A Novel Timestamp Ordering Approach for Co-existing Traditional and Cooperative Transaction Processing

Yanchun Zhang
Dept of Math & Comp.
University of Southern Qld
TWB, Q4350, Australia
yan@usq.edu.au

Yahiko Kambayashi
Dept of Information Sci.
Kyoto University
Kyoto 606-01, Japan
yahiko@kuis.kyoto-u.ac.jp

Yun Yang
School of Comp. & Math
Deakin University
Geelong, 3217, Australia
yun@deakin.edu.au

Chengzheng Sun
School of Comp. & IT
Griffith University
Brisbane, 4111, Australia
scz@cit.gu.edu.au

Abstract

In order to support both traditional short transaction and long cooperative transactions, we propose a novel timestamp ordering approach. With this timestamp ordering method, short transactions can be processed in the traditional way, as if there are no cooperative transactions, therefore not be blocked by long transactions; cooperative transactions will not be aborted when there is a conflict with short transactions, rather, it will incorporate the recent updates into its own processing; and the serialisabilities, among short transactions, and between a cooperative transaction (group) and other short transactions, are all preserved.

1. Introduction

In traditional database systems, user application programs are packaged as transactions and the concurrent data access is synchronised using concurrency control algorithms such as optimistic concurrency control (OCC), two-phase locking (2PL), and timestamp ordering (TO) [2, 3, 15]. The traditional database transaction model is suitable for conventional database applications, such as banking and airlines reservation systems. Transactions for these applications are generally simple, and are characterised as short duration that they will be finished within minutes or much shorter.

The traditional transaction concept has limited applicability in many of advanced applications such as cooperative work. In those environments, transactions tend to be very long, and need to cooperate with each other. For example, in cooperative environments, several designers might work on the same project. Each designer may start up a cooperative transaction. Those cooperative transactions joinly form a transaction group. Cooperative transactions in the same transaction group may read or update each others uncommitted object versions. Therefore, cooperative transactions may be interdependent.

The long transactions in traditional database systems may be aborted due to conflict operations or deadlocks. Aborting long transaction means the increased processing cost[25].

Cooperative transactions have been recently addressed in several research areas such as advanced database systems, groupware and CSCW, and workflow. [7] offers a comprehensive review and a collection of works on advanced transaction models from the database point of view. Saga[11], Cooperative Transaction Hierarchy[17], Cooperative SEE Transaction[13], DOM Transactions[4], Multilevel Transactions[14] and other transaction models [7] have been discussed, where all of them need some degrees of transaction cooperation and different correctness criteria from serialisability except Split Transaction model[19] (as Split Transaction model tries to reorganise a long transaction into independent and serialisable transactions). Other cooperative works have been conducted in the areas such as groupware and workflow systems, where the transaction cooperation is more evident [1, 8, 20, 21]. The major achievements in these areas are the models and some techniques to synchronise or coordinate the cooperative operations and to resolve the conflicts between cooperative transactions.

So far, it seems there is no work addressing the coexistence of both traditional transactions and cooperative transactions. In an envrironment with both traditional transactions and cooperative transactions, cooperative transactions should not be aborted due to conflict operations with short transactions. On the other hand, as the quick response is often required or preferred for short transactions, cooperative transactions should not block the short transactions.

In this paper, we will focus on the synchronisation and concurrency control between traditional transactions and cooperative transactions. As a long and cooperative transaction, it may take days even months to finish. It is rea-

sonable for this long transaction to see the results of other short transactions, no matter whether those short transactions start earlier or not. Therefore, we aim at developing a new timestamp approach to support both traditional transactions and cooperative transactions. Figure 1 shows a client-server system structure with both long and short transactions. We divide transactions into short transactions (T_s) and cooperative (or long) transactions, where long transactions are grouped to a cooperative group.

