1-1-2006

Application of snapshot isolation protocol to concurrent processing of long transactions

Yang Yang
University of Wollongong, yy214@uow.edu.au

Janusz R. Getta
University of Wollongong, jrg@uow.edu.au

Application of snapshot isolation protocol to concurrent processing of long transactions

Abstract

Snapshot Isolation (SI) protocol is a database transaction processing algorithm used by some of commercial database systems to manage the concurrent executions of database transactions. SI protocol is a special case of multi-version algorithm. It avoids many of the anomalies typical for the concurrent processing of database transactions. Unfortunately, SI protocol does not guarantee correct serialization of database transactions under certain conditions. A recent work [3] proposed a formal solution, which characterizes the correctness of transactions running under SI protocoL However, the protocol is inefficient when it comes to processing long transactions. In this paper, we show that the limitations imposed on the structures of long transactions improve performance of SI protocol. A different way to characterize the serializability of schedule under SI dynamically is proposed and proved.

Keywords

Application, snapshot, isolation, protocol, concurrent, processing, long, transactions

Disciplines

Physical Sciences and Mathematics

Publication Details

Yang, Y. & Getta, J. R. (2006). Application of snapshot isolation protocol to concurrent processing of long transactions. The IASTED International Conference on Databases and Applications, DBA 2006 (pp. 38-43). Anaheim, USA: ACTA Press.

Proceedings of the 24th IASTED International Multi-Conference
DATABASES AND APPLICATIONS
February 13-15, 2006, Innsbruck, Austria

APPLICATION OF SNAPSHOT ISOLATION PROTOCOL TO CONCURRENT PROCESSING OF LONG TRANSACTIONS

Yang Yang Janusz R.Getta
School of Information Technology and Computer Science
University of Wollongong
Wollongong, NSW 2522
Australia
yy214@uow.edu.au jrg@uow.edu.au

ABSTRACT
Snapshot Isolation (SI) protocol is a database transaction processing algorithm used by some of commercial database systems to manage the concurrent executions of database transactions. SI protocol is a special case of multi-version algorithm. It avoids many of the anomalies typical for the concurrent processing of database transactions. Unfortunately, SI protocol does not guarantee correct serialization of database transactions under certain conditions. A recent work [3] proposed a formal solution, which characterizes the correctness of transactions running under SI protocol. However, the protocol is inefficient when it comes to processing long transactions. In this paper, we show that the limitations imposed on the structures of long transactions improve performance of SI protocol. A different way to characterize the serializability of schedule under SI dynamically is proposed and proved.

KEY WORDS
Transaction Processing, Snapshot Isolation, Long Transaction

1. Introduction

Snapshot isolation protocol [1] is a special kind of multi-version concurrency control algorithm used in database systems to prevent the most of common anomalies typical for concurrent processing of database transactions. Under the protocol, every read operation of transaction t can only access the version of data item either created by t or the latest transaction that committed before t. Each write operation of transaction t creates a new version of a written data item and the original copy of the data item is not overwritten until t is committed successfully. The protocol enforces a mechanism commonly called as *first-committer-wins* where transaction t is able to commit only when it did not perform *write* operation on a data item written by another committed transaction concurrent with t. Otherwise, transaction t is aborted and it has to be resubmitted later. We say that transaction t_i is concurrent with transaction t_j when their time intervals from start-protocol an efficient concurrency control algorithm where no conflict operations are delayed by locking and *lost update* anomaly can be precluded. SI protocol provides an isolation level almost as strong as *serializable level* with as high concurrency as at *read uncommitted* level.

Unfortunately, as proved in [1] and [2], SI protocol does not guarantee serializability in every possible case of concurrent execution. Some of the concurrent executions acceptable under SI protocol are not serializable and contribute to the corruption of a database.

