(G_i))$: $act(ser_k(G_i))$ causes dependencies $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k)$ to be added to D for all transactions $\widehat{G}_j \in V$ such that $act(ser_k(G_j))$ has not yet executed. Addition of $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k)$ to D would cause D to become illegal if D already contained a dependency $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$. However, if D contains the dependency $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$ before $act(ser_k(G_i))$ is executed, then $cond(ser_k(G_i))$ would not hold, and thus, $act(ser_k(G_i))$ would not be executed. Thus, $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k) \notin D$, and $act(ser_k(G_i))$ preserves the legality of D.
$act(ack(ser_k(G_i)))$: The TSGD is not modified and thus legality of D is preserved.
$act(fin_i)$: No new dependencies are added during $act(fin_i)$ and thus, D stays legal. □

Lemma 4: For all sites s_k, for all transactions $\widehat{G}_i$, $\widehat{G}_j$, if $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$ executes, then at any point p during the execution of Scheme 2, after the execution of $act(ser_k(G_i))$ and $act(init_j)$, but before the execution of $act(fin_i)$, the following is true: $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k) \in D$.

Proof: If $\widehat{G}_j \in V$ when $act(ser_k(G_i))$ executes, then execution of $act(ser_k(G_i))$ causes dependency $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k)$ to be added to D (since $act(ser_k(G_j))$ executes after $act(ser_k(G_i))$). If $\widehat{G}_j \notin V$ when $act(ser_k(G_i))$ executes, then when $act(init_j)$ is executed, dependency $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k)$ is added to D (since $act(fin_i)$ has not yet been executed, $\widehat{G}_i$'s edges are not deleted from the TSGD when $act(init_j)$ executes). Further, the dependency is not deleted until $act(fin_i)$ is executed since

$wait(ack(ser_k(G_i)))$ since $cond(ser_k(G_l))$ holds for all of them. However, in order to reduce the total number of steps, we include only one such operation $ser_k(G_p)$. An operation $ser_k(G_q) \neq ser_k(G_p)$, such that $ser_k(G_q) \in$ WAIT and is unmarked, can be included in $wait(ack(ser_k(G_p)))$, and yet another can be included in $wait(ack(ser_k(G_q)))$ and so on.

- $wait(fin_i)$: $\{fin_j : fin_j \in \text{WAIT}\}$.
  For any operation $ser_k(G_j) \in$ WAIT, $cond(ser_k(G_j))$ cannot hold due to the execution of $act(fin_i)$ since no operations are deleted from any of the insert queues due to the execution of $act(fin_i)$.

Since the number of steps in $cond(o_j)$, for any operation o_j is $O(1)$, the number of steps in $cond(o_l)$ for any operation $o_l \in wait(fin_i)$ or $o_l \in wait(ack(ser_k(G_i)))$ is $O(1)$. Further, in the worst case, the number of operations in $wait(ack(ser_k(G_i)))$ is $O(1)$ and the number of operations in $wait(fin_i)$ is $O(n)$ (since there are at most n transactions in the TSG).

Proof of Theorem 4: The complexity of Scheme 1 is dominated by the number of steps in $act(init_i)$ which is $O(m + n + nd_{av})$. Thus, the complexity of Scheme 1 is $O(m + n + nd_{av})$. $\square$

- $act(init_i)$: $O(m + n + nd_{av})$. Since each transaction has d_{av} operations, the number of steps required to add $\widehat{G}_i$'s edges to the TSG and to add $\widehat{G}_i$'s operations to the end of the insert queue for all $s_k \in exec(G_i)$ is $O(d_{av})$. A simple depth-first search can be employed in order to detect cycles in the TSG that involve $\widehat{G}_i$'s edges. A complete depth-first search of the TSG takes $O(m + n + nd_{av})$ steps since the TSG has at most $m + n$ nodes and at most nd_{av} edges (the number of transactions in the TSG never exceeds n, and every transaction $\widehat{G}_i$ has d_{av} operations).
- $cond(ser_k(G_i))$: $O(1)$. The number of steps needed to check if $ser_k(G_i)$ in the insert queue is either unmarked or the first element is $O(1)$.
- $act(ser_k(G_i))$: $O(1)$.
- $cond(ack(ser_k(G_i)))$: $O(1)$.
- $act(ack(ser_k(G_i)))$: $O(1)$. Deletion and addition of an element to the queues takes $O(1)$ steps. In addition, if $ser_k(G_i)$ is not the first operation in the delete queue, then f_count_i is incremented by 1. This takes $O(1)$ steps.
- $cond(fin_i)$: $O(1)$. $cond(fin_i)$ holds only if $f\_count_i = 0$. This check takes $O(1)$ steps.
- $act(fin_i)$: $O(d_{av})$. Deleting $\widehat{G}_i$'s edges from the TSG takes $O(d_{av})$ steps. Also, deletion of its operations takes $O(d_{av})$ steps. For every site $s_k \in exec(G_i)$, along with the deletion of operation $ser_k(G_i)$ from the delete queue, f_count_j is decremented by 1, where operation $ser_k(G_j)$ immediately follows $ser_k(G_i)$ in the delete queue for s_k. This takes $O(d_{av})$ steps.

Since $cond(init_i)$ and $cond(ack(ser_k(G_i)))$ are both *true*, the only operations in WAIT are either $ser_k(G_i)$ for some transaction $\widehat{G}_i$ and site $s_k \in exec(G_i)$, or fin_i for some transaction $\widehat{G}_i$. Also, execution of $act(o_j)$, for an operation o_j, can cause $cond(ser_k(G_i))$ to hold only if

- $o_j = act(ack(ser_k(G_l)))$, for some transaction $\widehat{G}_l$, and
- if $ser_k(G_i)$ is marked, $act(o_j)$ deletes all the operations that precede $ser_k(G_i)$ in the insert queue for s_k thus causing $ser_k(G_i)$ to be the first operation in the insert queue for s_k.

In addition, execution of $act(o_j)$, for some operation o_j, can cause $cond(fin_i)$ for some transaction $\widehat{G}_i$ to hold only if $act(o_j)$ deletes operations from some of the delete queue thus causing $\widehat{G}_i$'s operations to be first in the delete queues.

We now specify $wait(o_j)$ for each of the operations o_j.

- $wait(init_i)$: $\emptyset$. Execution of $act(init_i)$ does not result in the deletion of operations from any of the queues.
- $wait(ser_k(G_i))$: $\emptyset$. Execution of $act(ser_k(G_i))$ does not result in the deletion of operations from any of the queues.
- $wait(ack(ser_k(G_i)))$: $\{ser_k(G_j) : (ser_k(G_j) \in \text{WAIT}) \wedge (ser_k(G_j) \text{ is the first operation in the queue for site } s_k)\} \cup \{ser_k(G_l) \in \text{WAIT} \wedge (ser_k(G_l) \text{ is unmarked})\}$.

  For any operation $fin_j \in$ WAIT, execution of $act(ack(ser_k(G_i)))$ cannot cause $cond(fin_j)$ to hold since no operations are deleted from any of the delete queues due to the execution of $act(ack(ser_k(G_i)))$.

  Further, execution of $act(ack(ser_k(G_i)))$ cannot cause $cond(ser_p(G_j))$ to hold for some operation $ser_p(G_j) \in$ WAIT, $s_p \neq s_k$, since execution of $act(ack(ser_k(G_i)))$ does not result in the deletion of any operations from the insert queue for site s_p. Thus, $wait(ack(ser_k(G_i)))$ is restricted to any unmarked operation in the insert queue for s_k and the first operation in the insert queue for s_k (since $cond(ser_k(G_i))$ for a marked operation $ser_k(G_i)$ holds only if it is first in the queue for s_k).

  Note that $wait(ack(ser_k(G_i)))$ specified above does not contain all the unmarked operations $ser_k(G_l) \in$ WAIT, even though by definition, all of them must be included in

-Appendix B-

Before we show that Scheme 1 ensures the serializability of $ser(S)$, we prove the following lemma.

Lemma 1: For all transactions $\widehat{G}_i, \widehat{G}_j$, for all sites s_k, if $ser_k(G_i)$ executes before $ser_k(G_j)$, then $ser_k(G_i)$ is inserted into the delete queue for site s_k before $ser_k(G_j)$ is inserted into the delete queue for site s_k.

Proof: We first show that if $ser_k(G_i)$ executes before $ser_k(G_j)$, then $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$. Suppose $act(ser_k(G_j))$ executes before $act(ser_k(G_i))$. Then, due to $cond(ser_k(G_i))$, $act(ser_k(G_i))$ executes only after $act(ack(ser_k(G_j)))$ executes. Thus, $ser_k(G_j)$ executes before $ser_k(G_i)$ executes, which leads to a contradiction. As a result, $act(ser_k(G_i))$ executes before $act(ser_k(G_j))$.

Further, due to $cond(ser_k(G_i))$, $act(ser_k(G_j))$ executes only after $act(ack(ser_k(G_i)))$. Thus, since $act(ack(ser_k(G_j)))$ executes after $act(ser_k(G_j))$, $act(ack(ser_k(G_j)))$ executes after $act(ack(ser_k(G_i)))$. Since operation $ser_k(G_i)$ is inserted into the delete queue for s_k when $act(ack(ser_k(G_i)))$ executes, $ser_k(G_i)$ is inserted into the delete queue for s_k before $ser_k(G_j)$ is inserted into the delete queue for s_k. □

Proof of Theorem 3: Let us assume that $ser(S)$ is not serializable. Thus, there exist distinct transactions, say, $\widehat{G}_1, \widehat{G}_2, \ldots, \widehat{G}_r$, $r > 1$, such that $ser_{i_1}(G_1)$ executes before $ser_{i_1}(G_2)$, $ser_{i_2}(G_2)$ executes before $ser_{i_2}(G_3)$, …, $ser_{i_r}(G_r)$ executes before $ser_{i_r}(G_1)$ and for all $j, k = 1, 2, \ldots, r$, $j \neq k$, $i_j \neq i_k$ (since for any s_k, transaction $\widehat{G}_j$ has at most one operation $ser_k(G_j)$). We first show that for all j, $j = 1, 2, \ldots, r$, $act(fin_j)$ cannot execute. By Lemma 1, $ser_{i_j}(G_j)$ is inserted into the delete queue for s_{i_j} before $ser_{i_j}(G_{(j \bmod r)+1})$ is inserted into the delete queue for s_{i_j}. Since, for $cond(fin_{(j \bmod r)+1})$ to hold, $ser_{i_j}(G_{(j \bmod r)+1})$ must be first in the delete queue for s_{i_j}, $ser_{i_j}(G_j)$ must be deleted from delete queue for s_{i_j} before $act(fin_{(j \bmod r)+1})$ can execute. Thus, for all j, $j = 1, 2, \ldots, r$, $act(fin_j)$ must execute before $act(fin_{(j \bmod r)+1})$ can execute. Thus, from above, if $act(fin_k)$ executes for some k, $k = 1, 2, \ldots, r$, then $act(fin_k)$ executes before $act(fin_k)$ executes, which is not possible. Thus, for all j, $j = 1, 2, \ldots, r$, $act(fin_j)$ does not execute.

