Towards consistent snapshot isolation

Nagwa M. El-Makky
Computer and Systems Dept., Faculty of Engg., Alexandria University, Alexandria, Egypt
nagwamakky@alex.edu.eg

Snapshot Isolation (SI) is a widely used optimistic concurrency control algorithm. It is especially beneficial for query intensive applications. Since SI was introduced as a relaxed isolation level, it is well known that it can allow consistency anomalies. In spite of this, SI is the highest level of consistency supported by some of the Popular Database Management Systems (DBMSs). Many organizations use these DBMSs, and so they can be at risk of violation of data consistency. The only way to prevent the consistency anomalies of SI was to statically analyse the application code, and then modify the application accordingly. Recently, a new algorithm, called Serializable Snapshot Isolation (SSI), has been proposed to prevent SI anomalies at runtime. However, under this algorithm, a read-only transaction can be aborted and can cause the abortion of update transactions. This violates one of the most attractive properties of SI. This paper proposes an optimistic concurrency control algorithm that preserves most of the attractive properties of SI while ensuring serialized executions. An analytical performance model is presented for estimating the abortion rate of transactions under the proposed algorithm compared to the SSI and SI algorithms. The model shows that, in terms of transaction abortion rate, the proposed algorithm outperforms SSI and approaches the performance of the original SI in many practical cases.

انتشر مؤخرا استخدام خوارزم "العزل باستخدام صورة للبيانات"، وهو خوارزم متفائل للتحكم فى تزامن العمليات فى نظم قواعد البيانات، وتتضح فائدة هذا الخوارزم تحديدا فى تطبيقات قواعد البيانات التى تغلب عليها الاستفسارات. من المعروف أن هذا الخوارزم يسمح بحدوث بعض الحالات الغير قياسية فى قواعد البيانات الا أن هذا لم يقلل استخدامه فى كثير من نظم قواعد البيانات الشائعة. حتى وقت قريب كانت الطريقة الوحيدة لمنع ظهورهذه الحالات غير القياسية هى تحليل برامج التطبيقات فى وقت التصميم وتعديل البرامج بما يؤدى الى غياب هذه الحالات، الى أن ظهر مؤخرا خوارزم حديث يعدل الخوارزم الأصلى بما يمنع ظهور هذه الحالات فى وقت التشغيل، ولكن هذا الخوارزم الحديث يسمح بانهاء بعض الاستفسارات قبل اتمامها ويسمح بجعل استفسار يتسبب فى انهاء عملية تحديث للبيانات قبل اتمامها مما ينقض احدى المميزات الرئيسية للخوارزم الأصلى. يقترح هذا البحث خوارزما متفائلا للتحكم فى تزامن العمليات فى نظم قواعد البيانات، وهو يعدل الخوارزم الأصلى بما يمنع ظهور الحالات الغير قياسية مع المحافظة على المزايا الأساسية للخوارزم الأصلى، وقد تم عمل نموذج تحليلى للخوارزم المقترح واظهرت النتائج تفوقه على الخوارزم المناظر واقترابه من الخوارزم الأصلى فى كثير من الحالات العملية.

Keywords: Optimistic concurrency control, Snapshot isolation, Serializability theory

1. Introduction

In recent years, many database vendors have built platforms which make use of an optimistic concurrency control technique called Snapshot Isolation (SI), [1]. A transaction executing with Snapshot Isolation always reads data from a snapshot of the committed data as of the time the transaction started. This is available in Oracle RDBMS, PostgreSQL, Microsoft SQL Server and Oracle Berkeley DB. Because SI does not delay reads, even if concurrent transactions have written the data involved, it generally offers higher throughput compared to Strict Two-Phase Locking, (S2PL), [2]. This is especially beneficial for query intensive applications, where a vast amount of complex read-only transactions is conducted together with a small number of short update transactions. Examples of such applications can be found in many practical database-backed web applications, web services, etc.

Since SI was introduced as a relaxed isolation level, it is well known that it can allow some consistency anomalies [1]. In particular, it is possible for a SI-based concurrency control to interleave some transactions, where each transaction preserves an integrity constraint when run alone, and where the final state after the interleaved execution does not satisfy the constraint. In spite of this, not only SI is widely used in many database management systems, but also it is the highest level of consistency supported by popular systems such as Oracle and PostgreSQL which, in fact, use SI even if the user requests the serializable level of isolation [2]. Many organizations use these databases for running their applications, and so, they are potentially at risk of violation of data consistency [2]. Moreover, several data replication techniques, based on snapshot isolation, have been recently proposed [3] which complicates the problem.

The only way to prevent the consistency anomalies of SI was to statically analyse the application code, and then modify the application accordingly by introducing artificial locking or update conflicts [4, 5]. In 2008, a new algorithm, called Serializable Snapshot Isolation (SSI) [6] was proposed to automatically detect and prevent snapshot isolation anomalies at runtime for arbitrary applications. Despite the nice properties of SSI, it allows read-only transactions to abort and to cause the abortion of update transactions. This violates one of the most attractive properties of SI.

This paper proposes an optimistic concurrency control algorithm which prevents the consistency anomalies of SI for arbitrary applications, while preserving most of its attractive properties. Under the proposed algorithm, read-only transactions never wait or abort and they never block or abort update transactions. The proposed algorithm can be easily implemented by adding simple modifications to database management systems that provide snapshot isolation.

Using a simple analytical model and a closed-form average-case analysis, this paper compares the abortion rate of read-only transactions and update transactions under the original SI, i.e, the proposed algorithm and the SSI algorithm. The analysis shows that, in terms of abortion rate, the proposed algorithm outperforms SSI and approaches the performance of the original SI in many practical cases.

The rest of the paper is divided into 6 sections. In section 2, the related work is reviewed. The proposed algorithm is presented in detail in section 3. The correctness of the proposed algorithm is proved in section 4. An analytical performance model is presented in section 5, together with a closed-form average-case analysis for the abortion rate of transactions under the original SI, the proposed algorithm and the SSI algorithm. The results of parametric studies based on the analytical model are presented and discussed in section 6. Finally, the conclusion of the paper is given in section 7.

2. Related work

2.1. Original snapshot isolation [1]

The following is a definition of the SI approach to concurrency control. This definition, which is slightly more formalized than the description introduced in [1], will be used for the purposes of this paper.