Example 1.1
This example follows [1] and considers a combined account, which consists of a check account and a cash account. Each one of sub-accounts can be overdrawn as long as their total balance is not negative. Suppose we have a schedule h running under SI protocol:
h: $read_1(x0=50)$ $read_1(y0=50)$ $read_2(x0=50)$ $read_2(y0=50)$ $write_1(y=-40)$ $write_2(x=-40)$ $commit_1$ $commit_2$

Since the write operations are performed on different data items, no transaction in the execution above violates the *first-committer-wins* principle. However, this execution sets a balance of the combined account to *-80,* which violates the constraint and corrupts a database. This defect makes conflict serializability of SI protocol not guaranteed in every possible case.

The most recently, a formal approach, which can characterize the correctness of executions of transactions under SI was proposed [3]. The method is based on the maintenance of a binary directed graph also called as *interference graph*, which represents interferences among the transactions. Nodes in interference graph are committed transactions. Edges between nodes are added by observing the following rules:

- There is an exposed edge $T_j \xrightarrow{expo} T_k$ between transactions T_j and T_k, when
  $readset(T_j) \cap writeset(T_k) \neq \emptyset$, and
  $writeset(T_i) \cap writeset(T_k) = \emptyset$.

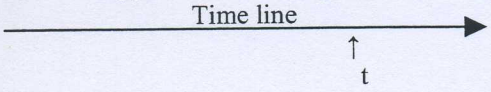
- There is an protected edge $T_j \xrightarrow{prot} T_k$, when writeset(T_j) ∩ writeset(T_k) ≠∅ , or readset(T_j) ∩ writeset(T_k) = ∅, and writeset(T_j) ∩ readset(T_k) ≠∅.

A node T_B in the interference graph is distinguished as *pivot* if it is in a chord-free cycle and $T_A \xrightarrow{expo} T_B$ and $T_B \xrightarrow{expo} T_C$. An execution is conflict serializable under SI protocol when an interference graph has no pivot. An assumption that significantly weakens this solution is that all the operations of concurrent transactions should be known in advance. It means that we can either apply this mechanism at the design phase of transactions or will not be able to characterize the serializability of concurrent transactions at runtime until all transactions reached their commit point. Since it is not practical to predict, which transactions will be executed concurrently at runtime or analyze the interference graph of all the transactions in a system, only the latter option should be considered. In order to prevent the nonserializable execution, those transactions characterized as pivots have to be aborted when they are about to commit. However, in a system running long transactions abortion at a commit point is not acceptable. Suppose two concurrent long transactions T_1 and T_2 are running concurrent under Snapshot Isolation:

Example 1.2

T_1: R(x) W(x) R(y).....C_1

T_2: R(y) R(x) W(y).....C_2



At point t on the time line, the information about the transactions at point t has been adequate to detect the unserializability of schedule (i.e. T_1 and T_2 can be characterized as pivots by [3]'s approach). Since transaction T_2 is destined to be aborted at commit point in order to avoid the unserializable schedule, time and system resources expended on operations of T_2 after point t are wasted. If a nonserializable execution of a long transaction is detected at the early stages of its execution, time and system resources do not need to be wasted to continue the execution. If transaction T_1 is aborted due to some accident (e.g. program bug, hard disk failure, etc) before commit point, the unserializability detected at point t is vanished automatically. Then, the proposal of aborting T_2 at time point t becomes unnecessary. That is one advantage of the interference graph based approach. The serializability can be characterized precisely at the very end of all transactions. However, since well-tested programs and stable host are basic requirements of modern database system, the frequency of accidental aborting is reasonably lower than the frequency of unserializable schedule emergence. Therefore, it is important to detect noserializable execution as soon as it is possible.

A schedule is an increasing sequence of operations, submitted by different transactions. In order to dynamically detect a nonserializable execution under SI protocol while the schedule is growing, the verification of serializability should be performed repeatedly at numbers of proper verification points in transaction but not only once before commit. The repeated verifications require a "smaller atomicity" of transaction. A concept of "breaking point" is proposed in [4], to partition a transaction into the sets of consecutive steps. An algorithm which finds the finest chopping of a set of transaction is given in [5].