Thus, after $act(init_j)$, for all $j = 1, 2, \ldots, r$, execute, there is a cycle $(\widehat{G}_1, s_{i_1})(s_{i_1}, \widehat{G}_2)(\widehat{G}_2, s_{i_2}) \cdots (\widehat{G}_r, s_{i_r})(s_{i_r}, \widehat{G}_1)$ in the TSG, since for all j, $j = 1, 2, \ldots, r$, $act(fin_j)$ does not execute, and thus, $\widehat{G}_j$'s edges are not deleted from the TSG. Let $act(init_j)$ execute last among $act(init_1)$, $act(init_2), \ldots, act(init_r)$. Thus, $\widehat{G}_j$'s edges are inserted into the TSG last among $\widehat{G}_1, \widehat{G}_2, \ldots, \widehat{G}_r$. Since the insertion of $\widehat{G}_j$'s edges into the TSG causes a cycle involving edge $(\widehat{G}_j, s_{i_j})$, $ser_{i_j}(G_j)$ is marked. Also, since $act(init_{(j \bmod r)+1})$ executes before $act(init_j)$ executes, $ser_{i_j}(G_{(j \bmod r)+1})$ is inserted into the insert queue for s_{i_j} before $ser_{i_j}(G_j)$ is inserted into the insert queue for s_{i_j}. Thus, $ser_{i_j}(G_{(j \bmod r)+1})$ executes before $ser_{i_j}(G_j)$ executes. However, this leads to a contradiction since we assumed that $ser_{i_j}(G_j)$ executes before $ser_{i_j}(G_{(j \bmod r)+1})$. Thus, Scheme 1 ensures $ser(S)$ is serializable. □.

Complexity Analysis of Scheme 1:

We first describe additional data structures involved in the implementation of Scheme 1. We then analyze, for every operation o_j, the number of steps in $cond(o_j)$ and $act(o_j)$, and the characteristics of $wait(o_j)$ (the number of operations and their types).

Implementation: For each transaction $\widehat{G}_i$, a counter f_count_i keeps a count of the number of operations belonging to $\widehat{G}_i$ that are not first in the delete queue.

The number of steps in $cond(o_j)$ and $act(o_j)$, for each operation o_j, are as follows.

- $cond(init_i)$: $O(1)$.

Thus, the number of steps in $cond(o_l)$ for every operation $o_l \in wait(ack(ser_k(G_i)))$ is $O(1)$, and the number of operations in $wait(ack(ser_k(G_i)))$ is $O(1)$.

Theorem 14: The complexity of Scheme 0 is $O(d_{av})$.

Proof: The complexity of Scheme 0 is dominated by the number of steps in $act(init_i)$, which is $O(d_{av})$. Thus, the complexity of Scheme 0 is $O(d_{av})$. $\square$

-Appendix A-

Proof of Theorem 1: Let us assume that global schedule S is not serializable. Since each of the local schedules is serializable, there must exist a cycle consisting of global transactions, say, $G_1, G_2, \ldots, G_r$, $r > 1$, such that G_1 is serialized before G_2 at site s_{i_1}, G_2 is serialized before G_3 at site s_{i_2}, …, G_r is serialized before G_1 at site s_{i_r}. We show that if G_j is serialized before G_k at site s_{i_j}, then $G_j \prec_G G_k$. If G_j is serialized before G_k at site s_{i_j}, then by the definition of serialization functions, $ser_{i_j}(G_j) \prec_{S_{i_j}} ser_{i_j}(G_k)$, and thus $G_j \prec_G G_k$. As a result, $G_1 \prec_G G_2 \prec_G \cdots \prec_G G_r \prec_G G_1$, a contradiction, since $\prec_G$ is a total order. Thus, S is serializable. □

Proof of Theorem 2: In order to show that S is serializable, by Theorem 1, it suffices to show that there exists a total order $\prec_G$ on global transactions such that for each site s_k, for all global transactions G_i, G_j such that $s_k \in (exec(G_i) \cap exec(G_j))$, if $ser_k(G_i) \prec_{S_k} ser_k(G_j)$ then $G_i \prec_G G_j$. Since $ser(S)$ is serializable, there exists a total order $\prec_{\widehat{G}}$ on all the transactions $\widehat{G}_i$ such that for all sites s_k, for all global transactions G_i, G_j such that $s_k \in (exec(G_i) \cap exec(G_j))$, if $ser_k(G_i) \prec_{S_k} ser_k(G_j)$ then $\widehat{G}_i \prec_{\widehat{G}} \widehat{G}_j$ (since $ser_k(G_i)$ and $ser_k(G_j)$ are assumed to conflict). Thus, S can be shown to be serializable by defining $\prec_G$ as follows: $G_i \prec_G G_j$ iff $\widehat{G}_i \prec_{\widehat{G}} \widehat{G}_j$. □

Complexity Analysis of Scheme 0:

We first analyze, for every operation o_j, the number of steps in $cond(o_j)$ and $act(o_j)$. We then analyze the characteristics of $wait(o_j)$ (the number of operations and their types).

- $cond(init_i)$: $O(1)$.
- $act(init_i)$: $O(d_{av})$. Since every transaction has d_{av} operations, the number of steps needed to add $\widehat{G}_i$'s operations at the end of the queue for all sites $s_k \in exec(G_i)$ is $O(d_{av})$.
- $(ser_k(G_i))$: $O(1)$. The number of steps required to check if $ser_k(G_i)$ is the first element in the queue for s_k is $O(1)$.
- $act(ser_k(G_i))$: $O(1)$.
- $cond(ack(ser_k(G_i)))$: $O(1)$.
- $act(ack(ser_k(G_i)))$: $O(1)$. Deletion of $ser_k(G_i)$ from the front of the queue for s_k takes $O(1)$ steps.

Since $cond(init_i)$ and $cond(ack(ser_k(G_i)))$ are both *true*, the only operations in WAIT are $ser_k(G_i)$ for some transaction $\widehat{G}_i$ and site $s_k \in exec(G_i)$. Also, execution of $act(o_j)$, for an operation o_j can cause $cond(ser_k(G_i))$ to hold only if $act(o_j)$ deletes all the operations that precede $ser_k(G_i)$ in the queue for s_k thus causing $ser_k(G_i)$ to be the first operation in the queue for s_k.

We now specify $wait(o_j)$ for each of the operations o_j.

- $wait(init_i)$: $\emptyset$. Execution of $act(init_i)$ does not result in the deletion of operations from any of the queues.
- $wait(ser_k(G_i))$: $\emptyset$. Execution of $act(ser_k(G_i))$ does not result in the deletion of operations from any of the queues.
- $wait(ack(ser_k(G_i)))$: $\{ser_k(G_j) : (ser_k(G_j) \in \text{WAIT}) \wedge$ $ser_k(G_j)$ immediately follows $ser_k(G_j)$ in the queue for site $s_k\}$
  Since the execution of $act(ack(ser_k(G_i)))$ causes only $ser_k(G_i)$ to be deleted from the front of queue s_k, the only operation o_l for which $cond(o_l)$ can hold due to the execution of $act(ack(ser_k(G_i)))$ is the operation $ser_k(G_j)$ that immediately follows it in the queue for s_k (since the execution of $act(ack(ser_k(G_i)))$ causes $ser_k(G_j)$ to become the first operation in the queue for s_k).

serializability in a centralized DBMS. Since concurrency control in centralized DBMSs is a well studied problem, the development of concurrency control schemes for MDBSs is simplified.

We have proposed a model for conservative concurrency control schemes, and have developed a number of conservative schemes for ensuring serializability, including one that permits the set of all serializable schedules. We have analyzed the complexities of each of the developed schemes and compared the degree of concurrency provided by the various schemes. Since conservative schemes delay the execution of operations belonging to transactions instead of aborting transactions later, in our analysis of the complexity of conservative schemes we have taken into account the cost of attempting to reschedule an operation that was previously made to wait. Further work still remains to be done on making the developed schemes fault-tolerant.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BS88] Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago*, pages 135–141, 1988.

[BST90] Y. Breitbart, A. Silberschatz, and G. R. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, New Jersey*, pages 215–224, 1990.

[DE89] W. Du and A. K. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In *Proceedings of the Fifteenth International Conference on Very Large Databases, Amsterdam*, pages 347–355, 1989.

[ED90] A.K. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, 1990.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[GRS91] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan*, 1991.

[MRKS91] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *Proceedings of the First International Conference on Parallel and Distibuted Information Systems, Miami Beach, Florida*, 1991.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Maryland, 1986.

[PRR91] W. Perrizo, J. Rajkumar, and P. Ram. Hydro: A heterogeneous distributed database system. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, Colorado*, pages 32–39, May 1991.

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proceedings of the Fourth International Conference on Data Engineering, Los Angeles*, 1988.

The above definition of starvation-freedom guarantees that if every transaction is of a finite duration, then it is not possible for the processing of operations belonging to a transaction to be delayed indefinitely. Schemes 0, 1 and 2 are starvation-free, while scheme 3 may cause transactions to starve. We illustrate, in the following example, that any scheme which permits all serializable schedules, cannot be starvation-free.

Example 8: Consider an MDBS environment consisting of two sites s_1 and s_2. Let G_1 and G_2 be global transactions such that transactions $\widehat{G}_1$ and $\widehat{G}_2$ are as follows:

$$\widehat{G}_1 : \ ser_2(G_1)\ ser_1(G_1)$$

$$\widehat{G}_2 : \ ser_1(G_2)\ ser_2(G_2)$$

Let GTM$_1$ insert operations into QUEUE in the following order.

$$init_1 \quad init_2 \quad ser_1(G_2) \quad ser_2(G_1) \quad ser_1(G_1) \quad ser_2(G_2) \quad fin_1 \quad fin_2$$

Let CC be a concurrency control scheme that permits all serializable schedules. As a result, after CC processes operations $init_1$ and $init_2$, it must process $ser_1(G_2)$ when it is selected from QUEUE. However, after $ser_1(G_2)$ has been processed, since $ser_2(G_2)$ has not yet been processed, CC cannot process $ser_2(G_1)$ when it is selected from QUEUE (processing $ser_2(G_1)$ when it is selected from QUEUE may cause $ser(S)$ to become non-serializable). Thus, since the processing of $ser_2(G_1)$ is delayed due transaction $\widehat{G}_2$, CC is not starvation-free. □

Scheme 3 can be made starvation-free by adding to $cond(ser_k(G_i))$ the following.

- $\boldsymbol{cond(ser_k(G_i))}$: Let $Set_1 = \{\widehat{G}_i\} \cup ser\_bef(\widehat{G}_i)$, and $Set_2 = \{\widehat{G}_j : (\widehat{G}_j \in (set_k - \widehat{G}_i)) \vee ((ser\_bef(\widehat{G}_j) \cap (set_k - \widehat{G}_i)) \neq \emptyset)\}$. There do not exist transactions $\widehat{G}_q, \widehat{G}_r$ and a set set_p, $set_p \neq set_k$, such that
  1. $\widehat{G}_q, \widehat{G}_r \in set_p$,
  2. $\widehat{G}_q \in Set_1$, $\widehat{G}_r \in Set_2$, and
  3. $init_r$ is processed before $init_q$.

Theorem 12: Scheme 3 with the addition is starvation-free.
Proof: See Appendix D. □

Since Scheme 3 ensures serializability of $ser(S)$, the above scheme, in addition to being starvation-free, also ensures serializability of $ser(S)$. However, Scheme 3 with the addition has a higher complexity than Scheme 3.

Theorem 13: The complexity of Scheme 3 with the addition is $O((mn^2 + n^3)d_{av}^2)$.
Proof: See Appendix D. □

8 Conclusion

There has been no systematic study of the concurrency control problem in MDBS environments. Existing schemes for ensuring global serializability in MDBSs are ad-hoc, and no analysis of their performance, the degree of concurrency provided by them, or their complexity has been made. In this paper, we have identified characteristics of the concurrency control problem and the additional requirements on concurrency control schemes for ensuring global serializability in MDBS environments. We have reduced the problem of developing schemes for ensuring global serializability in an MDBS environment to that of developing conservative schemes for ensuring

Since $ser_k(G_i)$ is not processed if there exists a transaction $\widehat{G}_j$ in both $(set_k - \{\widehat{G}_i\})$ and $ser\_bef(\widehat{G}_i)$ (that is, $\widehat{G}_j$ is serialized before $\widehat{G}_i$), Scheme 3 ensures that for all transactions $\widehat{G}_i$, $\widehat{G}_i \notin ser\_bef(\widehat{G}_i)$, and thus, $\widehat{G}_i$ is never serialized before itself. We illustrate the execution of Scheme 3 assuming that operations are inserted into QUEUE by GTM$_1$ in the order mentioned in Example 2. In Example 2, after $init_1$ and $init_2$ are processed by Scheme 3, both set_1 and set_2 are $\{\widehat{G}_1, \widehat{G}_2\}$, and $ser\_bef(\widehat{G}_1)$, $ser\_bef(\widehat{G}_2)$ are both $\emptyset$. Further, when $ser_2(G_2)$ is selected from QUEUE by Scheme 3, after $ser_1(G_1)$ has been processed, $cond(ser_2(G_2))$ does not hold (since processing of $ser_1(G_1)$ results in $ser\_bef(\widehat{G}_2) = \{\widehat{G}_1\}$ and $set_2 - \{\widehat{G}_2\} = \{\widehat{G}_1\}$) and $ser_2(G_2)$ is not processed. Thus, Scheme 3 does not permit the non-serializable schedule in Example 2.