A transaction Ti, that is executed under snapshot isolation, is assigned a start timestamp start (Ti) which reflects its starting time. This timestamp is used to define a snapshot Si for transaction Ti. The snapshot Si consists of the latest committed values of all objects of the database at the time start(Ti). Every read operation issued by transaction Ti on a database object x is mapped to a read of the version of x, which is included in the snapshot Si. Updated values by write operations of Ti (which make up the write set of Ti) are also integrated into the snapshot Si, so that they can be read again if the transaction accesses update data. Updates issued by transactions that did not commit before start (Ti) are invisible to the transaction Ti (they are applied to local versions of objects kept in transactions' work spaces). When transaction Ti tries to commit, it is assigned a commit timestamp, commit (Ti), which has to be larger than any other existing start timestamp or commit timestamp. An update Transaction Ti has to pass a validation test. It can successfully commit if and only if there exists no other committed transaction Tk having a commit timestamp commit(Tk) in the interval {start(Ti), commit(Ti)} and write set (Tk) ∩ write set (Ti) ≠ {}. If such a committed transaction Tk exists, then Ti has to be aborted (this is called the first-commiter wins rule [1], which is used to prevent lost updates). If no such transaction exists, then Ti is successfully validated. Its updates will be applied to the database in a write phase and become visible to transactions which have a start timestamp which is larger than commit (Ti); otherwise the updates are discarded and the transaction is restarted.

Even though SI avoids the classically known anomalies such as lost updates or inconsistent reads, there are some consistency anomalies that can occur under SI. Two types of anomalies have been identified in a set of transactions running using SI [2, 7]. The first anomaly type is "write skew, where a set of transactions, each of which preserves some integrity constraint, can execute under SI in a way that leaves the database in a corrupted state. Fekete et al. [7] describe another type of anomaly that they call a "read-only-transaction anomaly". This involves a read-only transaction seeing a state that could not occur in any serialized execution. This violates the previous assumption in [1] that read-only transactions always read consistent data under SI. The next subsection surveys static analysis techniques for removing SI anomalies.

2.2. Removing SI anomalies through static analysis

Because SI can allow data corruption, and is so common, there has been a body of work on how to remove consistency anomalies when running with SI as concurrency control. The main techniques proposed so far [4,5, 8-10] depend on doing a design-time static analysis of the application code and then modifying the application, if necessary in order to avoid the SI anomalies. For example, the work in [4] shows how one can introduce write-write conflicts into the application, so that all SI executions will be serializable.

Making SI serializable using static analysis has a number of limitations [6]. It is unable to cope with ad-hoc transactions and application developers have to be aware of SI anomalies. In addition, this must be a continual activity as an application evolves. In fact, the analysis requires a global graph of transaction conflicts, so every minor change in the application requires renewed analysis and perhaps additional changes (even in programs that were not altered). The next subsection presents a recent proposal [6] which guarantees serializable SI executions for arbitrary applications.

2.3. Serializable snapshot isolation [6]

SSI is a very recent concurrency control algorithm which automatically detects and prevents snapshot isolation anomalies at runtime for arbitrary applications, thus, providing serializable executions. The key idea of the algorithm is to detect, at runtime, conflict patterns that must occur in every non-serializable execution under SI, and abort one of the transactions involved. This is done based on the theory of [11] and its extension in [5], where some distinctive conflict patterns are shown to appear in every non-serializable execution of SI. The building block for this theory is the notion of a read-write dependency edge which occurs from T1 to T2 if T1 reads a version of an object x, and T2 produces a version of x that is later in the version order than the version read by T1. In [11], it was shown that in any non-serializable SI execution, there are two read-write dependency edges in a cycle in the multi-version serialization graph. The work in [5] extended this, to show that there are two read-write dependency edges which form consecutive edges in a cycle, and furthermore, each of these read-write edges involves two transactions that are active concurrently. The serializable SI concurrency control algorithm [6] detects a potential non-serializable execution whenever it finds such two consecutive read-write dependency edges in the serialization graph. Whenever such a situation is detected, one of the transactions will be aborted.

This is similar somehow to the way serialization graph testing works; however the algorithm does not operate purely as a certification at commit-time, but rather aborts transactions as soon as the problem is discovered. Also, the validation test does not require any cycle-tracing in a graph, but can be performed by considering conflicts between pairs of transactions, and a small amount of information which is kept for each of them. The proposed validation test is conservative, so it does prevent every non-serializable execution, but it may sometimes abort transactions unnecessarily. A prototype of the algorithm has been implemented in Oracle Berkeley DB. Evaluating the prototype showed that under a range of conditions, the overall throughput of the algorithm is close to that allowed by SI, and much better than that of strict two-phase locking [6].

Despite the nice properties of this algorithm, it does not give special treatment to read-only transactions. Under the algorithm, a read-only transaction can be aborted and can cause the abortion of update transactions. This violates one of the most attractive properties of snapshot isolation which made it very popular for query-intensive applications. In particular, SSI brings back the source of conflicts between read-only transactions and update transactions (a source that has been eliminated by using the original snapshot isolation).

3. Proposed algorithm

3.1. Idea of the proposed algorithm

The original SI algorithm gives a special treatment for read-only transactions by not involving them in validation tests. Accordingly, a read-only transaction can never abort or cause the abortion of an update transaction. However, unserializable execution does occur due to the nature of the validation test applied to update transactions (which was intended to relax the isolation level). The SSI algorithm eliminates unserializable SI executions by validating each write or read request (whether it comes from an update or a read-only transaction). This violates the special properties given by SI for read-only transactions.

The key design goal for the proposed algorithm was to add simple modifications, to the original SI algorithm, in order to achieve serialized executions while keeping the attractive SI properties for read-only transactions. To keep these properties for read-only transactions, the proposed algorithm treats read-only transactions exactly the same way as the original SI algorithm does. Read-only transactions will read a consistent database state if the execution of update transactions is serialized [12]. To achieve serialized executions, the proposed algorithm was inspired by the multi-version serial validation algorithm described in [13] (it is to be noted that the original SI algorithm itself was inspired by a similar optimistic multi-version algorithm that was described in [12]). However, the algorithm proposed in this paper modifies the algorithm described in [13]. The write phase and the validation phase in [13] are embedded together in a critical section, in order to prevent write-write conflicts. Such a critical section can easily become a bottleneck, especially for disk resident databases. So, the proposed algorithm separates the write phase from the critical section and updates the validation test accordingly. The write phase is handled using a proposed scheme that prevents write-write conflicts.

Using the proposed algorithm, both read-only transactions and update transactions can read consistent snapshots and all executions will be serializable as will be proved in section 4. Read-only transactions can keep the properties that they never wait or abort and never block or abort update transactions. The details of the proposed algorithm will be presented in subsection 3.2. A time complexity analysis of the validation test of the proposed algorithm is given in subsection 3.3.

3.2. Algorithm description

Under the proposed algorithm, transactions have to be declared as read-only transactions or update transactions at start time. As in the original SI, each transaction Ti (whether it is a read-only or an update transaction) is assigned a start timestamp; start(Ti), which reflects it's starting time. The proposed algorithm works exactly as the original SI algorithm for read-only transactions. It works also exactly as the original SI algorithm for update transactions until the update transaction requests committing.

When an update transaction Ti is to commit, it is assigned a commit timestamp, commit (Ti), which has to be larger than any other existing start timestamp or commit timestamp. An update transaction Ti has to pass a validation test (in a critical section) to be successfully committed. Since the write phase is separated from the critical section, the validation test will be as given below.