The rest of the paper is organized in the following way. Section 2 introduces a new transaction model, which achieves appropriate granularity without explicit "breaking points" or "transaction chopping". A dynamically managed graph which can be used to characterize the serializability of on going schedule under SI is presented in section 3. The comparison between our new proposed approach and [3]'s approach will also be included in the same section. The paper is concluded and further works are commented in section 4.

2. Transaction model

In this section we introduce the basic concepts of our transaction model. A transaction is a sequence of operations that ends with either *commit* or *abort* operation. For the sake of simplicity, we only consider only the transactions that end with *commit* operation. The remaining operations are *read* or *write* operations. *Write* is the operation which signs the new value of data item that already exists in a database. *Read* operation retrieves the value of data item from database. The purpose of reading a value of data item from a database is either to inform external user about the value or to use the value in the computations of new values or verification of the consistency constraints. A *write* operation on data item x is denoted by W(x) and it is a pair <(x), s> where s is the set of data items which's values are necessary to perform W(x). We say that *write* operation on x "depends on" a data set s.

A database transaction can be logically partitioned into the segments where *write* operations end each segment. Moreover, an application programmer must observe a rule *"Before a data item x is written, a transaction reads only the data items that write(x) depends on"*. Consequently, the read operations, which only inform data value to external user are automatically arranged between the last write and commit point in our transaction model. The following example provides more intuitions.

Example 2.1

An enrolment transaction of a university administration system verifies the following consistency constraints. In this transaction, the admission offer(o) and tuition fee payment(t) should be checked. If there is no unsatisfied

condition in the offer and no outstanding balance in payment, the status of student (s) will be changed to "enrolled". Also, preferred contact method (c) provided by student before should be replaced by university email account generated by system automatically. The number of student in certain school (n) will be increased by 1. At last, a welcome letter (l) retrieved from database will be print out.
Formerly, this transaction might be programmed as:
T: R(o) R(t) R(n) R(l) W(s) W(c) W(n) C
Just as all transaction models listed in [3]. However, by following the programming rule that we proposed above, the model of this enrolment transaction will be better organized like:
T: R(o) R(t) W(s) W(c) R(n) W(n) R(l) C
In this model, transaction is logically partitioned into smaller granularities and dependencies between write operations and read operations are self-revealed.

Definition 2.1
A segment s is a sequence of read operations followed by a write operation or commit. A segment starts either at the beginning of transaction or after a write operation and ends after the next write or commit.

Definition 2.2
Segmented transaction is a sequence s_1, ..., s_n, c where each s_i is a segment and c is a commit operation. Additionally,

1. *in each transaction each data item is read or written at most once,*
2. *no data item is read again after it has been written.*

When a data item is read its value is saved in a local variable available to any further operation in the transaction. Moreover, only the last write step determines the final value of data item produced by this transaction. Any other write operations on the same data item will be overwritten. This justifies condition (1) in the definition of segmented transactions.

Before a data item x is written, the new valued will be computed and stored in a local variable as long as the transaction is still alive. Then, after the write operation on x, the new value can be accessed from the local variable. Any further read of x from the database will lead to an unnecessary overload. This justifies condition (2) in the definition of segmented transactions.

3. The serializability of SI

3.1 Multiversion serialization graph for SI

A segmented model of long transactions allows for the identification of nonserializable executions at run time. Since it is very inefficient to abort a long transaction at the end of its execution, we propose to verify the serializability of schedule dynamically such that the nonserializable transactions is aborted as soon as possible. The dynamic verification requires identification of the points in which the verification can be performed. Over dense or spare separate points make the dynamic verification too resource demanding or just meaningless. The separate point should be set on each time the transaction significantly grows. Obviously, as an indicator of each segment, write operations are reasonable separate points in transaction model.

Generally, the interference graph of schedule can be created when the first operation is issued but no at the end of the whole schedule. A node, which indicates corresponding transaction is included when the transaction starts. Every time the scheduler receives a *write* operation, the interference between transactions is evaluated (i.e. edges between nodes in the interference graph are updated). The transaction, even its further part has not been received by scheduler will be aborted if it is characterised as a pivot in the interference graph. Then the node which indicates the aborted transaction is removed from interference graph. Consequently, system resources which used to be wasted on part of long transaction which is predetermined to be aborted can be saved.