Theorem 8: Scheme 3 ensures that $ser(S)$ is serializable.
Proof: See Appendix D. □

The complexity of Scheme 3 is dominated by the number of steps required in order to update $ser\_bef(\widehat{G}_j)$, for certain transactions $\widehat{G}_j$, when $act(ser_k(G_i))$ executes.

Theorem 9: The complexity of Scheme 3 is $O(n^2 d_{av})$.
Proof: See Appendix D. □

Furthermore, it can be shown that Scheme 3 ensures progress in the processing of operations.

Theorem 10: If for some set set_k, during the execution of Scheme 3, $set_k \neq \emptyset$, then for some transaction $\widehat{G}_i \in set_k$, $act(ser_k(G_i))$ is executed by Scheme 3.
Proof: See Appendix D. □

Corollary 1: If the size of transactions is finite and if only a finite number of them are initiated, then every transaction completes execution.
Proof: See Appendix D. □

Scheme 3 can also be shown to permit all serializable schedules, which is defined as follows. Operations $ser_k(G_i)$ are said to be inserted into QUEUE by GTM$_1$ in a *serializable order* if processing every $ser_k(G_i)$ operation when it is selected from QUEUE results in a serializable schedule.

Definition 3: Let GTM$_1$ insert $ser_k(G_i)$ operations into QUEUE in a serializable order. A concurrency control scheme CC is said to permit all serializable schedules if every $ser_k(G_i)$ operation is processed by CC when it is selected from QUEUE (that is, CC does not add any $ser_k(G_i)$ operation to WAIT). □

Theorem 11: Scheme 3 permits permits all serializable schedules.
Proof: See Appendix D. □

Thus, Scheme 3 permits a higher degree of concurrency than Schemes 0, 1 and 2. However, even though Scheme 3 ensures progress in the processing of operations and permits all serializable schedules, it does not guarantee freedom from starvation.

Definition 4: A concurrency control scheme is *starvation-free* if the following holds:
For all pairs of transactions $\widehat{G}_i, \widehat{G}_j$, if $act(init_i)$ executes before $act(init_j)$, then for all operations $ser_k(G_i) \in \widehat{G}_i$, processing of $ser_k(G_i)$ is not delayed (that is, $cond(ser_k(G_i))$ does not hold) due to transaction $\widehat{G}_j$. □

Furthermore, BT-schemes that attempt to provide even a moderately high degree of concurrency are intractable, as shown in the previous section.

In this section, we present an O-scheme that permits the set of serializable schedules, which we refer to as Scheme 3. Scheme 3 adds restrictions on the processing of $\widehat{G}_j$'s operations, to the data structures, every time an $init_i$ or $ser_k(G_i)$ operation is processed. As a result, when an $init_i$ or a $ser_k(G_i)$ operation is processed, Scheme 3 only adds minimum restrictions to the data structures such that processing the next $ser_k(G_i)$ operation cannot cause $ser(S)$ to be non-serializable (additional restrictions are added when the next $ser_k(G_i)$ operation is processed). Since, at any point, minimum restrictions are imposed on the processing of operations in order to ensure serializability of $ser(S)$, Scheme 3 permits the set of all possible serializable schedules. For instance, if GTM$_1$ inserts operations into QUEUE in the order mentioned in Example 7, then Scheme 3 processes every operation when it is selected from QUEUE. Further, the computation of the minimum restrictions to be added to the data structures every time an operation is processed is not too difficult, and Scheme 3 can be shown to have a complexity $O(n^2 d_{av})$.

In scheme 3, associated with every transaction $\widehat{G}_i$ is a set $ser\_bef(\widehat{G}_i)$ of transactions such that if $\widehat{G}_j \in ser\_bef(\widehat{G}_i)$, then $\widehat{G}_j$ is serialized before $\widehat{G}_i$ in $ser(S)$. Also, at any point p during the execution of Scheme 3, for every site s_k,

- $last_k$ is the transaction $\widehat{G}_i$ that is last among transactions in $\{\widehat{G}_j : ser_k(G_j) \in \widehat{G}_j\}$ to have executed $act(ser_k(G_i))$ before point p.
- set_k is the set of transactions $\{\widehat{G}_j : (ser_k(G_j) \in \widehat{G}_j) \wedge (act(init_j)$ has executed before $p) \wedge$ ($act(ser_k(G_j))$ has not executed before $p)\}$.

Initially, for all s_k, $last_k = null$, $set_k = \emptyset$, and for all $\widehat{G}_i$, $ser\_bef(\widehat{G}_i) = \emptyset$. For an operation o_j in QUEUE, $cond(o_j)$ and $act(o_j)$ are defined as follows.

- $\boldsymbol{cond(init_i)}$: $true$.
- $\boldsymbol{act(init_i)}$: For every operation $ser_k(G_i) \in \widehat{G}_i$, $\widehat{G}_i$ is added to set_k. The set $ser\_bef(\widehat{G}_i)$ is updated as follows to include all the transactions serialized before $\widehat{G}_i$.

$$ser\_bef(\widehat{G}_i) := \cup_{ser_k(G_i) \in \widehat{G}_i \wedge last_k \neq null} (ser\_bef(last_k) \cup \{last_k\}).$$

- $\boldsymbol{cond(ser_k(G_i))}$: $ser\_bef(\widehat{G}_i) \cap (set_k - \{\widehat{G}_i\}) = \emptyset$. If $last_k = \widehat{G}_j$, then $act(ack(ser_k(G_j)))$ has executed.
- $\boldsymbol{act(ser_k(G_i))}$: $\widehat{G}_i$ is deleted from set_k and $last_k$ is set to $\widehat{G}_i$. Since for all transactions $\widehat{G}_j \in set_k$, $ser_k(G_j)$ has not been processed when $ser_k(G_i)$ is processed, $ser_k(G_i)$ executes before $ser_k(G_j)$ executes, and $\widehat{G}_i$ is thus serialized before $\widehat{G}_j$ in $ser(S)$. Thus, for certain transactions $\widehat{G}_j$, $ser\_bef(\widehat{G}_j)$ is updated as follows to include all the transactions serialized before $\widehat{G}_j$. Let $Set_1 = (ser\_bef(\widehat{G}_i) \cup \{\widehat{G}_i\})$, $Set_2 = \{\widehat{G}_l : ser\_bef(\widehat{G}_l) \cap set_k \neq \emptyset\}$.
  For all transactions $\widehat{G}_j$ such that either $\widehat{G}_j \in set_k$, or $\widehat{G}_j \in Set_2$,

$$ser\_bef(\widehat{G}_j) := ser\_bef(\widehat{G}_j) \cup Set_1.$$

  Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.

- $\boldsymbol{cond(ack(ser_k(G_i)))}$: $true$.
- $\boldsymbol{act(ack(ser_k(G_i)))}$: Operation $ack(ser_k(G_i))$ is sent to GTM$_1$.

- $\boldsymbol{cond(fin_i)}$: $ser\_bef(\widehat{G}_i) = \emptyset$.
- $\boldsymbol{act(fin_i)}$: For all transactions $\widehat{G}_j$ such that $\widehat{G}_i \in ser\_bef(\widehat{G}_j)$, $\widehat{G}_i$ is deleted from $ser\_bef(\widehat{G}_j)$. For all $ser_k(G_i) \in \widehat{G}_i$ such that $last_k = \widehat{G}_i$, $last_k := null$.

Example 6: Consider an MDBS environment consisting of four sites s_1, s_2, s_3 and s_4. Let G_1, G_2, G_3 and G_4 be global transactions such that transactions $\widehat{G}_1$, $\widehat{G}_2$, $\widehat{G}_3$ and $\widehat{G}_4$ are as follows.

$$\widehat{G}_1 : \ ser_2(G_1)\ ser_1(G_1)$$

$$\widehat{G}_2 : \ ser_2(G_2)\ ser_3(G_2)$$

$$\widehat{G}_3 : \ ser_3(G_3)\ ser_4(G_3)$$

$$\widehat{G}_4 : \ ser_1(G_4)\ ser_4(G_4)$$

Let GTM$_1$ insert operations into QUEUE in the order

$$init_1 \ \ init_2 \ \ init_3 \ \ init_4 \ \ ser_2(G_2) \ \ ser_3(G_2) \ \ ser_1(G_4) \ \ ser_4(G_4) \ \ ser_2(G_1) \ \ldots\ldots$$

After $init_1$, $init_2$, $init_3$ and $init_4$ have been processed, Scheme 2 adds dependencies $(\widehat{G}_3, s_4)\rightarrow(\widehat{G}_4, s_4)$ and $(\widehat{G}_1, s_1)\rightarrow(\widehat{G}_4, s_1)$ to the TSGD. Operations $ser_2(G_2)$ and $ser_3(G_2)$ belonging to $\widehat{G}_2$ are then processed, and the following additional dependencies are added to the TSGD by Scheme 2: $(\widehat{G}_2, s_2)\rightarrow(\widehat{G}_1, s_2)$ and $(\widehat{G}_2, s_3)\rightarrow(\widehat{G}_3, s_3)$. Note that as a result of the processing of $\widehat{G}_2$'s operations, the dependencies $(\widehat{G}_3, s_4)\rightarrow(\widehat{G}_4, s_4)$ and $(\widehat{G}_1, s_1)\rightarrow(\widehat{G}_4, s_1)$ prevent $\widehat{G}_4$'s operations from being processed even though these restrictions are unnecessary, since $\widehat{G}_2$ is serialized before $\widehat{G}_1$ and $\widehat{G}_3$, and thus processing the remaining operations belonging to transactions $\widehat{G}_1$, $\widehat{G}_3$ and $\widehat{G}_4$ in an arbitrary order cannot cause $ser(S)$ to become non-serializable. □

Thus, a greater degree of concurrency could be obtained if Scheme 2 deleted unnecessary dependencies from D every time it processed an $init_i$ operation. However, in a TSGD (V, E, D), determining the set of dependencies that are unnecessary is an NP-hard problem (follows from Theorem 7).

7 The Scheme that Permits all Serializable Schedules

The problem with the BT-schemes presented in the previous sections is that they either provide a low degree of concurrency or have high complexity. This is due to the requirement that all the restrictions on the processing of $\widehat{G}_i$'s operations in order to ensure serializability of $ser(S)$ be added to the data structures when $init_i$ is processed (since no restrictions are added when $ser_k(G_i)$ operations are processed). Thus, BT-schemes cannot provide a very high degree of concurrency since they *a priori* restrict the processing of operations to permit only a subset of serializable schedules.