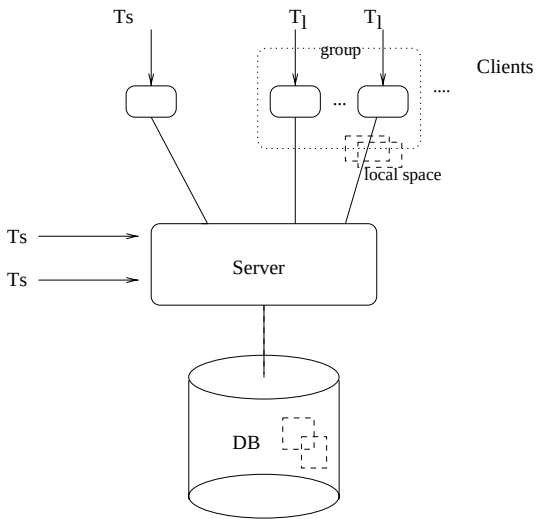

Figure 1. A system structure with both short transactions and cooperative transactions

The proposed method has the following features: (1) it allows both short (traditional) and cooperative transactions co-exist; (2) short transactions can be processed in the traditional way, as if there were no cooperative transactions, therefore not be blocked by long transactions; (3) cooperative transactions will not be aborted when there is a conflict with short transactions, rather it will incorporate the recent updates into its own processing; and (4) the f-conflict serialisability, a general correctness criteron for advanced transaction processing between a cooperative transaction (group) and other short transactions is preserved. Based on these features we believe this method is very suitable for advanced database applications to support both short transaction and cooperative work.

The paper is organised as follows. In section 2, we review some concepts and techniques on traditional transaction processing, including transactions, conflict serialisability and concurrency controls, especially the timestamp ordering approach. Section 3 develops a cooperative transaction model and a general correctness theory for cooperative transaction processing. Section 4 proposes a novel timestamp ordering method to deal with the interaction between traditional transactions and cooperative transactions. In section 5, we give some proofs on consistency and serialisability theorems. Conclusions and future work are included in section 6.

2. Traditional transaction processing

The database system refers to a database and the access facilities (database management system) to the database. One of the common objectives of database management systems, centralised or distributed, is the control and coordination of the execution of concurrent database transactions. The concepts of database transactions, concurrency control and serialisability are the common notions in the transaction management of centralised and distributed database management systems. The most popular concurrency control approaches developed for centralised database systems have been successfully adopted in the distributed environments.

2.1. Transactions

A transaction could be viewed as a transformation of a database from one consistent state to another consistent state[5]. Usually, a transaction can be issued interactively by users or be embedded in the application program written in a high-level language such as C, PASCAL or COBOL etc. A *transaction* is a particular execution of a program that manipulates the database by means of read and write operations. It can be defined as a sequence of database read and write operations, or more formally as a partial order of read and write operations. In addition, the transaction contains a *Commit* or *Abort* as its last operation, to indicate whether the execution it represents terminated successfully or not. In general, we use r_i[x] (or w_i[x]) to denote the execution of a read (or write) issued by transaction T_i on data item x. In a transaction a data item can be read and/or written at most once.

Traditionally, transactions are expected to satisfy the following four conditions, known as ACID properties [10], namely, Atomicity, Consistency, Isolation and Durability. The ACID properties can be trivially achieved by the serial execution of transactions. However, this is not a practical solution for the database management since it seriously impairs system performance. Usually, the database systems are operating in the multi-programming, multi-user environments, and the transactions are expected to be executed in the database system concurrently.

When several transactions execute concurrently, their operations are executed in an interleaved manner. Operations from one transaction may be executed between operations of other transactions. This interleaving can cause an inconsistent database. The database system must monitor and control the concurrent executions of transactions so

that overall correctness and database consistency are maintained.

A *schedule* indicates the interleaved order in which the operations of transactions were executed. If the operations of various transactions are not interleaved (i.e. the executions of transactions are ordered one after another) in a schedule, the schedule is *serial*. The serial execution of a set of transactions preserves the consistency of the database. As serial execution does not support concurrency, the equivalent schedules has been developed and applied for comparisons of a schedule with a serial schedule, such as view equivalence and conflict equivalence of schedules. In general, two schedules are *equivalent* if they have the same set of operations producing the same effects in the database.

Definition 1 *Two schedules Sc_1, Sc_2 are* view equivalent *if:*

1. *for any transactions T_i, the data items read by T_i in both schedules are the same;*
2. *for each data item x, the latest value of x is written by the same transaction in both schedules Sc_1 and Sc_2.*

Definition 2 *Two operations are said to be in* conflict *if:*

1. *they come from different transactions;*
2. *they both operate on the same data item; and*
3. *at least one of them is a write operation.*

Definition 3 *Two schedules Sc_1, Sc_2 are* conflict equivalent *if:*

for any pair of transactions T_i, T_j in both schedules and any two conflicting operations $o_{ip} \in T_i$ and $o_{jq} \in T_j$, such that if the execution order o_{ip} precedes o_{jq} in a schedule, say, Sc_1, the same execution order must exist in Sc_2.