However, the mechanism of detecting pivots in interference graph does have some critical disadvantages. Firstly, the interference graph of transactions is pretty complicated. As the author of [3] indicated, the interference edges always come in pairs: When there is an edge from T_j to T_k then there is a reverse edge from the reverse direction, from T_k to T_j. Secondly, the dynamic management of interference graph has not been discussed. In the following part of this section, we will present a different and more concise graphical mechanism, which plays the same role as interference graph introduced in [3].

In [6], the author proposed and proved a theorem which characterizes the correctness of general multiversion concurrency control algorithm. A multiversion schedule S is one-copy-serializable if and only if the multiversion serialization graph (MVSG) is acyclic. Obviously, with a few revisions on the definition of MVSG, this theorem also works for SI. This is because SI is a special instance of multiversion concurrency control algorithm. The original definition of MVSG in [6] states:

Definition 3.1.1
For a given MV schedule S and a version order <<, the multiversion serialization graph for S and <<, MVSG(S,<<), is serialization graph(S) with the following version order edges added: for each $r_k[x_j]$ and $w_i[x_i]$ where i, j, and k are distinct, if $x_i << x_j$ then include $T_i \rightarrow T_j$, otherwise include $T_k \rightarrow T_i$.

Sine MVSG actually is a classical serialization graph together with some additional edges caused by dependencies between different versions of the same data item, edges in MVSG can be catalogued as the following:

Edges in MVSG
- A, Edges between same versions of the same data item
- B, Edges between different versions of the same data item

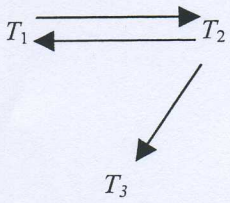
In fact, only edges in catalogue B is decided by two rules in definition 3.1. The first condition figures out the edge between two write operations from older version to newer version of the same data item. The second condition figures out the edge between a read operation on older version and a write operation on newer version of the same data item. For example, given the following schedule under SI:

Example 3.1.1

T_1: $R_1(z_0)$ $W_1(x_1)$ $R_1(y_0)$ $W_1(y_1)...C_1$
T_2: $R_2(y_0)$ $W_2(x_2)$ C_2
T_3: $R_3(x_2)...C_3$

For $W_2(x_2)$ and $R_3(x_2)$, because they operate on the same versions of the same data item and $W_2(x_2)$ precedes $R_3(x_2)$, there is an edge from T_2 to T_3
For $R_2(y_0)$ and $W_1(y_1)$, because $y_1>>y_0$, there is an edge from T_2 to T_1.
For $R_3(x_2)$ and $W_1(x_1)$, because $x_1<<x_2$, there is an edge from T_1 to T_2.

MVSG:



According to the definition 3.1 presented in [6], this schedule is not serializable under a general multiversion concurrency control algorithm.

However, SI has already excluded the situation in which two concurrent write on the same data item by applying "first-committer-wins". We can also exclude the edge between two concurrent transactions which write the same data item from the multiversion serialization graph which is especially for SI. Then, the edge form T_1 to T_2 in the MVSG of schedule given in example 3.1 will be eliminated. As a result, the cycle disappears which means the schedule is serializable under Snapshot Isolation although T_1 will be aborted by "first-committer-wins" later.

Preceding discussion leads us to the following definition and theorem.

Definition 3.1.2
Given a schedule S running under Snapshot Isolation and a version order <<, the multiversion serialization graph for S under SI and <<, SIMVSG(S,<<), is serialization graph(S) with the following version order edges added: for each $r_k[x_j]$ and $w_i[x_i]$ where i, j, and k are distinct, if $x_i << x_j$ and T_i is not concurrent with T_j, then include $T_i \rightarrow T_j$, otherwise include $T_k \rightarrow T_i$.

Theorem 3.1.1
The execution of a schedule S running under Snapshot Isolation is serializable if and only if SIMVSG(S,<<) is acyclic.