Example 7: Consider an MDBS environment consisting of two sites s_1 and s_2. Let G_1 and G_2 be global transactions such that $\widehat{G}_1$ and $\widehat{G}_2$ are as follows:

$$\widehat{G}_1 : \ ser_2(G_1) \ \ ser_1(G_1)$$

$$\widehat{G}_2 : \ ser_2(G_2) \ \ ser_1(G_2)$$

Let GTM$_1$ insert operations into QUEUE in the order

$$init_1 \ \ init_2 \ \ ser_2(G_2) \ \ ser_1(G_2) \ \ ser_2(G_1) \ \ ser_1(G_1) \ \ fin_1 \ \ fin_2$$

All of the BT-schemes discussed earlier, when they select $ser_2(G_2)$ from QUEUE do not process $ser_2(G_2)$ (since processing of $ser_2(G_2)$ is restricted to follow the processing of $ser_2(G_1)$ by every scheme) even though processing every operation when it is selected from QUEUE cannot cause $ser(S)$ to be non-serializable. □

be shown by a simple induction argument on the number of $init_i$ operations processed, that the TSGD is always acyclic, and thus $ser(S)$ is serializable. As a result, Scheme 2 does not permit the non-serializable schedule in Example 2, even if GTM$_1$ inserts operations into QUEUE in the order mentioned in Example 2. In Example 2, when $init_2$ is processed by Scheme 2 after $init_1$ has been processed, dependencies $(\widehat{G}_1, s_1) \rightarrow (s_1, \widehat{G}_2)$ and $(\widehat{G}_1, s_2) \rightarrow (s_2, \widehat{G}_2)$ are added to D due to procedure Eliminate_cycles (since the TSGD, after the insertion of $\widehat{G}_2$'s edges contains the edges $(\widehat{G}_1, s_1)$, $(s_1, \widehat{G}_2)$, $(\widehat{G}_2, s_2)$ and $(s_2, \widehat{G}_1)$). Thus, when operation $ser_2(G_2)$ is selected from QUEUE by Scheme 2, after $ser_1(G_1)$ has been processed, $cond(ser_2(G_2))$ does not hold (since dependency $(\widehat{G}_1, s_2) \rightarrow (s_2, \widehat{G}_2) \in D$) and thus $ser_2(G_2)$ is not processed.

Theorem 5: Scheme 2 ensures that $ser(S)$ is serializable.
Proof: See Appendix C. □

The number of steps in Eliminate_Cycles dominates the complexity of Scheme 2. It can be shown that Eliminate_Cycles terminates in $O(n^2 d_{av})$ steps.

Theorem 6: The complexity of Scheme 2 is $O(n^2 d_{av})$.
Proof: See Appendix C. □

Scheme 2 provides a higher degree of concurrency than Scheme 0. Also, if GTM$_1$ inserts operations into QUEUE in the order mentioned in Example 5, Scheme 2, unlike Scheme 1, does not impose any restrictions on the processing of operations and processes every operation when it is selected from QUEUE. However, Scheme 2 does not provide a higher degree of concurrency than Scheme 1 since certain dependencies in the set of dependencies Δ returned by Eliminate_Cycles may be unnecessary for the purpose of ensuring that $(V, E, D \cup \Delta)$ contains no cycles involving $\widehat{G}_i$. Thus, there may exist a set of dependencies, Δ_1, such that $\Delta_1 \subset \Delta$ and $(V, E, D \cup \Delta_1)$ does not contain a cycle involving $\widehat{G}_i$. We formally define the notion of *minimality* as follows.

Definition 2: A set of dependencies Δ is minimal with respect to the TSGD and a transaction $\widehat{G}_i \in V$ iff

- $(V, E, D \cup \Delta)$ does not contain any cycles involving $\widehat{G}_i$, and
- for all $d \in \Delta$, $(V, E, D \cup \Delta - d)$ contains a cycle involving $\widehat{G}_i$. □

The set of dependencies Δ returned by Eliminate_Cycles may not be minimal with respect to (V, E, D) and $\widehat{G}_i$, and thus unnecessary restrictions may be imposed on the processing of operations, hurting the degree of concurrency. In order to impose minimal restrictions on the processing of operations and to provide maximal concurrency without jeopardizing the serializability of $ser(S)$, Δ must be minimal with respect to (V, E, D) and $\widehat{G}_i$. However, the problem of computing such a Δ is NP-hard [GJ79].

Theorem 7: Given a TSGD (V, E, D), and a transaction node $\widehat{G}_i \in V$ in the TSGD such that for all transactions $\widehat{G}_j \in V$, for all sites s_k, dependency $(\widehat{G}_i, s_k) \rightarrow (\widehat{G}_j, s_k) \notin D$. Also, TSGD (V', E', D') resulting from the deletion of $\widehat{G}_i$, its edges and dependencies from (V, E, D), is acyclic. The problem of computing a set of dependencies, Δ, that is minimal with respect to (V, E, D) and transaction $\widehat{G}_i$ is NP-hard.
Proof: See Appendix C. □

Note that, in Scheme 2, dependencies are only added to D. This could affect the degree of concurrency since the order in which operations are processed may make certain dependencies that were previously added to D unnecessary. This is illustrated by the following example.

```
procedure Eliminate_Cycles($(V, E, D), \widehat{G}_i$):
  1. Mark all edges "unused".
     $v := \widehat{G}_i$, $\Delta := \emptyset$, $s\_par(\widehat{G}_i) := null$, $t\_par(\widehat{G}_i) := null$.
  2. If for all pairs of edges $(v, u), (u, w)$, either
       • $w \neq \widehat{G}_i$ and $(u, w)$ is marked "used", or
       • there is a dependency $(v, u) \rightarrow (u, w)$ in $D \cup \Delta$, or
       • $head(s\_par(v)) = u$.
     then go to step (4).
  3. Choose a pair of edges $(v, u), (u, w)$ such that
       • $w = \widehat{G}_i$ or $(u, w)$ is not marked "usedv", and
       • there is no dependency $(v, u) \rightarrow (u, w)$ in $D \cup \Delta$, and
       • $head(s\_par(v)) \neq u$.
     Mark $(u, w)$ "used".
     If $w = \widehat{G}_i$, then add to $\Delta$ the dependency $(v, u) \rightarrow (u, \widehat{G}_i)$.
     If $w \neq \widehat{G}_i$, then $s\_par(w) := u \circ s\_par(w)$, $t\_par(w) := v \circ t\_par(w)$, and $v := w$.
     Go to step (2).
  4. If $v \neq \widehat{G}_i$, then begin $temp := head(t\_par(v))$; $t\_par(v) := tail(t\_par(v))$; $s\_par(v) := tail(s\_par(v))$;
     $v := temp$; go to step (2) end.
  5. return($\Delta$).
```

Figure 5: The procedure Eliminate_Cycles

- Every dependency in Δ is of the form $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$, for some transaction $\widehat{G}_j \in V$ and some site $s_k \in V$.
- In $(V, E, D \cup \Delta)$ there are no cycles involving $\widehat{G}_i$.

Eliminate_Cycles attempts to detect cycles involving $\widehat{G}_i$ in the TSGD, and then eliminates them by adding dependencies to Δ. It traverses edges in the TSGD "marking" them as it goes along so that an edge is not traversed multiple times. If an edge incident on $\widehat{G}_i$ is traversed, then Eliminate_Cycles concludes that there is a cycle involving $\widehat{G}_i$ and adds appropriate dependencies to Δ in order to eliminate the cycle.

In Eliminate_Cycles, v is the current transaction node being visited (site nodes are not visited). Unlike depth-first search[AHU74], in Eliminate_Cycles, a transaction node may be visited multiple times. For a transaction node $\widehat{G}_j$, $t\_par(\widehat{G}_j)$ stores the list of transaction nodes to which backtracking from $\widehat{G}_j$ must take place, and $s\_par(\widehat{G}_j)$, the list of site nodes from which $\widehat{G}_j$ is visited, every time it is visited. Functions *head*, *tail* and $\circ$ are as defined for lists[4].

Eliminate_Cycles returns a set of dependencies Δ such that $(V, E, D \cup \Delta)$ contains no cycles involving $\widehat{G}_i$. Since Eliminate_Cycles is invoked every time an $init_i$ operation is processed, it can

---

[4] For a list $l = [l_1, l_2, \ldots, l_p]$ and element l_0, $head(l)$ returns l_1, $tail(l)$ returns $[l_2, \ldots, l_p]$, and $l_0 \circ l$ returns $[l_0, l_1, l_2, \ldots, l_p]$.

Acyclicity of the TSGD plays an important role in ensuring that *ser*(S) is serializable. Below, we formally state the conditions under which a TSGD is said to be *acyclic*.

Definition 1: Consider a TSGD containing edges (v_1, v_2), (v_2, v_3), ..., (v_k, v_1), $k > 2$. This set of edges form a *cycle* if $v_i \neq v_j$, for all $i, j = 1, 2, \ldots, k$, $i \neq j$, and either one of the following is true.

- For all i, $i = 2, 3, \ldots, k$, dependency $(v_{i-1}, v_i) \rightarrow (v_i, v_{(i \bmod k)+1}) \notin D$.
- For all i, $i = 2, 3, \ldots, k$, dependency $(v_{(i \bmod k)+1}, v_i) \rightarrow (v_i, v_{i-1}) \notin D$.

We say the TSGD is acyclic if it does not contain any cycles. □

Scheme 2, specified below, ensures that *ser*(S) is serializable by ensuring that the TSGD is acyclic; that is, if the TSGD contains edges, say, $(\widehat{G}_1, s_{i_1})$, $(s_{i_1}, \widehat{G}_2)$, $(\widehat{G}_2, s_{i_2})$, ... $(\widehat{G}_r, s_{i_r})$, $(s_{i_r}, \widehat{G}_1)$, for distinct transactions $\widehat{G}_1, \widehat{G}_2, \ldots, \widehat{G}_r$, $r > 1$, and $i_p \neq i_q$, $p \neq q$, then the TSGD also contains dependencies, $(\widehat{G}_j, s_{i_j}) \rightarrow (s_{i_j}, \widehat{G}_{(j \bmod r)+1})$ and $(\widehat{G}_{(k \bmod r)+1}, s_{i_k}) \rightarrow (s_{i_k}, \widehat{G}_k)$. The scheme also ensures that for the above dependencies, $j \neq k$, $ser_{i_j}(G_j)$ is processed before $ser_{i_j}(G_{(j \bmod r)+1})$ and $ser_{i_k}(G_{(k \bmod r)+1})$ is processed before $ser_{i_k}(G_k)$. As a result, Scheme 2 ensures that there is no cycle in the serialization graph of *ser*(S) involving transactions $\widehat{G}_1, \widehat{G}_2, \ldots, \widehat{G}_r$ due to the operations $ser_{i_j}(G_j)$ and $ser_{i_j}(G_{(j \bmod r)+1})$, $j = 1, 2, \ldots, r$.

We now specify, for every operation o_j in QUEUE, *cond*(o_j) and *act*(o_j) that preserve the acyclicity of the TSGD. Initially, for the TSGD, $V = \emptyset$, $E = \emptyset$, $D = \emptyset$.

- ***cond*$(init_i)$**: *true*.
- ***act*$(init_i)$**: $\widehat{G}_i$ and its edges are inserted into the TSGD. For every operation $ser_k(G_i) \in \widehat{G}_i$, for all transactions $\widehat{G}_j \in V$ such that $ser_k(G_j) \in \widehat{G}_j$ and *act*$(ser_k(G_j))$ has executed, dependencies $(\widehat{G}_j, s_k) \rightarrow (s_k, \widehat{G}_i)$ are added to D. The set of dependencies, D, is further modified as follows.

  $$D := D \cup \text{Eliminate\_Cycles}((V, E, D), \widehat{G}_i)$$

  The procedure Eliminate_Cycles (specified in Figure 5) returns a set of dependencies Δ such that $(V, E, D \cup \Delta)$ does not contain any cycles involving $\widehat{G}_i$.

- ***cond*$(ser_k(G_i))$**: For all transactions $\widehat{G}_j \in V$, if dependency $(\widehat{G}_j, s_k) \rightarrow (s_k, \widehat{G}_i) \in D$, then *act*$(ack(ser_k(G_j)))$ has completed execution.
- ***act*$(ser_k(G_i))$**: For every transaction $\widehat{G}_j \in V$ such that $ser_k(G_j) \in \widehat{G}_j$ and *act*$(ser_k(G_j))$ has not yet been executed, dependencies $(\widehat{G}_i, s_k) \rightarrow (s_k, \widehat{G}_j)$ are added to D. Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.