Definition 4 *A schedule is* conflict serialisable *if it is conflict equivalent to a serial schedule. A schedule is view serialisable if it is view equivalent to a serial schedule.*

A conflict serialisable schedule is also view serialisable but not vice versa, since definition of view serialisability accepts a schedule which may not necessarily be conflict serialisable. There is no efficient mechanism to test schedules for view serialisability. It was proven that checking for view serialisability is NP-complete problem[18]. The conflict serialisability can be verified through a conflict graph.

The conflict graph among transactions is constructed as following: for each transaction T_i, there is a node in the graph (we also name the node T_i). For any pair of conflicting operations (o_i, o_j), where o_i from T_i and o_j from T_j, respectively, and o_i comes earlier than o_j, then add an arc from T_j to T_i in the conflict graph.

Theorem 1 *A schedule is conflict serialisable if and only if its conflict graph is acyclic [3].*

The serialisation order of a set of transactions can be determined by their conflicting operations in a serialisable schedule.

In order to produce conflict serialisable schedules, many concurrency control algorithms have been developed such as *Two Phase Locking (2PL), Timestamp Ordering (TO), Optimistic Concurrency (OC) Control*, etc.. As our work is closely related to Timestamp Ordering algorithm, we will review TO in the next subsection.

2.2. Timestamp Ordering (TO)

The timestamp ordering is a technique whereby the serialisation order is determined by the unique timestamp assigned to each transaction before it starts. Each transaction is assigned a timestamp by its transaction manager (TM). The TM attaches such a timestamp to all operations issued by the transaction. Furthermore, for each accessed data item A, there is a read timestamp t_r(A) to record the timestamp of the latest read, and a write timestamp t_w(A) to record the latest write. Any write operation of a timestamp earlier than the item's read timestamp is rejected. Likewise, any read or write operation earlier than the item's write timestamp is rejected. Rejecting an operation of a transaction leads to transaction abort and restart. Restarted transactions adopt new and later timestamps.

The basic TO algorithm has been proved to be a correct method which can produce serialisable schedules[2]. The rules of basic TO can be summerised as follows.
TO Rules: (Basic Timestamp Ordering Concurrency Control).

1. For each data item A, there are two timestamps t_r and t_w, where t_r is the largest timestamp of those transactions which have read A and t_w is the largest timestamp of those transactions which have written to A.

   Each transaction T has also a timestamp *t* to record its starting time.

2. *Read operation:* When a transaction T requests to read A, observe the following rules:
   (a) if $t(T) > t_w(A)$, T reads A, and sets max(t(T), $t_r(A)$) as the new value of $t_r(A)$.
   (b) if $t(T) < t_w(A)$, T cannot read A, and has to be rolled back.

3. *Write operation:* When T requests to write A, observe the following three rules:
   (a) if $t(T) > max(t_r(A), t_w(A))$, the request can be granted and the new value of $t(A)$ is set to $t(T)$.

(b) if $t_w(A) > t(T) > t_r(A)$, the write operation needs not be performed.

(c) if $t(T) < t_r(A)$, T cannot write A and it has to be rolled back.

The timestamp ordering is free from deadlock and starvation since there is no waiting in this algorithm. In fact waiting is meaningless in this approach as the rejection of an operation only depends on the greatest TO value of the item. When an access request of a transaction is rejected, the transaction is aborted. Other advantage of the basic timestamp ordering method over 2PL is that it provides a great degree of concurrency by producing some more flexibility schedules.

3. Cooperative transaction processing

In traditional database applications such as banking and airline reservation systems, transactions are short and non-cooperative, and usually can be finished in minutes or much shorter. The serialisability is a well accepted correctness criterion for these applications. The cooperative transaction processing in different applications may have different requirements and require different system supports to coordinate the work of multiple users and to maintain the consistency. So far there is no single common accepted correctness criteria and concurrency control approaches. In this section we will give a definition of advanced or cooperative transactions, and develop a general correctness criterion.