Proof: A schedule S is running under SI.
(If)
Assume that SIMVSG(S, <<) is acyclic. According to the definition, MVSG(S, <<) is the SIMVSG(S, <<) plus possible edges between two concurrent transactions which write the same data item (i.e. MVSG(S, <<) is equivalent to or the superset of SIMVSG(S, <<). If MVSG(S, <<) equals to SIMVSG(S, <<), MVSG(S, <<) is also acyclic. Since SI is a multiversion concurrency control algorithm, according to the theorem in [6], H is serializable. On the other hand, if MVSG(S, <<) is the SIMVSG(S, <<) plus edges between two concurrent transactions which write the same data item, one of these transactions will be forced aborted by "first-committer-wins". The execution of remain transaction is still serializable under SI. Therefore, if SIMVSG(S, <<) is acyclic then S is serializable.

(Only if)
Assume that a schedule S is serializable. Since S is serializable and SI is a multiversion concurrency control algorithm, according to the theorem in [6], MVSG(S, <<) must be acyclic. According to the definition, SIMVSG(S, <<) is the MVSG(S, <<) without possible edges between two concurrent transactions which write the same data item (i.e. SIMVSG(S, <<) is the subset of MVSG(S, <<)). The subset of an acyclic graph is also an acyclic graph. So, SIMVSG(S, <<) is acyclic. Therefore, if S is serializable then SIMVSG(S, <<) is acyclic.

Compare SIMVSG(s) with Interference Graph(s) of the same schedule s, we can find the number of edges in SIMVSG(s) is definitely less than Interference Graph(s). Topological sorting is a classical algorithm which can be used to decide the acyclicity of a directed graph. The worst time complexity of this algorithm is O(n+e) in which n indicates the number of nodes and e stands for the number of edges. Obviously, the time consumed on detecting cycle in SIMVSG(s) is guaranteed less than Interference Graph(s). Furthermore, a schedule s can be characterised as unserializable as soon as a cycle is detected in SIMVSG. On the other hand, the

unserializability of s can not be characterized immediately when a cycle was detected in interference graph(s). More time has to be consumed on evaluating every node in the cycle to find out whether there are some nodes satisfy the conditions of "pivot". What worse is, if no node in this cycle is characterised as pivot, the searching of other cycles in interference graph(s) has to be continued. In conclusion, by studying the differences between two approaches of characterizing the serializability of SI, detecting pivot in interference graph proposed from [3] and detecting cycle in SIMVSG proposed from this paper, our approach is proved to be more efficient.

3.2 Dynamically management of SIMVSG

After the proposal and clarification of SIMVSG, we will present that how to manage this graph dynamically so that the approach can be implemented more efficiently for long transactions.

A node for T_i in its SIMVSG is added when scheduler receives the first operation of transaction T_i. As soon as a write operation $W_i(x)$ is received, the edges between nodes in SIMVSG will be evaluated by following definition 3.2. A significant practical consideration is when the scheduler may discard the information it has collected about a transaction, i.e. remove the node for a particular transaction from the graph. To detect conflicts, we have to maintain the read set and write set of every transaction exists in SIMVSG, which could consume a lot of space. It is therefore important to discard this information as soon as possible. For an aborted transaction, all the information of it has been discarded automatically, so its corresponding node can be removed as soon as the transaction is aborted. For a committed transaction, one may also assume that the schedule can delete information about a transaction and remove the node as soon as it commits. Unfortunately, this is not so. For instance, consider the scheduler in example 3.1, if we remove the node for transaction T_2 after point C_2, the edge from T_2 to T_1 and T_2 to T_3 will be missed. Then the unserializability which could have lead to the aborting of T_1 is missed. So, the scheduler can delete information about a committed transaction T_i iff T_i could not, at any time in the future, be involved in a cycle of SIMVSG. For a node to form a cycle with an acyclic graph, it must have at least one incoming edge from and one outgoing edge to that acyclic graph. According to definition 3.2, if T_i is concurrent with T_j, edges between them can be both direction. On the other hand, if T_i is committed before T_j starts, edge between them can only from T_i to T_j. So, for a set of committed transactions S which's SIMVSG is acyclic and a transaction T_i which is started after all transactions in S have been committed, cycle will never be formed. Then the information of all transactions in S can be discarded by schedule. The corresponding nodes for them can be removed from SIMVSG. In conclusion, the node of a transaction can be removed from SIMVSG if all the other transactions which have node in SIMVSG have already been committed.