Assume that Last(x) is the commit timestamp of the most recent successfully validated transaction that has x in its write set. Ti passes the validation test successfully if and only if start (Ti) > Last (x) for each object x in Ti 's read set.

To commit the write set of the transaction to the database while avoiding the lost update problem, a proposed scheme is given below for allowing concurrent write phases while avoiding write-write conflicts.

During the validation phase of transaction Ti, a Wait-For-List (WFL) is determined for Ti that identifies the update transactions whose write phases conflict with and therefore must be processed before the write phase of Ti. The following is an outline of how to compute and use the WFL. If Ti is successfully validated, then for each object x belonging to the write set of Ti, add Last(x) (if it corresponds to a validated but not yet completed transaction) to WFL (Ti) and then let Last(x) be the commit timestamp of Ti. If WFL (Ti) is empty, then Ti can begin its write phase, where it stamps the created versions of it's write set objects by its commit timestamp. Otherwise, T_i has to wait until the conflicting update transaction(s) have been processed. Algorithms 1 and 2 show pseudo code for validating and committing a transaction Ti, under the proposed algorithm, respectively.

Algorithm 1: Validating a Transaction T_i

```
valid = true; WFL ($T_i$) = {};
for each x in $T_i$'s read set do
  if start ($T_i$) < Last(x) then valid =false and exit loop;
if valid =false then
  discard $T_i$'s updates and restart $T_i$;
else
   for each object x in $T_i$'s write set do
     if Last(x) corresponds to an uncompleted transaction then
       add Last(x) to WFL($T_i$);
     Last (x) = commit($T_i$) ;
   end for
   if WFL($T_i$) is empty then
      commit Transaction $T_i$ ;
   else wait;
```

Algorithm 2: Committing a Transaction T_i

```
for each object x in $T_i$'s write set do
  stamp the created version of x by the commit timestamp of $T_i$ ;
begin
commit the write set of $T_i$ to the database;
find all Wait-For-Lists (WFLs) for other transactions that $T_i$ belongs
to;
remove $T_i$ from these WFLs;
if any WFL($T_k$) becomes empty then wakeup $T_k$ to commit;
end
```

It is to be noted that the proposed algorithm is more conservative than the original SI algorithm. It does prevent every unserializable execution, but it may sometimes abort a transaction unnecessarily (the probability of transaction abortion for both the original SI and the proposed algorithm will be calculated in section 5). This is the cost paid for obtaining serialized executions with a validation test that has constant time complexity (time complexity of the proposed validation test will be calculated in subsection 3.3). An alternative solution is to design a non- conservative validation test, which aborts an update transaction only when an operation will result in a non-serializable execution; this would be a serialization-graph-testing algorithm. However, serialization-graph testing requires expensive cycle detection calculations, and would be very prohibitive [6].

It has also to be noted that although the proposed scheme for concurrent write phases depends on blocking update transactions, current SI implementation for concurrent write phases in some database management systems such as Oracle and PostgreSQL[6] depend also on blocking. Moreover, the proposed scheme has the advantages of achieving serialized schedules besides being deadlock-free, in contrast to the above implementations.

3.3. Validation complexity analysis

In this subsection, the time complexity of the validation test of the proposed algorithm is analysed and compared to that of the original SI algorithm.

The unit of cost for this analysis will be the cost of a probe into a set of items organized as a hash table in main memory. For the original

SI algorithm, it is assumed that the most recent timestamps of objects' versions written by recently committed transactions are stored in a hash table. It is also assumed that Last(x) for each x in the write sets of recently validated transaction are stored in a hash table for the proposed algorithm. For simplicity, it is assumed that all update transactions have the same fixed read set and write set sizes. Let R_u be the size of update transactions read sets and let W be the size of their write sets.

For the proposed algorithm, validating a transaction T requires that its start timestamp be compared to the timestamp Last(x) for each item x in T's read set. This requires R_u probes into the corresponding hash table. In addition, if T passes the validation test successfully, another W probes into the table will be required to add Last(x) to WFL(T_i) (if necessary) and to update Last (x), for each object x in T's write set. Therefore, the total validation cost is $R_u + W$.

Regarding the original SI algorithm, validating T requires that the start timestamp of T be compared to the most recent commit timestamp of each item x in T's write set. If T passes the validation test successfully, it will be required to update the most recent commit timestamp of each item x in T's write set. This requires W probes into the corresponding hash table. Therefore the total validation cost is W.

It is clear that, similar to the original SI algorithm, the proposed algorithm has a constant time complexity (that is independent of the number of recently committed transactions; n) for the validation test.

4. Correctness proof

In this section, it is proved that the proposed algorithm guarantees serializability. First, subsection 4.1 gives a formal model for correctness, and then subsections 4.2 and 4.3 give the required proof.

4.1. The formal model for correctness

The original snapshot isolation algorithm is an instance of multiversion concurrency control [1]. Since the proposed algorithm is a modification of the original snapshot isolation algorithm, it is also an instance of multiversion concurrency control. The following gives a formal model for correctness of multiversion concurrency control algorithms.

A database consisting of a set of objects is assumed**.** Users interact with the database system by invoking transaction programs. A transaction T_i, is an ordered pair ($\sum_i$, $<_i$,), where $\sum_i$, is the set of read and write operations in T_i, that are executed atomically, and $<_i$, is a partial order that represents the execution order of these operations. Read and write operations executed by T_i, on an object x are denoted by $r_i[x]$ and $w_i[x]$, respectively. The set of transactions that executed in a system is denoted as $T = \{T_1, \ldots . T_n\}$. The execution of transactions in T is modelled by a structure called schedule. A schedule, H, over T is defined as a partial order ($\sum$, $<_H$) , where $\sum$ is the set of all operations executed by transactions in T , and $<_H$ indicates the execution order of those operations.

The database is assumed to be multiversion, in which each write operation on an object x produces a new version of x. Thus for each object x in the database, there is a list of associated versions. A read operation on x is performed by returning the value of x from an appropriate version in the list.

A multiversion (*MV*) schedule H over a set of transactions T represents the sequence of operations on the versions of objects. Thus each $w_i[x]$ in an *MV* schedule is mapped into $w_i[x_i]$, and each $r_i[x]$ is mapped into $r_i[x_j]$, for some j (which is determined by the multiversion concurrency control algorithm) . A transaction T_j reads x from T_i, in H if T_j reads a version of x produced by T_i.

Two *MV* histories over a set of transactions are equivalent if they have the same operations [12]. An *MV* schedule is one-version serializable if it is equivalent to a serial schedule over the same set of transactions executed over a single version database [12].