Regarding to cooperative transactions, one can also find other terms in literature such as non-traditional transaction, long transactions, advanced transactions and interactive transactions etc.. The advanced transaction processing or cooperative transactions processing have been investigated in groupware[9, 22, 23], workflow systems[12, 20] and advanced database transaction models[1, 7]. From these areas, one can derive the following common properties and requirements:

1. The cooperative work involves multiple concurrent users/transactions.
2. Exchanging and sharing of persistent data between different users/transactions need to be supported.
3. It needs to ensure the consistence between individual and group.
4. The users/transactions are often distributed.
5. Activities are often performed interactively by human and usually of long duration.

Based on these, we give the following definition:

Definition 5 *An advanced transaction (cooperative transaction group) is defined as a set (group) of cooperative transactions $T_1, T_2, ..., T_n$, with the following properties:*

1. *each cooperative transaction T_i is a sequence (or partial order) of read(x) and write(y) operations;*
2. *for the same data item, there might be more than one read(x), written as $read^1(x), read^2(x), \cdots$, in a cooperative transaction, each read(x) will get a different value depending on the time and interaction with other transactions; and*
3. *similarly, for each y, there might be more than one write(y), written as $write^1(y), write^2(y), \cdots$.*

The first part shows that an advanced transaction is a cooperative transaction group. If the size of group is one, it will become a single transaction. Property 1 is the same as that in the traditional transaction. Properties 2 and 3 indicate some cooperative features. The first read(x) may read other transaction's committed or uncommitted data depending on the concurrency control employed. After the first read operation on x, the data item might be updated by another traditional transaction or another cooperative transaction, then it can read the new value in the next read(x). Similarly, after the first write operation on x, due to the cooperative feature, a transaction may read some new data from other transactions, then issue another write(x) to incorporate this to the current processing. The later write(x) can undo the previous write, or do further update to show the new semantics.

Similar to distributed database transactions, the advanced transaction definition could be extended to a distributed advanced transaction as following:

Definition 6 *A distributed advanced transaction (distributed cooperative transaction group) is defined as a set (group) of cooperative transactions $T_1, T_2, ..., T_n$, with the following properties:*

1. *each transaction T_i consists of a set of subtransactions T_i^j at site j, j ∈ [1..m], m is the number of sites in a distributed system. Some T_i^j might be empty if T_i has no subtransaction at site j;*
2. *each subtransaction is a sequence (or partial order) of read(x) and write(y) operations;*
3. *for the same data item x, there might be more than one read(x), written as $read^1(x), read^2(x), \cdots$, in a cooperative transaction, each read(x) will get a different value depending on the time and interaction with other transactions; and*
4. *similarly, for each y, there might be more than one write(y), written as $write^1(y), write^2(y), \cdots$.*

Similar to that in traditional transactions, we assume that for write operations on x, there must be a read operation before the the first write in a cooperative transaction. It is natural to read the data first before the update, ie. one's update may depend on the read value or one may use read operation to copy the data into the local memory, then update the data and write it back (when commits).

In advanced transaction applications, cooperative transactions could read and write a data item more than once, which is different from traditional transactions. The reason to read a data item more than once is to know the recent result and therefore make the current transaction more accurate. However, this will bring the problem on the serialisability among cooperative transactions, as a cooperative transaction may read a data item before another transaction starts and also read the data updated by the same transaction. If so, the schedule between these two transactions will not be serialisable. However, from the semantic point of view, the most important read or write on the same data item will be the last read or write. If we give high priority on the last read or write conflicts in developing the correctness criteria, we could have an f-conflict (final conflict) graph and therefore propose an f-conflict serialisability theorem.

Definition 7 *The f-conflict graph among transactions is constructed as following: for each transaction T_i, there is a node in the graph (we also name the node T_i). For any pair of final conflicting operations (o_i, o_j), where o_i from T_i and o_j from T_j, respectively, and o_i comes earlier than o_j, add an arc from T_j to T_i in the conflict graph.*

Definition 8 *A schedule is f-conflict serialisable if and only if its f-conflict graph is acyclic.*

The f-conflict serialisation order of a set of transactions can be determined by their f-conflicting operations in a f-conflict serialisable schedule. From the definition, one can see the relationship between conflict serialisability and f-conflict serialisability:

Theorem 2 *If a schedule is conflict serialisable, it is also f-conflict serialisable, the vise versa is not true.*

The conflict serialisability is a special case of f-conflict serialisability in traditional transaction processing.