- ***cond*$(ack(ser_k(G_i)))$**: *true*.
- ***act*$(ack(ser_k(G_i)))$**: Operation $ack(ser_k(G_i))$ is sent to GTM$_1$.

- ***cond*(fin_i)**: For every operation $ser_k(G_i) \in \widehat{G}_i$, there does not exist a $\widehat{G}_j \in V$ such that $(\widehat{G}_j, s_k) \rightarrow (s_k, \widehat{G}_i) \in D$.
- ***act*(fin_i)**: $\widehat{G}_i$, along with its edges and dependencies is deleted from the TSGD.

We now discuss the procedure Eliminate_Cycles that takes as arguments the TSGD and a transaction $\widehat{G}_i \in V$. Eliminate_Cycles exploits the knowledge of the order of in which operations are processed and returns a set of dependencies Δ with the following properties.

- ***act*(*init*$_i$)**: $\widehat{G}_i$ and its edges are inserted into the TSG. Also, for every operation $ser_k(G_i) \in \widehat{G}_i$, $ser_k(G_i)$ is inserted at the end of the insert queue for site s_k. If the TSG contains a cycle, then all of $\widehat{G}_i$'s operations in the insert queues for the sites are marked.

Since the number of steps required in order to determine if an undirected graph is acyclic is proportional to the number of nodes in the graph, the simplified version of Scheme 1 has complexity $O(m+n)$ (the TSG has at most $m+n$ nodes). Further, if $m \ll n$, then since the TSG is a bipartite graph, at most m nodes can be visited before a cycle is detected. Thus, the complexity of the simplified scheme reduces to $O(m)$. The above simplified scheme ensures $ser(S)$ is serializable, but provides a lower degree of concurrency than Scheme 1.

6 The Transaction-site Graph-with-dependencies Scheme

Scheme 1, presented in the previous section, does not exploit the knowledge of the order in which operations are processed (the TSG is checked only for cycles). As a result, Scheme 1 places unnecessary restrictions on the processing of operations as illustrated in the example below.

Example 5: Consider an MDBS environment consisting of four sites s_1, s_2, s_3 and s_4. Let G_1, G_2, G_3 and G_4 be global transactions such that transactions $\widehat{G}_1$, $\widehat{G}_2$, $\widehat{G}_3$ and $\widehat{G}_4$ are as follows.

$$\widehat{G}_1 : \; ser_2(G_1) \;\; ser_1(G_1)$$

$$\widehat{G}_2 : \; ser_2(G_2) \;\; ser_3(G_2)$$

$$\widehat{G}_3 : \; ser_3(G_3) \;\; ser_4(G_3)$$

$$\widehat{G}_4 : \; ser_1(G_4) \;\; ser_4(G_4)$$

Let GTM$_1$ insert operations into QUEUE in the order

$$init_1 \;\; init_2 \;\; init_3 \;\; ser_2(G_2) \;\; ser_3(G_2) \;\; init_4 \;\; ser_1(G_4) \;\; ser_4(G_4) \;\; ser_2(G_1) \;\ldots\ldots$$

After $init_1$, $init_2$ and $init_3$ have been processed, Scheme 1 processes $ser_2(G_2)$ and $ser_3(G_2)$ when they are selected from QUEUE (insertion of edges belonging to $\widehat{G}_1, \widehat{G}_2$ and $\widehat{G}_3$ into the TSG does not cause cycles in the TSG, and thus none of their operations are marked). However, when $init_4$ is processed by Scheme 1, since insertion of $\widehat{G}_4$'s edges into the TSG causes a cycle in the TSG, Scheme 1 marks operations $ser_1(G_4)$ and $ser_4(G_4)$ in the insert queues for s_1 and s_4 respectively, thus restricting them to be processed after operations $ser_1(G_1)$ and $ser_4(G_3)$ have been processed. These restrictions are, however, unnecessary since $\widehat{G}_2$ is serialized before $\widehat{G}_1$ and $\widehat{G}_3$, and thus processing the remaining operations belonging to transactions $\widehat{G}_1$, $\widehat{G}_3$ and $\widehat{G}_4$ in an arbitrary order cannot cause $ser(S)$ to be non-serializable. $\square$

The transaction-site graph-with-dependencies scheme, referred to in the sequel as Scheme 2, is presented below and exploits the knowledge of the order in which operations are processed. In order to permit schedules not permitted by Scheme 1, Scheme 2 utilizes a structure similar to the TSG. The structure contains, in addition to transaction and site nodes, *dependencies* (denoted by $\rightarrow$) between edges incident on a common site node, and is referred to as *Transaction Site Graph with Dependencies* (TSGD). A TSGD is a 3-tuple (V, E, D), where V is the set of transaction and site nodes, E is the set of edges and D is the set of dependencies. Dependencies specify the relative order in which operations are processed and are used to restrict the processing of operations. If $(\widehat{G}_i, s_k)$ and $(s_k, \widehat{G}_j)$ are edges in the TSGD, then a dependency of the form $(\widehat{G}_i, s_k)\rightarrow(s_k, \widehat{G}_j)$ denotes that $ser_k(G_i)$ is processed before $ser_k(G_j)$.

cycle $(\widehat{G}_1, s_1)(s_1, \widehat{G}_2)(\widehat{G}_2, s_2)(s_2, \widehat{G}_1)$. Further, since $init_1$ is processed before $init_2$, $\widehat{G}_1$'s operations are inserted into the insert queues for sites s_1 and s_2 before $\widehat{G}_2$'s operations. Thus, when operation $ser_2(G_2)$ is selected from QUEUE by Scheme 1, after $ser_1(G_1)$ has been processed, $cond(ser_2(G_2))$ does not hold (since $ser_2(G_1)$ is in front of $ser_2(G_2)$ in the insert queue for s_2 and $ser_2(G_2)$ is marked) and thus $ser_2(G_2)$ is not processed.

Theorem 3: Scheme 1 ensures that $ser(S)$ is serializable.
Proof: See Appendix B. □

Note that, in Scheme 1, it is essential that for $cond(fin_i)$ to hold, all of $\widehat{G}_i$'s operations must be at the front of the delete queues for the sites, else $ser(S)$ may not be serializable. This is illustrated by the following example.

Example 4: Consider an MDBS environment consisting of two sites s_1 and s_2. Let G_1, G_2 and G_3 be global transactions such that transactions $\widehat{G}_1$, $\widehat{G}_2$ and $\widehat{G}_3$ are as follows.

$$\widehat{G}_1: \quad ser_2(G_1) \;\; ser_1(G_1)$$

$$\widehat{G}_2: \quad ser_2(G_2) \;\; ser_3(G_2)$$

$$\widehat{G}_3: \quad ser_3(G_3) \;\; ser_1(G_3)$$

Let GTM$_1$ insert operations into QUEUE in the order

$$init_1 \;\; init_2 \;\; ser_2(G_1) \;\; ser_2(G_2) \;\; ser_3(G_2) \;\; fin_2 \;\; init_3 \;\; ser_3(G_3) \;\; ser_1(G_3) \;\; ser_1(G_1) \;\; fin_1 \;\; fin_3$$

Operations $init_1$, $init_2$, $ser_2(G_1)$, $ser_2(G_2)$ and $ser_3(G_2)$ are processed by Scheme 1 when they are selected from QUEUE (since insertion of $\widehat{G}_1$'s and $\widehat{G}_2$'s edges into the TSG does not result in a cycle, no operations are marked). Since $ser_2(G_1)$ is processed before $ser_2(G_2)$ by Scheme 1, $ser_2(G_1)$ is inserted into the delete queue for s_2 before $ser_2(G_2)$ is inserted. If fin_2 is processed by Scheme 1 when it is selected from QUEUE even though $ser_2(G_2)$ is not the first operation in the delete queue for s_2, then since $\widehat{G}_2$'s edges are deleted from the TSG, the insertion of $\widehat{G}_3$'s edges into the TSG, when $init_3$ is processed by Scheme 1, does not result in a cycle. As a result, no operations are marked, and operations $ser_3(G_3)$, $ser_1(G_3)$ and $ser_1(G_1)$ are processed when they are selected from QUEUE, resulting in the following non-serializable schedule $ser(S)$.

$$ser_2(G_1) \;\; ser_2(G_2) \;\; ser_3(G_2) \;\; ser_3(G_3) \;\; ser_1(G_3) \;\; ser_1(G_1) \quad \square$$

Note that Scheme 1 provides a higher degree of concurrency than Scheme 0. In Example 3, Scheme 1 permits operations belonging to transactions $\widehat{G}_1$ and $\widehat{G}_2$ to be processed in any order, since insertion of their edges into the TSG does not cause any cycles. The number of steps required to detect cycles in the TSG dominates the complexity of Scheme 1. Cycles in the TSG can be detected using *depth-first search* [AHU74]. Note that the TSG has at most $m + n$ nodes and at most nd_{av} edges.

Theorem 4: The complexity of Scheme 1 is $O(m + n + nd_{av})$.
Proof: See Appendix B. □

Instead of checking the TSG for cycles involving $\widehat{G}_i$, when $act(init_i)$ executes, Scheme 1 can be simplified by checking the TSG simply for a cycle (which may or may not involve $\widehat{G}_i$). In the simplified version of Scheme 1, $cond(o_j)$ and $act(o_j)$ are as defined for Scheme 1, except $act(init_i)$ is defined as follows.

$$init_1 \;\; init_2 \;\; ser_2(G_2) \;\; ser_2(G_1) \;\; ser_1(G_1) \;\; ser_3(G_2) \;\; fin_1 \;\; fin_2$$

Since Scheme 0 processes $init_1$ before $init_2$, $\widehat{G}_1$'s operations are inserted into the queues for s_1 and s_2 before G_2's operations are inserted into the queues for s_2 and s_3. As a result, when $ser_2(G_2)$ is selected from QUEUE by Scheme 0, it is not processed until $ser_2(G_1)$ has been processed (since $ser_2(G_1)$ is ahead of $ser_2(G_2)$ in the queue for site s_2) even though $ser(S)$ would be serializable irrespective of the order in which $ser_2(G_2)$ and $ser_2(G_1)$ are processed. $\square$

Below we present a scheme, which we refer to as Scheme 1, and that provides a higher degree of concurrency than Scheme 0. It utilizes a data structure similar to the site graph introduced in [BS88], which we refer to as the *transaction-site graph* (TSG). A TSG is an undirected bipartite graph consisting of a set of nodes V corresponding to sites (site nodes) and transactions in $ser(S)$ (transaction nodes), and a set of edges E. Site and transaction nodes are labeled by the corresponding sites and transactions, respectively. Edges in the TSG may be present only between transaction nodes and site nodes. An edge between a transaction node $\widehat{G}_i$ and a site node s_k is present only if operation $ser_k(G_i) \in \widehat{G}_i$, and is denoted by either $(s_k, \widehat{G}_i)$ or $(\widehat{G}_i, s_k)$. The set of edges $\{(\widehat{G}_i, s_k) : ser_k(G_i) \in \widehat{G}_i\}$ are referred to as $\widehat{G}_i$'s edges.

Associated with every site s_k, are two queues : an *insert queue* and a *delete queue*. Initially, all queues are empty, and for the TSG, both $V = \emptyset$ and $E = \emptyset$. Processing of certain operations in the insert queues is constrained by "marking" them. For an operation o_j in QUEUE, *cond*(o_j) and *act*(o_j) are defined as follows:

- $\boldsymbol{cond(init_i)}$: *true*.
- $\boldsymbol{act(init_i)}$: $\widehat{G}_i$ and its edges are inserted into the TSG. Also, for every operation $ser_k(G_i) \in \widehat{G}_i$, $ser_k(G_i)$ is inserted at the end of the insert queue for site s_k. If the TSG contains a cycle involving edge $(\widehat{G}_i, s_k)$, then operation $ser_k(G_i)$ in the insert queue for site s_k is marked.