The serialization graph of an *MV* schedule H, $SG(H)$, is a directed graph whose nodes represent transactions and whose edges are all $T_i \rightarrow T_j$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in *H.* However, *SG*(*H*) by itself does not contain enough information to determine whether *H* is one-version serializable or not. To determine if an *MV* schedule is one-version serializable, a modified serialization graph is used [12]. Given an *MV* schedule *H,* a multiversion serialization graph (*MVSG*(*H*)) is a *SG*(*H*) with additional edges such that the conditions, given below, hold.

1- For each object x, *MVSG*(*H*) has a total order (denoted $<<_x$ on all transactions that write x).

2- For each object x , if T_j reads x from T_i , and if $T_i <<_x T_k$, then *MVSG*(*H*) has an edge from T_j to T_k (i.e., $T_j \rightarrow T_k$) *;* otherwise, if $T_k <<_x T_i$, then *MVSG*(*H*) has an edge from T_k to T_i, (i.e. $T_k \rightarrow T_i$).

The additional edges are called version order edges. An *MV* schedule *H* is one-version serializable if and only if *MVSG*(*H*) is acyclic[12].

It was proved in [12] that if read-only transactions satisfy a set of conditions and the used multiversion concurrency control algorithm serializes update transactions, then every read-only transaction will read a consistent state of the database. So, proving the correctness of the proposed algorithm consists of proving the one-version serializability of update transactions' schedules and then proving that each read-only transaction satisfies the mentioned conditions and hence sees a consistent state of the database. These proofs are given in subsections 4.2 and 4.3, respectively.

4.2. Correctness proof of update transactions schedules

The following lemmas state certain properties of the proposed algorithm. These properties will be used to prove that update transactions schedules produced by the proposed algorithm are one-version serializable (as stated in subsection 4.1, the analysis needs to be done only for update transactions).

Lemma 1: For every $r_k[x_j]$, $w_j[x_j] < r_k[x_j]$ and commit (T_j) < commit (T_k).

Proof: according to the proposed algorithm definition, the execution of $r_k[x_j]$, will return the version x_j with the largest timestamp such that commit (T_j) < start (T_k). Since start (T_k) < commit (T_k), it follows that commit (T_j) < commit (T_k).

Lemma 2: For every $r_k[x_j]$ and $w_i[x_i]$, i ± j, one of the following conditions must hold:

1- commit (Ti) < commit(T_j), or

2- commit (T_k) < commit(T_i), or

3- i = k and $r_k[x_j] < w_i[x_i]$.

Proof: Assume that i ± k. $r_k[x_j]$ implies that commit (T_j) < commit (T_k), according to Lemma 1. Having commit (T_j) < commit (T_i) < commit (T_k) is impossible, otherwise T_k should have been aborted and $r_k[x_j]$ will not exist in the schedule. Therefore, either commit (Ti) < commit(T_j), or commit (T_k) < commit(T_i). The case i = k holds according to the definition of the proposed algorithm (see subsection 3.2).

By using the above lemmas as formal specifications of the proposed algorithm, the following theorem demonstrates that the proposed algorithm guarantees one-version serializable executions of update transactions.

Theorem 1: The proposed algorithm guarantees one-version serializable executions of update transactions.

Proof: Define the version order $<<_x$ for an object x as the total order on the commit timestamps, of the transactions creating versions of x, i.e., $x_i <<_x x_j$ if and only if commit (Ti) < commit(T_j).

Let *H* be a schedule of update transactions produced by the proposed algorithm. It will be proved that *MVSG(H)* is acyclic by showing that for each edge $T_i \rightarrow T_j$ in *MVSG*(*H*), commit (T_i) < commit (T_j).

Recall that *MVSG*(*H*) includes edges in *SG*(*H*) and additional version order edges. First, consider an edge $T_i \rightarrow T_j$ in *SG* (*H*). Each such edge is due to a reads-from relationship; i.e., T_j has read some object written by T_i. By Lemma 1 of the proposed algorithm, it follows that commit (T_i)< commit (T_j).

Next, consider a version order edge in *MVSG* (*H*). Let $r_k[x_j]$ and $w_i[x_i]$ be in H , where i,j, and k are distinct. Consider the cases given below.

1- $x_i <<_x x_j$, which implies that $T_i \rightarrow T_j$ is in *MVSG (H)*;

2- $x_j <<_x x_i$, which implies that $T_k \rightarrow T_i$ is in *MVSG* (*H*).

In case 1, from the definition of version order, commit (T_i) < commit (T_j). In case 2, from Lemma 2, it follows that commit (Ti) <

commit(T_j), or commit (T_k) < commit(T_i). Since $x_j <<_x x_i$, commit (Ti) < commit(T_j) is not possible. Hence, commit (T_k) < commit(T_i).

If *MVSG*(*H*) has a cycle, it violates the total order of the commit timestamps of transactions involved in that cycle. Thus by the application of the serializability theorem for multiversion databases, every schedule of update transactions produced by the proposed algorithm is one-version serializable.

4.3. Correctness proof of the read-only transactions synchronization scheme

It was proved in [12] that if read-only transactions satisfy the conditions given below and the used concurrency control algorithm serializes update transactions, then every read-only transaction will read a consistent state of the database. Since the proposed concurrency control algorithm was proved to serialize update transactions, it is sufficient to show that read-only transactions satisfy the given conditions under the proposed algorithm.

Condition 1: a read-only transaction reads the output of transactions that have committed or that will eventually commit.

Condition 2: If T_i belongs to the set of update transactions from which a read-only transaction T_q reads and if T_i depends on T_j , then T_j must belong to the set of update transactions from which T_q reads (see the definitions given below for the dependency relationship between transactions).

Definition 1: A transaction T_i directly depends on transaction T_j if there exists some x such that T_i reads x from T_j.

Definition 2: A transaction T_i depends on transaction T_j if T_i directly depends on T_j, or if there is a sequence T_1, T_2,..., T_n such that T_i directly depends on T_1 , T_1 directly depends on T_2,….and T_n directly depends on T_j.

The read-only transactions synchronization scheme of the proposed algorithm satisfies condition 1 because read-only transactions read only from committed update transactions (according to the definition of the proposed algorithm in subsection 3.2). Condition 2 is also satisfied as will be proved below.

Proof: the direct case of condition 2 will be proved, namely if T_i belongs to the set of update transactions from which a read-only transaction T_q reads, and if T_i directly depends on T_j, then T_j must also belong to the set of update transactions from which T_q reads. Proof of the indirect case (T_i depends on T_j) follows easily by induction.

T_i belongs to the set of update transactions from which a read-only transaction T_q reads, means that commit (T_i)< start (T_q) (according to the definition of the proposed algorithm). The fact that T_i directly depends on T_j means that T_i has read the output of T_j. Using the definition of the proposed algorithm, this is possible only if T_j has been committed before the start of T_i; i.e., commit (T_j) < start (T_i). This means that commit (T_j) < start (T_q); i.e., T_j belongs to the set of update transactions from which T_q reads.