Definition 9 *A schedule of distributed advanced transactions is f-conflict serialisable if and only if the following two properties are satisfied:*

1. *the schedule of subtransactions at each site is f-conflict serialisable; and*

2. *the f-conflict serialisation order at all sites are the same.*

Advanced transaction or cooperative transaction processing in different applications may have different application dependent requirements and require different system supports to coordinate the work of multiple users and to maintain the consistency. As a result, different synchronisation, coordination and control mechanisms within a cooperative transaction group are developed. The f-conflict serialisability in conjunction with application dependent semantics could be used for designing and testing advanced transactions processing approaches. The application dependent requirements can be reflected in the detailed transaction structures. For example, when there are several write operations on the same x, the later writes might be an undo and then a redo of the operation (or perform a different operation). The undo operations might be reverse operations or compensating operations, and the redo operations could be a contingency operations or new operations which may need to keep the intention (user intention) of the original write [21, 22], or to incorporate the new semantics.

4. A novel timestamp ordering approach

Though the cooperative transaction processing has been addressed in several research areas, the interactivity and long duration of cooperative transactions have not been explored thoroughly, especially their interactions with traditional transactions when both traditional transactions and cooperative transactions co-exist. In this case, both types of transactions co-exist, traditional (short) transactions should not be blocked due to conflict with cooperative (long) transactions, and the cooperative (long) transactions should not be aborted due to the conflict with traditional transactions. To explore the interactive feature, cooperative transactions should be able to read the recent data.

For example, suppose that, a cooperative transaction starts today, and will take a few days or maybe months to finish, and a short conflicting transaction will only need a few minutes or much shorter, but due to late arrival of a write operation or conflict with a long transaction, it has been aborted, and started again and succeeded with a later timestamp. In such a case, the long transaction would not be able to read this updated value when it finishes one month later. Just imaging that a design or a result from a long transaction is based on some old database value, which has been updated several times already. How would one rank the result of this long transaction, updated or obsolete? So we argue that the result of such short transactions should be revealed to long cooperative transactions. Due to the interactivity of cooperative transactions, users should be able to incorporate the new result into the transaction by issuing some new operations or compensating some earlier operations.

In this section, we will discuss the relationship between

short transactions and cooperative transactions and develop a concurrency control approach to support the coordination between them. We consider two kinds of transactions, traditional short transactions T and long cooperative transactions T_c. To simplify the discussion we treat a cooperative transaction group as one logical transaction to avoid the discussions on operation synchronisation within a cooperative transaction group.

In a cooperative transaction, it is reasonable to assume that for each write operation there must be a read operation earlier. It is natural to read the data first before the update, ie. one's update can depend on the read value or use the read operation to copy the data in the local memory, then update the data and write it back (when commits).

Similar to the optimistic concurrency control approach, we use a private work space for the cooperative transaction group where all data items read or updated are stored. All read operations are performed by copying the data items into this work space. Write operations are performed on the private work space. These values will be reflected to the database after the transaction is committed.

Due to the interactive feature, a cooperative transaction can be formed with great flexibility as an user can dynamically issue an operation depending on the most current information. If a datum has been updated recently after the first read, the cooperative transaction may wish to read the data again due to the cooperative feature. Therefore we *assume* that in a cooperative transaction, it could read and write the same data item more than once. It can issue another read operation if knowing the datum has been changed recently. In order to incorporate the recent changes into its own transaction, it can perform additional operations or compensate the previous operations. That is the flexibility of interactive work.

4.1. Transaction interactions

We argue that the cooperative transactions and traditional transaction would not be aborted when there are conflicts between cooperative operations and traditional operations. When traditional transactions commit, the cooperative transaction may need to read or re-read the new value from them in order to continue the work based on the recent results. In such a case, some partial rollback is required. According to the order of the conflict operations with respect to the commit time, and the order of the timestamps, we consider the following six cases:

- Case (a).

  As shown in Figure 2, $t(T_c) < t(T)$, $r_c(A)$ conflicts with w(A), and $r_c(A)$ comes after T is committed. In this case we will let T_c continue and change the timestamp t(T_c) to a value t(T)+ δ (small value, just to indicate it is bigger than t(T)).

  According to the traditional timestamp ordering method, T_c would be aborted.

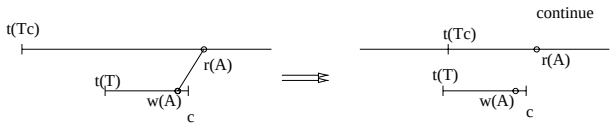

Figure 2. Case (a)

- Case (b).