- $\boldsymbol{cond(ser_k(G_i))}$: For every transaction $\widehat{G}_j$ such that $ser_k(G_j) \in \widehat{G}_j$, if $act(ser_k(G_j))$ has executed, then $act(ack(ser_k(G_j)))$ has also completed execution. In addition, if $ser_k(G_i)$ is marked, then it is the first element in the insert queue for site s_k.
- $\boldsymbol{act(ser_k(G_i))}$: Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.

- $\boldsymbol{cond(ack(ser_k(G_i)))}$: *true*.
- $\boldsymbol{act(ack(ser_k(G_i)))}$: Operation $ser_k(G_i)$ is deleted from the insert queue for site s_k (note that $ser_k(G_i)$ may not be at the front of the insert queue for site s_k), and it is added to the end of the delete queue for site s_k. Operation $ack(ser_k(G_i))$ is sent to GTM$_1$.

- $\boldsymbol{cond(fin_i)}$: For every operation $ser_k(G_i) \in \widehat{G}_i$, $ser_k(G_i)$ is the first element in the delete queue for site s_k.
- $\boldsymbol{act(fin_i)}$: $\widehat{G}_i$ and its edges are deleted from the TSG. For every operation $ser_k(G_i) \in \widehat{G}_i$, $ser_k(G_i)$ is deleted from the delete queue for site s_k.

Scheme 1 permits the TSG to contain cycles, but prevents cycles in the serialization graph of $ser(S)$ by marking operations whose processing may potentially lead to cycles in the serialization graph. Further, processing of a marked operation is delayed until all the operations ahead of it in the insert queue have been processed (note that processing of unmarked operations is not constrained in any way). If Scheme 1 were used, the non-serializable schedule in Example 2 cannot result even if GTM$_1$ inserts operations into QUEUE in the order mentioned in Example 2. In Example 2, when $init_2$ is processed by Scheme 1, after $init_1$ has been processed, $\widehat{G}_2$'s operations in the insert queues for site s_1 and site s_2 are marked since the TSG contains the

- $d_{av} \times$(the number of steps required by CC to process $ack(ser_k(G_i))$), and
- The number of steps required by CC to process fin_i.

Note that every time an operation o_j is processed by CC (that is, $act(o_j)$ is executed), operations $o_l \in$ WAIT for which $cond(o_l)$ holds are processed, too. However, evaluating $cond(o_l)$ for all operations $o_l \in$ WAIT, in order to determine if o_l can be processed is wasteful. Let $wait(o_j)$ denote the set of operations

$$\{o_l : o_l \in \text{WAIT} \wedge \text{execution of } act(o_j) \text{ may cause } cond(o_l) \text{ to hold}\}.$$

As a result, when $act(o_j)$ executes, it suffices to evaluate $cond(o_l)$ only for operations o_l belonging to $wait(o_j)$. Thus, the number of steps required by CC to process an operation o_j is the sum of:

- The number of steps in $cond(o_j)$,
- The number of steps in $act(o_j)$, and
- The number of steps in $cond(o_l)$ for each $o_l \in wait(o_j)$.

Scheme 0 can be shown to have complexity $O(d_{av})$. A detailed analysis of the complexity of Scheme 0 can be found in Appendix A.

In Scheme 0, when $init_i$ is processed, the processing of every operation $ser_k(G_i) \in \widehat{G}_i$ is required to follow the processing of operations ahead of $ser_k(G_i)$ in the queue for s_k. No restrictions on the processing of operations are added when $ser_k(G_i)$ or $ack(ser_k(G_i))$ is processed. We refer to such schemes in which restrictions on the processing of $ser_k(G_i)$ operations are added to DS only when an $init_i$ operation is processed, as *begin transaction schemes* or *BT-schemes.* The schemes proposed [BS88, ED90] are BT-schemes. On the other hand, a scheme in which restrictions on the processing of $ser_k(G_i)$ operations are added every time an $init_i$ or a $ser_k(G_i)$ operation is processed is referred to as an *operation scheme* or *O-scheme.* O-schemes, in general, result in a *higher degree of concurrency* than BT-schemes (A concurrency control scheme, say CC_1, is said to provide a higher degree of concurrency than another concurrency control scheme CC_2 if, for any given order of insertion of operations into QUEUE by GTM_1, CC_2 does not cause a fewer number of operations to be added to WAIT than CC_1). In this paper, we present an O-scheme that permits the set of all serializable schedules. Even though BT-schemes cannot provide a high degree of concurrency, certain BT-schemes (e.g., Scheme 0) are attractive, since they have low complexities compared to O-schemes.

In the following sections, we present two BT-schemes, and an O-scheme. The schemes ensure that $ser(S)$ is serializable. We specify the concurrency control schemes by specifying the data structures maintained by the scheme, and $cond(o_j)$, $act(o_j)$ for the various operations. We also state the complexity of each of the schemes, and compare the degree of concurrency provided by the various schemes. A detailed analysis of the complexity of the schemes and proofs of their correctness can be found in the appendices.

5 The Transaction-site Graph Scheme

Even though Scheme 0 has a low complexity, $O(d_{av})$, it has a serious drawback in that it permits a low degree of concurrency as illustrated by the following example.

Example 3: Consider an MDBS environment consisting of three sites s_1, s_2 and s_3. Let G_1 and G_2 be global transactions such that transactions $\widehat{G}_1$ and $\widehat{G}_2$ are as follows.

$$\widehat{G}_1 : \; ser_1(G_1) \;\; ser_2(G_1)$$

$$\widehat{G}_2 : \; ser_2(G_2) \;\; ser_3(G_2)$$

Let GTM_1 insert operations into QUEUE in the order the various operations, and the data structures associated with the scheme. Thus, a conservative concurrency control scheme can be specified by specifying $cond(o_j)$, $act(o_j)$ for the various operations, and the data structures maintained by the scheme.

We now illustrate, by an example, how our abstraction for the structure of conservative schemes can be used to specify a simple scheme similar to the conservative TO scheme [BHG87], which we refer to as Scheme 0. In the conservative TO scheme, every transaction T_i predeclares its operations and is assigned a timestamp (timestamps are assigned in an increasing order) before any of its operations execute. Further, an operation o_j belonging to a transaction T_i is permitted to execute only if every other operation that conflicts with o_j, and belongs to a transaction with a timestamp lower than T_i's timestamp, has completed its execution. Scheme 0 can be specified as follows using our model for conservative schemes. The data structures maintained by Scheme 0 consist of queues (initially empty), one associated with every site s_k. For an operation o_j in QUEUE, $cond(o_j)$ and $act(o_j)$ are defined as follows.

- $\boldsymbol{cond(init_i)}$: *true*.
- $\boldsymbol{act(init_i)}$: Every operation $ser_k(G_i)$ is inserted at the end of the queue for site s_k.

- $\boldsymbol{cond(ser_k(G_i))}$: Operation $ser_k(G_i)$ is the first operation in the queue for site s_k.
- $\boldsymbol{act(ser_k(G_i))}$: Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.

- $\boldsymbol{cond(ack(ser_k(G_i)))}$: *true*.
- $\boldsymbol{act(ack(ser_k(G_i)))}$: Operation $ser_k(G_i)$ is dequeued from the front of the queue for s_k, and $ack(ser_k(G_i))$ is sent to GTM$_1$.

- $\boldsymbol{cond(fin_i)}$: *true*.

No actions are performed by Scheme 0 when a fin_i operation is processed. In Scheme 0, inserting $\widehat{G}_i$'s operations into the queues associated with sites when $init_i$ is processed, serves a function similar to assigning $\widehat{G}_i$ a timestamp. Since transactions $\widehat{G}_i$ are serialized in the order in which the $init_i$ operations are processed, trivially, Scheme 0 ensures that $ser(S)$ is serializable. Thus, the non-serializable schedule in Example 2 cannot result if Scheme 0 is used, even if GTM$_1$ inserts operations into QUEUE in the order mentioned in Example 2. To see this, observe that after $init_1$ and $init_2$ are processed by Scheme 0, since $init_1$ is processed before $init_2$, $\widehat{G}_1$'s operations are inserted into the queues for sites s_1 and s_2 before $\widehat{G}_2$'s operations. Thus, when operation $ser_2(G_2)$ is selected from QUEUE by Scheme 0, after $ser_1(G_1)$ has been processed, $cond(ser_2(G_2))$ does not hold (since $ser_2(G_1)$ is in front of $ser_2(G_2)$ in the queue for s_2) and thus $ser_2(G_2)$ is not processed.

We now analyze the complexity of CC which has the basic structure as shown in Figure 4. The *complexity* of CC is the average number of steps it takes CC to schedule a transaction $\widehat{G}_i$. For the purpose of analyzing the complexity of the various schemes, we assume the following.

- Every transaction $\widehat{G}_i$ has, on an average, d_{av} operations (that is, the average number of sites at which a global transaction executes is d_{av}).
- At no point during the execution of CC does the difference between the number of $init_i$ and fin_i operations processed by CC exceed n.

Since every transaction $\widehat{G}_i$ is assumed to contain d_{av} operations, the average number of steps taken by CC to schedule $\widehat{G}_i$ is the sum of:

- The number of steps required by CC to process $init_i$,
- $d_{av}\times$(the number of steps required by CC to process $ser_k(G_i)$),

```
procedure Basic_Scheme():
Initialize data structures;
while (true)
begin
Select operation $o_j$ from the front of QUEUE;
if $cond(o_j)$ then begin
            $act(o_j)$;
            while (there exists an operation $o_l \in$ WAIT such that $cond(o_l)$ is true)
                begin
                $act(o_l)$;
                WAIT := WAIT -$\{o_l\}$
                end
            end
        else WAIT := WAIT $\cup$ $\{o_j\}$;
end
```

Figure 4: Basic Structure of Conservative Schemes

- ***fin***$_i$: Information relating to $\widehat{G}_i$ is deleted from DS.

We denote by $act(o_j)$, the actions performed by CC when it processes an operation o_j in QUEUE. An essential feature of conservative schemes is that they ensure serializability without resorting to transaction aborts. As a result, it may not always be possible for CC to process an operation when it is selected from QUEUE since processing every operation when it is selected from QUEUE may result in non-serializable schedules.