The following is an example of dynamically management of SIMVSG.

Example 3.2.1

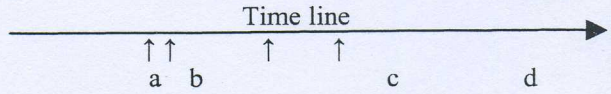
The following is the schedule S of four executing long transactions under SI.

T_1: $R_1(x_0)..C_1$

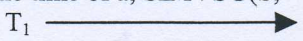
T_2: .....$W_2(x_2)$.....$R_2(y_0)$....C_2

T_3:..... $W_3(y_3)$...$R_3(x_2)$...$W_3(z_3)$..C_3

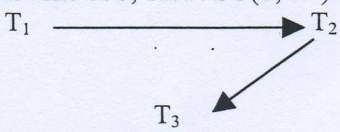
T_4: $W_4(p_4)$..



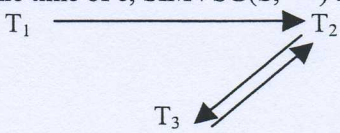
At the time of a, SIMVSG(S, <<) is:



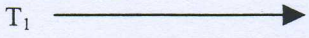
At the time of b, SIMVSG(S, <<) is updated as:



At the time of c, SIMVSG(S, <<) is updated as:



A cycle is formed, so T_3 is forced to abort and SIMVSG(S, <<) becomes like:



At the time of d, SIMVSG(S, <<) is updated as:

T_4

The serializability of S is guaranteed and SIMVSG(S, <<) will keep being updated and verified as furthering operations of transactions coming.

4. Conclusion

This paper addresses the performance problems of Snapshot Isolation (SI) protocol in the context of long transactions. A recently proposed approach, which can only characterize the serializability of SI at design phase is considered as inefficient due to time and system resource wasted during the execution of "to be aborted" transaction. When dealing with long transactions nonserializable executions should be immediately detected when transactions change the contents of database.

A revision of traditional transaction model and relationship between read and write operations,

contributes to a new model that organizes long transactions into the sequences of segments. The model improves the efficiency of processing of long transactions under SI protocol through verification of conflict serializability at the end of each segment.

Because of the complexity of the former approach, a different way to characterize the serializability of Snapshot Isolation, SIMVSG, is presented. After the presentation of advantages of SIMVSG by comparing the features of [3]'s and our approach, we discussed about the dynamically management of SIMVSG, which makes the unserializability of schedule can be detected from SIMVSG as soon as possible.

In the future works, time and space complexity of dynamical verification will be studied and computed formally. An experiment will also be made to give the evidence of efficiency enhancement of long transactions under SI.

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, A Critique of ANSI SQL Isolation Levels. Proc. of the ACM SIGMOD International Conference on Management of Data, 1995. 1-10.

[2] A. Fekete, E. O'Neil, P. O'Neil, A Read-Only Transaction Anomaly Under Snapshot Isolation. *SIGMOD Record, Vol. 33, No. 3, September* 2004.

[3] A. Fekete, Allocating Isolation Levels to Transactions. PODS 2005.

[4] N. A. Lynch, Multilevel atomicity- a new correctness criterion for database concurrency control. *ACM Transaction Database Systems, Vol. 8, No. 4, December* 1983. 484-502

[5] D. Shasha, Transaction Chopping: Algorithms and Performance Studies. *ACM TransactIons on Database Systems, Vol. 20, No. 3, September* 1995. 325-363.

[6] P. A. Berstein, V. Hadzilacos, N. Goodman, *Concurrency control and recovery in database system* (Addison-wesley publishing company, 1987)