Thus, any read-only transaction will read a consistent state of the database under the proposed algorithm.

5. Analytical performance model

This section presents an analytical model for assessing the performance (in terms of transaction abortion rate) of the proposed algorithm compared to the original snapshot isolation algorithm and the serializable snapshot isolation algorithm. A simple model for single-site database systems, similar to that in [14], is used. A closed –form average-case analysis is performed to estimate the abortion rate of transactions under the above algorithms.

5.1. The analytical model

A single-site database system is assumed with a set of distinct database objects. The total number of database objects is referred to as *DBSize*. It is assumed that access to these objects is uniform (there are no hot spots). Assume that *TPSupdate* update transactions are originated per second and that *TPSquery* read-only transactions are originated per second. Each transaction is assumed to involve a fixed number of read / write actions, with a fixed time, *Action_Time*, for each one. Each update transaction reads R_u data objects, updates W data objects, and takes L_u seconds to finish. Each read-only transaction reads R_q data objects and takes L_q seconds to finish. A transaction's duration (L_u or L_q) is estimated as the number of the transaction's read/write actions multiplied by the *Action_Time*[14]. Table 1 lists the model parameters.

In subsections 5.2, 5.3, 5.4 and 5.5, an average-case analysis, based on that in [14], is conducted to estimate the abortion rate of transactions under the original SI algorithm, the proposed algorithm and the SSI algorithm, respectively.

5.2. Snapshot Isolation (SI) abortion rate

Under the original snapshot isolation algorithm, read-only transactions never abort. Therefore, it is only necessary to estimate the probability of abortion of update transactions.

Table 1
Parameters of the analytical model

Symbol	Meaning
DBSize	Total number of data objects
TPSupdate	Number of originated update transactions per second
TPSquery	Number of originated read-only transactions per second
Action_Time	Time to perform a read/write action
R_u	Read set size of update transactions
W	Write set size of update transactions
R_q	Read set size of read-only transactions
L_u	Duration of a single update transaction (in seconds)
L_q	Duration of a single read-only transaction (in seconds)

On the average, number of update transactions that commit in time L_u can be expressed as $TPSupdate * L_u$. Hence, the number of writes in that time = $W*(TPSupdate* L_u)$. Since objects are chosen uniformly from the database, the probability that a specific update of a transaction T conflicts with one of these writes = (number of writes)/(database size)= $W*(TPSupdate*L_u)/DBsize$. If any such conflict occurs, transaction T must abort according to the rules of SI. Since T, has W updates, it follows that the probability, P_{SI}, that an update transaction T aborts= $W*$ (probability of a single conflict)=$(W^2*TPSupdate*L_u)/DBSize$.

The rate of aborted update transactions can be calculated as: (rate of update transactions) *(probability of one transaction abortion)=$TPSupdate*(W^2*TPSupdate*L_u)/DBSize$.

Therefore, the abortion rate of update transactions at the database site under the SI algorithm, AR_{SI}, can be represented as:

$$AR_{SI} = (TPSupdate* W)^2 * (L_u/DBSize). \quad (1)$$

5.3. The Proposed Algorithm (PA) abortion rate

Like the original snapshot isolation algorithm, the proposed algorithm never aborts read-only transactions. According to the algorithm description in subsection 3.2, an update transaction T aborts if any update transaction that committed after the start of T has a write set that intersects with T's read set. Similar to the analysis in section 5.2, it follows that the probability that an update transaction T has to abort can be approximated as $(W*R_u*TPSupdate*L_u)/DBSize$. Also, the abortion rate of update transactions at the database site, under the proposed algorithm, AR_{PA}, can be approximated as:

$$AR_{PA} = (TPSupdate)^2 * W *R_u* L_u)/DBSize. \quad (2)$$

It is to be noted that if each update transaction writes all the objects it has read, i.e., if $W = R_u$, then the abortion rate of update transactions under the proposed algorithm will be equal to the corresponding abortion rate under the original snapshot isolation algorithm.

5.4. Abortion rate of read- only transactions under the SSI algorithm

The SSI algorithm detects a potential non-serializable execution whenever it finds two consecutive read-write dependency edges in the serialization graph. Whenever such a situation is detected, one of the transactions will be aborted. The aborted transaction can be a read-only or an update transaction.

The abortion rate of read-only transactions is estimated in this subsection, while the abortion rate of update transactions is estimated in the next subsection.

From the definition of the SSI algorithm in subsection 2.3, it can be deduced that the probability of abortion of a read-only transaction T is the probability that there are two consecutive read-write dependency edges in the serialization graph such that the first edge is created by T. In particular, this means that the first edge is the result of a read/write conflict between T and one of the update transactions T' that has committed since the start of T, and the second edge is due to a read/write conflict between T' and one of the update transactions that has committed since the start of T'.

To calculate the probability of creating such a first edge, consider one of the reads of a read-only transaction T and assume that it is in the transaction halfway. On the average, number of update transactions that commit in time $L_q/2$ can be expressed as $(TPSupdate*L_q)/2$. Hence, the number of writes in that time $=W* (TPSupdate *L_q/2)$. The probability that a specific read operation of T conflicts with one of these writes = (number of writes)/(database size)= $W*TPSupdate*L_q/(2DBsize)$. T has R_q such read requests, so the probability that it will conflict sometime in its lifetime (i.e., the probability of creating the first edge) can be approximated as $(R_q*W*TPSupdate*L_q)/(2\ DBsize)$.

Using similar analysis to that in subsection 5.3, it can be seen that the probability of creating the second edge equals $(W*R_u* TPSupdate* L_u)/DBSize$.

Therefore, the probability of aborting a read-only transaction can be approximated as: $(R_q* W*TPSupdate*L_q/2)*(W*R_u*TPSupdate*L_u)/(DBSize)^2$.

Since $L_q = R_q * Action\text{-}Time$ and $L_u =(R_u+W)* Action\text{-}Time$, the probability of aborting a read-only transaction(query) under the SSI algorithm, APQ_{SSI}, can be approximated as:

$$APQ_{SSI}=(R_q*TPSupdate*W*Action\text{-}Time/DBSize)^2 *R_u(R_u+W)/2. \quad (3)$$

Hence, the abortion rate of read-only transactions under the SSI algorithm, ARQ_{SSI}, can be approximated as:

$$ARQ_{SSI} = R_q*TPSupdate*W*Action\text{-}Time/DBSize)^2 *TPSquery* R_u(R_u+W)/2. \quad (4)$$

The above analysis points to a serious problem with the serializable snapshot isolation algorithm (that has been already eliminated in the original SI algorithm and the proposed algorithm). Read-only transactions can be aborted due to conflicts with update transactions. Moreover, the abortion rate of read-only transactions rises as the second power of the following factors: the read-only transaction size, the rate of originated update transactions and the size of the write set of update transactions. A ten-fold increase in any of these factors increases the abortion rate by a factor of 100.