Example 2: Consider an MDBS environment consisting of two sites s_1 and s_2. Let G_1 and G_2 be global transactions such that transactions $\widehat{G}_1$ and $\widehat{G}_2$ are as follows:

$$\widehat{G}_1 : \; ser_1(G_1) \;\; ser_2(G_1)$$

$$\widehat{G}_2 : \; ser_2(G_2) \;\; ser_1(G_2)$$

Let GTM$_1$ insert operations into QUEUE in the following order.

$$init_1 \;\; init_2 \;\; ser_1(G_1) \;\; ser_2(G_2) \;\; ser_2(G_1) \;\; ser_1(G_2) \;\; fin_1 \;\; fin_2$$

If operations are processed by CC when they are selected from QUEUE, the following non-serializable schedule $ser(S)$ results.

$$ser_1(G_1) \;\; ser_2(G_2) \;\; ser_2(G_1) \;\; ser_1(G_2) \qquad \Box$$

Thus, associated with every operation o_j in QUEUE is a condition, $cond(o_j)$, that is defined over DS and that must hold if o_j is to be processed by CC. If $cond(o_j)$ does not hold when operation o_j is selected from QUEUE by CC, then o_j is added to a set of waiting operations, WAIT, to be processed at a later time when $cond(o_j)$ becomes true. Thus, every conservative scheme for ensuring the serializability of $ser(S)$ has the same basic structure as shown in Figure 4. However, different conservative schemes differ in the values for $act(o_j)$ and $cond(o_j)$ for

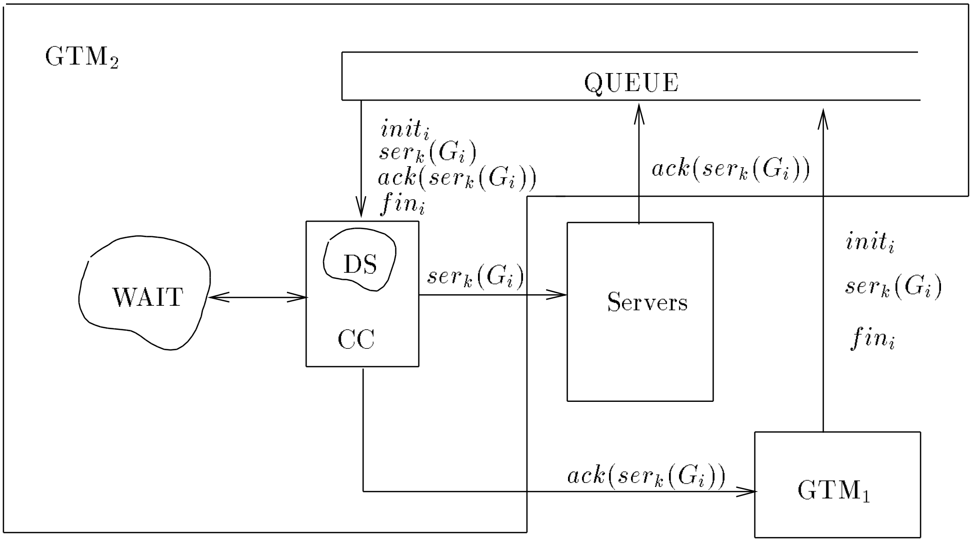

Figure 3: Basic Structure of GTM$_2$

4 Structure and Complexity of Conservative Schemes

In this section, we describe the basic structure of conservative concurrency control schemes employed by GTM$_2$, and the methodology we adopt for analyzing their complexity. As mentioned earlier, GTM$_1$ submits the $ser_k(G_i)$ operations belonging to each global transaction G_i (or alternatively, each transaction $\widehat{G_i}$) to GTM$_2$. GTM$_1$ inserts these operations into a queue, QUEUE. In addition, for every transaction $\widehat{G_i}$, GTM$_1$ inserts into QUEUE, the operations $init_i$ and fin_i (whose utility is discussed below). We now briefly describe the operations in QUEUE for an arbitrary transaction $\widehat{G_i}$ and site s_k.

- $\boldsymbol{init}_i$: This operation is inserted into QUEUE by GTM$_1$ before any other operation belonging to $\widehat{G_i}$ is inserted into QUEUE.
- $\boldsymbol{ser}_k(\boldsymbol{G}_i)$: This operation is inserted into QUEUE by GTM$_1$ in order to request the execution of operation $ser_k(G_i)$.
- $\boldsymbol{ack}(\boldsymbol{ser}_k(\boldsymbol{G}_i))$: This operation is inserted into QUEUE by the servers when the local DBMSs complete executing operation $ser_k(G_i)$.
- $\boldsymbol{fin}_i$: This operation is inserted into QUEUE by GTM$_1$ after $ack(ser_k(G_i))$, for all $ser_k(G_i) \in \widehat{G_i}$ have been received by GTM$_1$.

Note that the $init_i$ and fin_i operations do not belong to transaction $\widehat{G_i}$.

In Figure 3, we present the basic structure of GTM$_2$. CC is any conservative concurrency control scheme for ensuring serializability of $ser(S)$. CC selects operations from the front of QUEUE, in order to *process* them. Associated with CC are certain data structures (DS) that are manipulated every time an operation selected from QUEUE is processed by it. In addition, the following actions are performed by CC when it processes an operation o_j in QUEUE.

- $\boldsymbol{init}_i$: Operation $init_i$ contains information relating to transaction $\widehat{G_i}$ (e.g., the operations in $\widehat{G_i}$, the set of sites G_i executes at). This information is utilized by CC to determine conflicting operations and is added to DS.
- $\boldsymbol{ser}_k(\boldsymbol{G}_i)$: Operation $ser_k(G_i)$ is submitted to the local DBMSs for execution (through the servers).
- $\boldsymbol{ack}(\boldsymbol{ser}_k(\boldsymbol{G}_i))$: Operation $ack(ser_k(G_i))$ is forwarded to GTM$_1$.

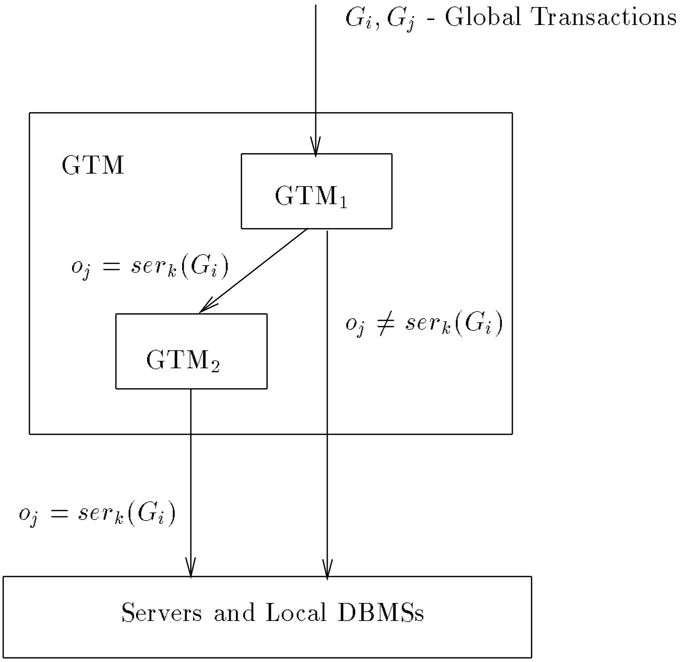

Figure 2: The GTM Components

protocols to ensure serializability of $ser(S)$ must avoid transaction aborts, that is, they must be *conservative* (e.g., conservative 2PL, conservative TO [BHG87]). This is quite feasible in an MDBS environment.

2. Concurrency control protocols that provide a low degree of concurrency may be unsuitable for ensuring that $ser(S)$ is serializable since such protocols may cause a number of operations in $ser(S)$ to be delayed unnecessarily. Such delays may adversely affect the performance of the system since unnecessarily delaying an operation in $ser(S)$ may correspond to delaying the execution of an entire global subtransaction. For example, for a site s_k that uses the TO scheme, ser_k may map each transaction to its begin operation. As a result, causing an operation of $ser(S)$ to wait could cause the execution of an entire global subtransaction to be delayed.
3. A common problem with concurrency control protocols that provide a high degree of concurrency is that they incur substantial overhead for scheduling a single operation (e.g., SGT). However, it may be justifiable to use such concurrency control schemes with high overhead in order to ensure the serializability of $ser(S)$, since the overhead involved in scheduling an operation of $ser(S)$ is amortized, not over one operation, but over all the operations belonging to the corresponding global subtransaction. Thus, the gain in terms of increased throughput, faster response times, and the number of global subtransactions that may be permitted to execute concurrently by a concurrency control scheme that permits a high degree of concurrency may outweigh the overhead associated with the concurrency control scheme.

The above factors imply that concurrency control schemes for ensuring the serializability of $ser(S)$ must be conservative, and must provide high degrees of concurrency (even though they may involve a high overhead). Conservative schemes for ensuring global serializability in MDBS environments have been proposed in [BS88, ED90], while non-conservative schemes have been proposed in [Pu88, GRS91].

$$ser(S): \; b_{11} \;\; b_{21} \;\; c_{12} \;\; c_{22} \quad \square$$

In the global schedule S, two operations conflict if both access the same data item and one of them is a write operation. Thus in Example 1, operations $w_1(c)$ and $r_2(c)$ conflict in S. However, the notion of conflict between operations in $ser(S)$ is defined differently. Operations $ser_k(G_i)$ and $ser_l(G_j)$ conflict in $ser(S)$ if and only if $k = l$. Thus, in Example 1, operations b_{11} and b_{21} conflict in $ser(S)$, whereas operations b_{11} and c_{22} do not conflict in $ser(S)$. Note that operations b_{11} and b_{21} do not conflict in S. From Theorem 1, it follows that S is serializable if $ser(S)$ is serializable.

Theorem 2: Consider an MDBS where each local schedule is serializable. A global schedule S is serializable if $ser(S)$ is serializable.
Proof: See Appendix A. $\square$

In Example 1, note that $ser(S)$ is serializable (the serialization order being $\widehat{G}_1$ before $\widehat{G}_2$). As a result, global schedule S is serializable. We have thus reduced the problem of ensuring serializability in an MDBS environment to the problem of ensuring that $ser(S)$ is serializable. Since global transactions execute under the control of the GTM, the GTM can control the execution of the operations in $ser(S)$ in order to ensure that $ser(S)$ is serializable. Thus, for ensuring global serializability in an MDBS environment, we can restrict ourselves to the development of schemes for ensuring that $ser(S)$ is serializable.

To do so, we split the GTM into two components, GTM_1 and GTM_2 (see Figure 2). GTM_1 utilizes the information on serialization functions for various sites in order to determine for every global transaction G_i, operations $ser_k(G_i)$, and submits them to GTM_2 for processing. The remaining global transaction operations (that are not $ser_k(G_i)$) are directly submitted to the local DBMSs (through the servers). Further, GTM_1 does not submit an operation belonging to a global transaction G_i (except the first operation) to either the local DBMSs or GTM_2 unless an acknowledgement for the completion of the execution of the previous operation belonging to G_i (at the local DBMSs) has been received.

GTM_2 is responsible for ensuring that the operations submitted to it by GTM_1 execute at the local DBMSs in such a manner that $ser(S)$ is serializable. Our concern, for the remainder of the paper, shall be the development of concurrency control schemes for GTM_2 that ensure $ser(S)$ is serializable. A discussion on mechanisms that GTM_1 can adopt in order to determine operations in $ser(S)$ can be found in Appendix E.

3 Characteristics of the Concurrency Control Problem

A number of schemes for ensuring serializability in centralized DBMSs exist in the literature (e.g., 2PL, TO, SGT). Any one of them can be employed by GTM_2 in order to ensure that $ser(S)$ is serializable. However, certain characteristics of MDBS environments make some of the existing schemes unsuitable for ensuring the serializability of $ser(S)$. Below, we list some of the factors that play an important role in the design of concurrency control protocols for ensuring the serializability of $ser(S)$.

1. In most MDBS environments, we expect the number of sites to be small in comparison to the number of *active* global transactions in the system (that is, global transactions that have begun execution, but have not yet completed execution). Thus, since any pair of operations $ser_k(G_i)$ and $ser_k(G_j)$ conflict in $ser(S)$, $ser(S)$ may contain a large number of conflicting operations. As a result, if, for example, the 2PL protocol were used to ensure the serializability of $ser(S)$, then there would be frequent deadlocks; if instead, the TO or optimistic protocols were used, a large number of transaction aborts would result. An abort of transaction $\widehat{G}_i$ in $ser(S)$ corresponds to the abortion of the global transaction G_i, which may be expensive, and thus highly undesirable in an MDBS environment. Thus,

Unfortunately, serialization functions may not exist for sites following certain protocols (e.g., *serialization graph testing* (SGT)). For such sites, serialization functions can be introduced using *external* means by forcing conflicts between transactions [GRS91]. For example, every transaction in τ_k can be forced to write a particular data item at site s_k, say, *ticket*. Thus, if some transaction $T_i \in \tau_k$ is serialized before another transaction $T_j \in \tau_k$ in S_k, then T_i must have written *ticket* before T_j wrote it. Thus, the function that maps every transaction $T_i \in \tau_k$ to its write operation on *ticket* is a serialization function for s_k. We denote by ser_k, any one of the possible serialization functions for site s_k.

2.3 Global Serializability

Serialization functions can be used to ensure global serializability in an MDBS environment. In the following theorem, we state a sufficient condition for ensuring global serializability in an MDBS environment.