5.5. Abortion rate of update transactions under the ssi algorithm

The abortion rate of update transactions can be deduced from the definition of the SSI algorithm in [6]. An update transaction can be aborted during one of its read operations, one of its write operations or at commit time.

The probability, P_1, that an update transaction T is aborted during one of its read operations can be obtained using a similar analysis to that for finding the abortion probability of a read-only transaction during one of its reads. P_1 can be approximated as:

$$P_1 = (R_u* TPSupdate*W* L_u / DBSize)^2 /2. \quad (5)$$

The probability, P_2, that an update transaction *T* is aborted during one of its writes, is the probability that there are two consecutive read-write dependency edges in the serialization graph such that the second edge is created by T. In particular, the second edge has to be the result of a read/write conflict concerning any object, x, between T and one of the committed update transactions T' that has a snapshot isolation read (SIREAD) lock on x(according to the terminology of [6]) and has committed since the start of T. The first edge has to be the result of a read/write conflict between T' and any concurrent read-only or update transaction.

The following calculates the probability of creating the first edge. On the average, there are $TPSquery*L_q$ read-only transactions and $TPSupdate*L_u$ update transactions to conflict with T'. Assume that each is about half way complete. On the average, the number of read operations performed by these transactions can be expressed as $(TPSquery*L_q*R_q + TPSupdate*L_u*R_u)/2$. Since objects are chosen uniformly from the database, the chance that an update by T' will conflict with one of these reads is: $(TPSquery*L_q*R_q+TPSupdate*L_u*R_u)/(2*DBSize)$.
T' has W updates, so the probability, P_{21}, of a read/write conflict with T' (which is the probability of creating the first edge) can be approximated as:

$$P_{21}= TPSquery*L_q*R_q+TPSupdate*L_u*R_u) * (W/2DBSize). \qquad (6)$$

Regarding the second edge, there are $TPSupdate*L_u$ update transactions to conflict with T. Assume that each of them is about half way complete. On the average, the number of read operations performed by these transactions are $TPSupdate*L_u*R_u/2$. Therefore, the probability that one of the write operations of T conflicts with these reads is $TPSupdate*L_u*R_u/(2*DBSize)$. T has W updates, so the probability, P_{22}, of a read/write conflict with T (which is the probability of creating the second edge) can be approximated as:

$$P_{22} = W* TPSupdate*L_u*R_u/(2*DBSize). \qquad (7)$$

Since the two edges are independent, the probability, P_2, of aborting an update transaction during one of its writes can be calculated by multiplying the probabilities P_{21} and P_{22} as:

$$P_2 = (TPSquery*L_q*R_q + TPSupdate*L_u*R_u) * (R_u*TPSupdate*L_u/4) * (W/DBSize)^2. \qquad (8)$$

Finally, an update transaction T can be aborted at commit time. This can occur if T does not pass the validation test of the original SI algorithm or if there are two consecutive read-write dependency edges in the serialization graph such that T is the pivot for these edges. It is only required to calculate the probability of the second condition, since the probability of the first condition is the same as P_{SI} (the probability of abortion of update transactions under the original SI algorithm which was calculated in subsection 5.2).

Regarding the probability of the second condition, the first edge of the two consecutive ones has to be the result of a read/write conflict between a concurrent read-only transaction or an update transaction and the transaction T. The second edge has to be the result of a read/write conflict between T and an update transaction that has committed since the start of T.

The probability of creating such a first edge has been calculated before as P_{21}. Using similar analysis to that in subsection 5.3, it follows that the probability of creating such a second edge is $(W*R_u*TPSupdate*L_u)/DBSize$.

Hence, the probability, P_3, that an update transaction is aborted due to the existence of two independent consecutive edges of the above types, can be approximated as:

$$P_3 = (TPSquery*L_q*R_q + TPSupdate*L_u*R_u) * (R_u*TPSupdate*L_u/2) * (W/DBSize)^2. \qquad (9)$$

Hence, the total abortion probability of an update transaction under the SSI algorithm, APU_{SSI}, can be approximated as:

$$\begin{aligned} APU_{SSI} = {} & P_1 + P_2 + P_3 + P_{SI} = \\ & (R_u*TPSupdate*W*L_u/DBSize)^2/2 + \\ & (TPSquery*L_q*R_q + TPSupdate*L_u*R_u) * \\ & (R_u*TPSupdate*L_u/4) * (W/DBSize)^2 + \\ & (TPSquery*L_q*R_q + TPSupdate*L_u*R_u) * \\ & (R_u*TPSupdate*L_u/2) * (W/DBSize)^2 + \\ & (W^2*TPSupdate*L_u)/DBSize. \qquad (10) \end{aligned}$$

Simplifying the above expression a little gives the abortion probability of update transactions as:

$$APU_{SSI} = (R_u * TPSupdate*W* L_u / DBSize)^2 /2 +0.75*(TPSquery * L_q* R_q + TPSupdate*L_u* R_u) * (R_u* TPSupdate* L_u) * (W / DBSize)^2 + (W^2 * TPSupdate* L_u)/DBSize. \quad (11)$$

Therefore, the abortion rate of update transactions under the SSI algorithm can be approximated as:

$$ARU_{SSI} = TPSupdate^3 * (R_u *W* L_u / DBSize)^2 /2 +0.75*(TPSquery * L_q* R_q + TPSupdate*L_u* R_u)* (R_u* TPSupdate^2 * L_u) * (W / DBSize)^2 + (TPSupdate * W)^2 *(L_u / DBSize). \quad (12)$$

The above analytical model has been used in some parametric studies to evaluate the performance of the proposed algorithm compared to the original SI and the SSI algorithms, in terms of transactions' abortion rate. These parametric studies are presented in the next section.

6. Parametric studies and results

In this section the results of three parametric studies, based on the analytical model of section 5, are presented. The studies were performed to compare the performance (in terms of transactions' abortion rate) of the proposed algorithm to the original SI algorithm and the SSI algorithm under different parameters. Typical values of parameters for workloads on a single database site were mainly obtained using the Transaction Processing Council benchmark, TPC-W [15] as a reference. This can be explained knowing that the proposed algorithm targets query-intensive applications, e.g., many database-backed web applications and web services, and TPC-W is a transactional web e-Commerce benchmark. However, the values of some of these parameters have been varied in some studies to study the effect of this variation on the performance of the algorithms. The first study examines the three algorithms under a workload for which the original SI is expected to be beneficial; a mix of small update transactions and larger read-only transactions. The second study investigates the effect of varying the fraction of update transactions in the previous workload on the behaviour of the algorithms. The third study examines the algorithms under a less favourable workload. The size of update transactions is somewhat larger, and the ratio of writes to reads in update transactions is varied in order to compare the performance of the proposed algorithm to the other algorithms under varying conflict probabilities.