Theorem 1: Consider an MDBS where each local schedule is serializable. Global schedule S is serializable if there exists a total order $\prec_G$ on global transactions such that at each site s_k, for all pairs of global transactions G_i, G_j executing at site s_k, if $ser_k(G_i) \prec_{S_k} ser_k(G_j)$, then $G_i \prec_G G_j$.
Proof: See Appendix A. □

We denote the set of sites at which a global transaction G_i executes by $exec(G_i)$. For every global transaction G_i, we define transaction $\widehat{G}_i$ to be a restriction of G_i consisting of all the operations in $\{ser_k(G_i) : G_i \text{ executes at site } s_k\}$. For global schedule S, we define schedule $ser(S)$ to be the set of operations belonging to all transactions $\widehat{G}_i$, with a partial order on them. Further, $ser(S)$ is a restriction of S.

Example 1: Consider an MDBS environment consisting of two sites: s_1 containing data items a and b, and s_2 containing data item c. Suppose that the local DBMS at site s_1 follows the TO scheme in which a timestamp is assigned to a transaction when it begins execution, and the local DBMS at site s_2 follows the strict 2PL protocol [BHG87]. Consider the following global transactions G_1 and G_2 that execute at sites s_1 and s_2.

$$G_1 : \ b_{11} \ \ w_1(a) \ \ b_{12} \ \ w_1(c) \ \ c_{11} \ \ c_{12}$$

$$G_2 : \ b_{21} \ \ r_2(b) \ \ b_{22} \ \ r_2(c) \ \ c_{21} \ \ c_{22}$$

Let L_3 be a local transaction executing at site s_1.

$$L_3 : \ b_3 \ \ r_3(a) \ \ w_3(b) \ \ c_3$$

Let ser_1 be the function that maps every transaction in τ_1 to its begin operation. Also, let ser_2 be the function that maps every transaction in τ_2 to its commit operation. Thus, $ser_1(G_1) = b_{11}$, $ser_1(G_2) = b_{21}$, $ser_2(G_1) = c_{12}$ and $ser_2(G_2) = c_{22}$. As a result, transactions $\widehat{G}_1$, $\widehat{G}_2$ are as follows.

$$\widehat{G}_1 : \ b_{11} \ \ c_{12}$$

$$\widehat{G}_2 : \ b_{21} \ \ c_{22}$$

Consider the global schedule S resulting from the concurrent execution of transaction G_1, G_2 and L_3 such that the local schedules at sites s_1 and s_2 are as follows.

$$\begin{array}{l} S_1 : \ b_{11} \ \ b_3 \ \ w_1(a) \ \ b_{21} \ \ r_3(a) \ \ w_3(b) \ \ c_3 \ \ r_2(b) \ \ c_{11} \ \ c_{21} \\ S_2 : \ b_{22} \ \ b_{12} \ \ w_1(c) \ \ c_{12} \ \ r_2(c) \ \ c_{22} \end{array}$$

Schedule $ser(S)$ (which is a total order in this case) is as follows.

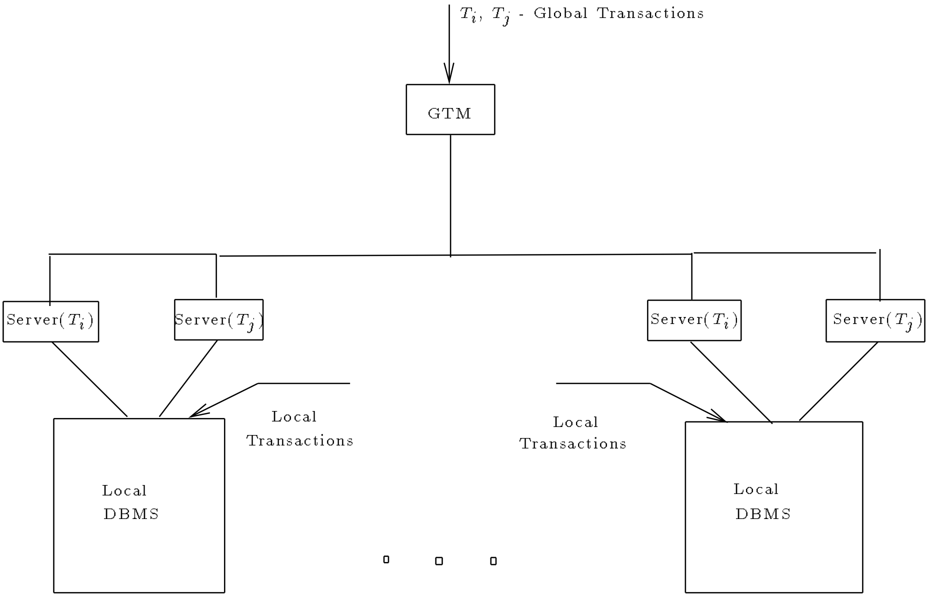

Figure 1: The MDBS Model

We assume that the GTM is centrally located, and controls the execution of global transactions. It communicates with the various local DBMSs by means of *server* processes (one per transaction per site) that execute at each site on top of the local DBMSs (see Figure 1). We assume that the interface between the servers and the local DBMSs provides for operations to be submitted by the servers to the local DBMSs, and the local DBMSs to acknowledge the completion of operations to the servers. The local DBMSs do not distinguish between local transactions and global subtransactions executing at its site. In addition, each of the local DBMSs ensures that local schedules are serializable[2].

2.2 Serialization Functions

In order to develop our idea, we need to first introduce the notion of *serialization function*, which is similar to the notion of serialization event [ED90]. Let τ_k be the set of all global subtransactions in S_k. A serialization function for s_k, ser, is a function that maps every transaction in τ_k to one of its operations such that for any pair of transactions $T_i, T_j \in \tau_k$, if T_i is serialized before T_j in S_k, then $ser(T_i) \prec_{S_k} ser(T_j)$.

For example, if the *timestamp ordering* (TO) concurrency control protocol is used at site s_k, and the local DBMS at site s_k assigns timestamps to transactions when they begin execution, then the function that maps every transaction $T_i \in \tau_k$ to T_i's begin operation is a serialization function for s_k.

For a site s_k, there may be multiple serialization functions. For example, if the local DBMS at s_k follows the *two phase locking* (2PL) protocol, then a possible serialization function for s_k maps every transaction $T_i \in \tau_k$ to the operation that results in T_i obtaining its last lock. Alternatively, the function that maps every transaction $T_i \in \tau_k$ to the operation that results in T_i releasing its first lock is also a serialization function for s_k[3].

---

[2] In this paper, we limit ourselves to conflict serializability (CSR) [Pap86], which we shall refer to, in the remainder of the paper, as serializability.

[3] Actually, any function that maps every transaction $T_i \in \tau_k$ to one of its operations that executes between the time T_i obtains its last lock and the time it releases its first lock is a serialization function for s_k.

pre-existing and autonomous local database management systems (DBMSs) located at different sites. Transactions in an MDBS are of two types:

- **Local transactions.** Those transactions that execute at a single site.
- **Global transactions.** Those transactions that may execute at several sites.

The execution of the global transactions is co-ordinated by the *global transaction manager* (GTM) – a software package built on top of the existing DBMSs whose function is to ensure that the concurrent execution of local and global transactions is serializable. Ensuring global serializability in an MDBS is complicated by the fact that each of the participating local DBMSs is a pre-existing database system whose software cannot be modified. As a result,

- Each local DBMS may follow a different concurrency control protocol.
- Local DBMSs may not communicate any information (e.g., conflict graph) relating to concurrency control to the GTM.
- The GTM is unaware of indirect conflicts between global transactions due to the execution of local transactions at the local DBMSs. This is due to the fact that pre-existing local applications make calls to the local DBMS interfaces, and thus the GTM, which is built on top of the local DBMSs, is not involved in the execution of the local transactions.

Various schemes to ensure global serializability in an MDBS environment have been previously proposed (e.g., [BS88, Pu88, ED90, GRS91]). These proposed schemes have been ad-hoc in nature, and no analysis of their performance, the degree of concurrency provided by them, or their complexity has been made. In this paper, we provide a unifying framework for the study and development of concurrency control schemes in an MDBS environment. We utilize a notion similar to *serialization events* [ED90] (referred to as *O-elements* in [Pu88]) in order to reduce the problem of ensuring global serializability in an MDBS to the problem of ensuring serializability in a centralized DBMS. We then develop a range of concurrency control schemes for ensuring global serializability in an MDBS environment. Finally, we compare the degree of concurrency provided by each of the various schemes and analyze their complexities.

2 MDBS Concurrency Control

In this section, we show how the problem of ensuring global serializability in an MDBS can be reduced to the problem of ensuring serializability in a centralized DBMS. Since centralized concurrency control is a well studied problem and a number of schemes for ensuring serializability in centralized DBMSs have been proposed in the literature, the development of concurrency control schemes for MDBSs is thus simplified. We begin by first discussing the MDBS model.

2.1 The MDBS Model

An MDBS is a collection of pre-existing DBMSs located at sites $s_1, s_2, \ldots, s_m$. A transaction T_i in an MDBS environment is a totally ordered set of **read** (denoted by r_i), **write** (denoted by w_i), **begin** (denoted by b_i) and **commit** (denoted by c_i) operations. A global transaction may have multiple begin and commit operations, one for each site at which it executes. We denote by b_{ik} and c_{ik}, the begin and commit operations of global transaction T_i at site s_k respectively. A *global schedule* S is the set of all operations belonging to local and global transactions with a partial order $\prec_S$ on them. The *local schedule* at a site s_k, denoted by S_k, is the set of all operations (belonging to local and global transactions) that execute at s_k with a total order $\prec_{S_k}$ on them. The schedule S_k is a *restriction*[1] of the global schedule S.

---

[1] A set P_1 with a partial order $\prec_{P_1}$ on its elements is a *restriction* of a set P_2 with a partial order $\prec_{P_2}$ on its elements if $P_1 \subseteq P_2$, and for all $e_1, e_2 \in P_1$, $e_1 \prec_{P_1} e_2$ if and only if $e_1 \prec_{P_2} e_2$.

The Concurrency Control Problem in Multidatabases: Characteristics and Solutions*

Sharad Mehrotra[1]
Rajeev Rastogi[1]
Yuri Breitbart[2]
Henry F. Korth[3]
Abraham Silberschatz[1]

[1]Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA

[2]Department of Computer Sciences
University of Kentucky
Lexington, KY 40506

[3]Matsushita Information Technology Laboratory
182 Nassau Street, third floor
Princeton, NJ 08542-7072

Abstract

A *Multidatabase System* (MDBS) is a collection of local database management systems, each of which may follow a different concurrency control protocol. This heterogeneity makes the task of ensuring global serializability in an MDBS environment difficult. In this paper, we reduce the problem of ensuring global serializability to the problem of ensuring serializability in a centralized database system. We identify characteristics of the concurrency control problem in an MDBS environment, and additional requirements on concurrency control schemes for ensuring global serializability. We then develop a range of concurrency control schemes that ensure global serializability in an MDBS environment, and at the same time meet the requirements. Finally, we study the tradeoffs between the complexities of the various schemes and the degree of concurrency provided by each of them.

1 Introduction

The problem of transaction management in a multidatabase system (MDBS) has received considerable attention from the database community in recent years [BS88, Pu88, DE89, BST90, ED90, GRS91, PRR91, MRKS91]. The basic problem is to design an effective and efficient transaction management scheme that allows users to access and update data items managed by

---

*Work partially supported by NSF grants IRI-8805215, IRI-9003341 and IRI-9106450, and by grants from the IBM corporation and the NEC corporation.

THE CONCURRENCY CONTROL PROBLEM IN MULTIDATABASES: CHARACTERISTICS AND SOLUTIONS

Sharad Mehrotra, Rajeev Rastogi, Yuri Breitbart,
Henry F. Korth, and Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-37 December 1991

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712