6.1. Effect of varying the size of read-only transactions

This study examines the behaviour of the three algorithms under a mix of transactions for which the original SI algorithm was designed to be beneficial [1]. The mix used consists of update transactions and dominating read-only transactions. Update transactions are small, and the size of read-only transactions is varied from small to large as a fraction of the overall database size. The workload parameter settings for the study are given in table 2.

Table 2
Parameter settings for the first parametric study

Parameter	Value / Range
Database size (total number of data objects)	500,000
Range of read-set sizes for read-only transactions (in data objects)	5-50
Read-set and write-set sizes of update transactions (in data objects)	2,2 (respectively)
Action time (in seconds)	0.01
Number of originating update transactions per second	2000
Number of originating read-only transactions per second	8000

As can be seen from table 2, twenty percent of the transactions are update transactions, reading and then updating two data objects, and the other 80 percent are read-only transactions. Each read-only transaction reads a fixed number of objects, and the size of these transactions is varied from 5 data objects up to 50 data objects. The performance metric used is the per-class transaction abortion rate (i.e., the read-only transactions abort rate and the update transactions abort rate). Abortion rates are measured in transactions/second.

Given the small size of the update transactions in this study, almost all conflicts are between read-only and update transactions. Since the proposed algorithm and the original SI algorithm completely eliminate this source of conflicts, the read-only transactions abortion rates for both algorithms equal zero for all values of the read-only transaction size, as can be seen from fig. 1. However, under the SSI algorithm, as the read-only transaction size increases, read-only transactions quickly begin to be aborted because of conflicts with update transactions in the workload. A ten-fold increase in the size of the read-only transaction increases the abortion rate by a factor of 100 as can be seen from fig. 1. In fact, the abortion of read-only transactions is a weakness point of the SSI algorithm.

As expected, increasing the size of read-only transactions has no effect on the abortion rate of update transactions under the original SI algorithm and the proposed algorithm. However, increasing the size of read-only transactions, under the SSI algorithm, increases the abortion rate of update transactions as can be seen from fig. 2. This can be easily explained using the analysis in subsection 5.5.

Under the SSI algorithm, increasing the size of read-only transactions increases the probability that an update transaction will be aborted during one of its writes or at commit time (due to a conflict with a read-only transaction). Consequently, the abortion rate of update transactions will increase.

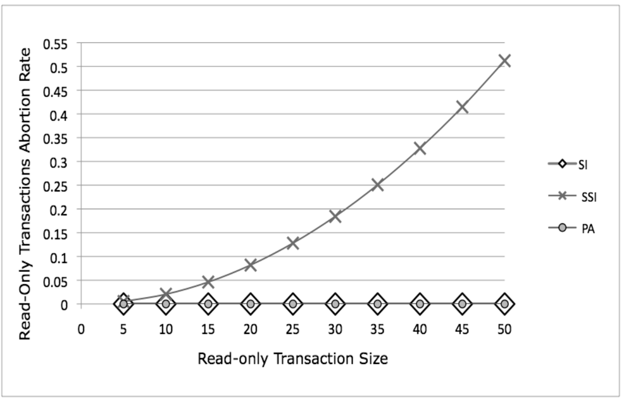

Fig 1. Read-only transactions abortion rate for the SI algorithm, the SSI algorithm and the Proposed Algorithm (PA).

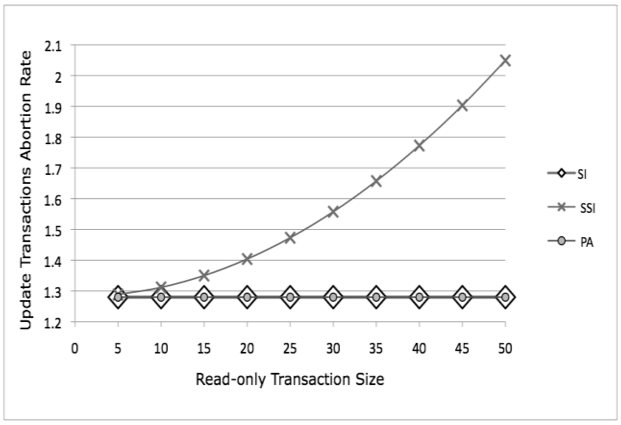

Fig. 2. Update transactions abortion rate for the SI algorithm, the SSI algorithm and the Proposed Algorithm (PA).

It is to be noted that the update transaction abortion rate for the proposed algorithm is identical to that for the original SI algorithm. This can be deduced from the analysis in section 5, knowing that the ratio of writes/reads in update transactions equals 1 in this parametric study.

*6.2. Effect of varying the fraction of update transactio*ns

This study examines the behaviour of the algorithms under a mix of transactions similar to that of the first study. However, in this study, the size of read-only transactions is held fixed (at 50 data objects). The variable here is the fraction of update versus read-only transactions in the mix. In particular, workloads with 0, 20, 40, 60, 80, and 100 percent update transactions are studied, with the remainder of the workload consisting of read-only transactions. The values of other parameters are identical to those used in the first study. The performance metric used is the expected abortion rate of transactions, which is the weighted average of the abortion rates of read-only transactions and update transactions. The point of this study is to find out how the fraction of update transactions affects the expected abortion rate of transactions under the three algorithms.

Fig. 3 shows the results for the expected abortion rate of transactions under the three algorithms. Since the ratio of writes/reads in update transactions equals 1, the abortion rate of the proposed algorithm is identical to that of the original SI algorithm as can be deduced from the analysis in subsection 5.3. As expected, when the mix has no update transactions, the abortion rates of all algorithms equal zero. The expected abortion rates of the three algorithms increase as the fraction of update transactions increases. This is due to the increase in the probability of conflicts between transactions. Among the 3 algorithms, the SSI algorithm is the only one that allows conflicts between update and read-only transactions in addition to conflicts between update transactions. This explains the higher abortion rates for the SSI algorithm compared to the other two algorithms. When the fraction of update transactions reaches 1, i.e., in case of absence of read-only transactions which represent a weakness point of the SSI algorithm, its performance becomes very close to that of the other two algorithms.

6.3. *Effect of varying the writes/reads ratio of update transactions*

The aim of this study is to evaluate the behaviour of the proposed algorithm under less favourable conditions. The size of update transactions is somewhat larger, the update transactions fraction in the workload is 80%, and the ratio of writes to reads in update transactions is varied in order to evaluate the performance of the proposed algorithm, compared to other algorithms under different conflict probabilities. The workload parameter settings for this study are given in table 3. The performance metric used is the expected abortion rate of transactions. Fig. 4 gives the results for the expected abortion rate of transactions under the three algorithms.

From fig. 4, it can be shown that the original SI algorithm and the SSI algorithm outperform the proposed algorithm at low ratios of writes/reads of update transactions. This can be explained knowing that the proposed algorithm is more conservative than the other two algorithms. For a low writes/reads ratio, this causes an overhead

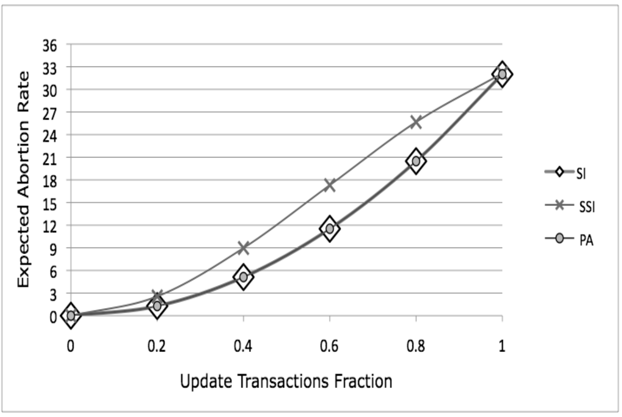

Fig. 3. Expected transaction abortion rate for the SI algorithm, the SSI algorithm and the Proposed Algorithm (PA).

Table 3
Parameter settings for the third parametric study

Parameter	Value / Range
Database size (in data objects)	500000
Read-only transaction size (in data objects)	50
Read-set size of update transactions(in data objects)	6
Ratio of writes/reads	1/6, 1/3, 1/2, 2/3, 5/6 and 1
Action time (in seconds)	0.01
Number of originating update transactions per second	8000
Number of originating read-only transactions per second	2000

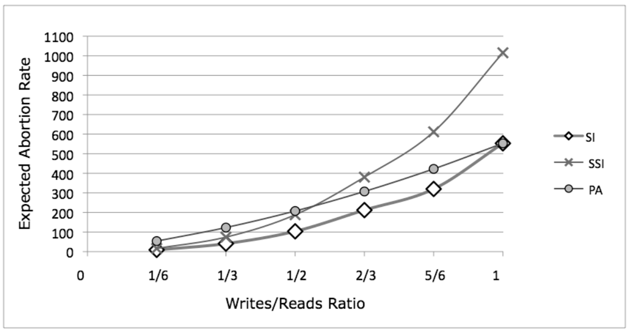

Fig. 4. Expected transaction abortion rate for the SI algorithm, the SSI algorithm and the Proposed Algorithm (PA).

due to unnecessary aborts. However, as the writes/reads ratio of update transactions increases, the conflicts between update transactions increase under all the algorithms and the conflicts between update and read-only transactions increase under the SSI algorithm.

This explains why the proposed algorithm outperforms the SSI algorithm for high ratios of writes/reads of update transactions. As the ratio of writes/reads becomes close to 1, the performance of the SSI algorithm continues to degrade, giving its worst performance when the ratio equals 1.

On the other hand, as the ratio of writes/reads approaches 1, the abortion rate of the proposed algorithm approaches that of the original SI algorithm as can be deduced from the analysis of subsection 5.3.

7. Conclusions

This paper proposes an optimistic concurrency control algorithm that avoids the consistency anomalies of the original Snapshot Isolation algorithm. It is based on simple modifications of the original SI algorithm and preserves most of its attractive properties. Under the proposed algorithm, read-only transactions never wait or abort and they never block or abort update transactions.

The paper presents a simple analytical performance model and a closed-form average-case analysis for the abortion rate of transactions under the proposed algorithm compared to the original SI algorithm and the SSI algorithm. Parametric studies were performed using the analytical model to have an insight into the behaviour of the proposed algorithm compared to the other two algorithms.

From the parametric studies, it is clear that, in terms of transaction abortion rate, the proposed algorithm outperforms the SSI algorithm and approaches the performance of the original SI algorithm for query-intensive workloads. These are the workloads that the original SI algorithm was designed to be beneficial for. In these workloads there is a vast amount of large read-only transactions conducted together with a small number of short update transactions. Examples of such workloads can be found in many practical database-backed web applications and web services.

For less favourable workloads, the proposed algorithm outperforms the SSI algorithm at high conflict cases. For low conflict cases, e.g., a low ratio of writes/reads in update transactions, the proposed algorithm can cause an overhead due to unnecessary aborts. In such cases, the SSI algorithm can be a viable choice.

As a future work, it is planned to perform a detailed performance evaluation study of the proposed algorithm compared to the original snapshot isolation algorithm and the serializable snapshot isolation algorithm.

References

[1] H. Berenson, P. Bernstien, J. Gray, J. Melton, E. O'Neil and P. O'Neil, "A Critique of ANSI SQL Isolation Levels", In Proc. of SIGMOD' 95, ACM Press, June (1995).
[2] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the Detection of Snapshot Isolation Anomalies", In Proc. of VLDB, ACM Press, September (2007).
[3] E. Cecchet, G. Candea and A. Ailamaki, "Middleware-Based Replication: the Gaps Between Theory and Practice", In Proc. of SIGMOD'08, ACM Press, June (2008).
[4] Fekete. Allocating Isolation Levels to Transactions. In Proc. of PODS'05, ACM Press (2005).

[5] Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha, "Making Snapshot Isolation Serializable", ACM Transaction on Database Systems, Vol. 30 (2) (2005).

[6] M. Cahill, U. Rohm and A. Fekete. Serializable Isolation for Snapshot Databases. In Proc. of SIGMOD'08, ACM Press, June (2008).

[7] Fekete, E. O'Neil and P. O'Neil. A Read-Only Transaction Anomaly under Snapshot Isolation", ACM SIGMOD Record, Vol. 33 (3) (2004).

[8] Fekete. Serializability and Snapshot Isolation. In Proc. of Australian Database Conference, Australian Computer Society, January (1999).

[9] Bernstein, P. Lewis and S. Lu, "Semantic Conditions for Correctness at Different Isolation Levels", In Proc. of IEEE International Conference on Data Engineering, IEEE, February (2000).

[10] Alomari, M. Cahill, A. Fekete and U. Rohm, "The Cost of Serializability on Platforms that use Snapshot Isolation", In ICDE'08; Proc. of the 24th International Conference on Data Engineering (2008).

[11] Adya, Weak Consistency: a Generalized Theory and Optimistic Implementation for Distributed Transactions. Ph. D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, March (1999).

[12] P.A. Bernstein, V. Hadzilacos and N. Goodman, Concurrency Control and Recovery in Database Systems. Reading, MA, Addison Wesley (1987).

[13] M.J. Carey, "Improving the Performance of an Optimistic Concurrency Control Algorithm through Timestamps and Versions", IEEE Transactions on Software Engineering, Vol. SE-13 (6) (1987).

[14] J. Gray, P. Helland, P. O'Neil and D. Shasha, "The dangers of replication and a solution", In Proc. of the 1996 SIGMOD, ACM Press (1996).

[15] Transaction Processing Council.http://www.tpc.org/ (last access date: 2008-5-17).

Received May 25, 2009
Accepted June 10, 2009