  As shown in Figure 3, $t(T_c) < t(T)$, $r_c(A)$ conflicts with w(A), and $w(A)$ comes later. In order to let the cooperative transaction know this update, a message needs to be sent to T_c about the update when t commits. T_c may need to read this new value. In such a case, some partial rollback may be needed to incorporate this new result. If so, change the timestamp t(T_c) to a value t(T)+ δ. After this, T_c's virtual timestamp will be bigger than T's timestamp. According to the traditional timestamp ordering method, both T and T_c will continue as usual, then T_c will never know this update no matter how long it is, which does not reflect the advantage of cooperative and interactive feature.

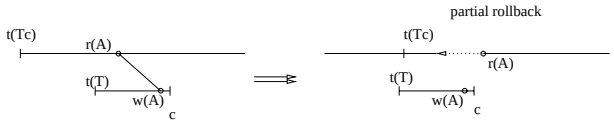

Figure 3. Case (b)

- Case (c).

  As shown in Figure 4, $t(T_c) > t(T)$, $r_c(A)$ conflicts with w(A), and $w(A)$ comes later. We let T continue, and send a message about the update to T_c when T commits. T_c may need to read this value after T commits. In such a case, some partial rollback may be needed to incorporate this new result. No change on timestamp t(T_c) in this case.

  Here the partial rollback and redo are not the same as that in the database systems recovery, where rollback refers to undo all the operation to a point and then redo. Depends on the applications, we may only need to re-read the new value and then perform some additional operations to incorporate the changes, or undo some affected operations and issue some new ones.

  According to the traditional timestamp ordering method, T will be aborted and restarted with a later timestamp. When it is successfully committed, there is

still a conflict between w(A) and $r_c(A)$. This will become the case in Figure 3. Following the same discussion, in order to incorporate the recent changes in the cooperative transactions, one needs to re-read A and perform some additional operations. As this re-read is later than the re-read case in Figure 4, it is reasonable to say the cost is much higher in case b than that in case a. This arguement can be supported by some models on human's memory, reference and correlation ability. For two related concepts, when reading with two different intervals t1 and t2, $t1 < t2$, the human's correlation degree is higher in the first case of t1 interval. Therefore, the comprehension time is shorter. In addition, as the interval increases, one might have done more work which needs to be modified. Therefore, to let T continue rather than abort is a reasonable solution.

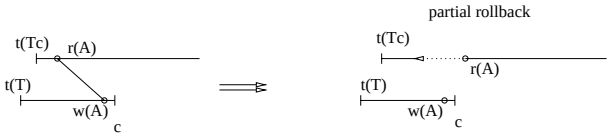

Figure 4. Case (c)

- Case (d).

  As shown in Figure 5, $t(T_c) > t(T)$, $r_c(A)$ conflicts with w(A), and $r(A)$ comes later. T_c will continue, and this read will read the current value of A, the same as that in TO.

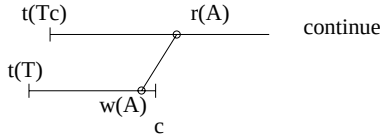

Figure 5. Case (d)

- Case (e).

  As shown in Figure 6, $t(T_c) < t(T)$, $r_c(A)$ conflicts with w(A), and r(A) comes before T commits. This is similar to case b, but the conflict is found when r_c(A) arrives. Let the cooperative transaction continue (read A), record the conflict information and request the system to send the commit or abort information when T commits. After t commits, change the timestamp $t_c(T)$ to t(T)+ δ. If T aborts, T_c will need partial rollback to this read, and re-read.

  According to the traditional timestamp ordering method, T_c will be aborted.

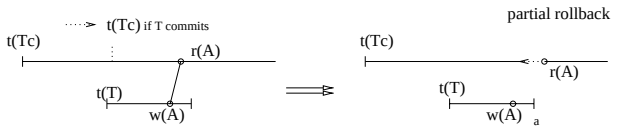

Figure 6. Case (e)

- Case (f).

  As shown in Figure 7, $t(T_c) > t(T)$, $r_c(A)$ conflicts with w(A), and $r_c(A)$ comes later, Similar to case (e), let the cooperative transaction continue (read A), record the conflict information, and request the system to send the commit or abort information when T commits. After t commits, change the timestamp $t_c(T)$ to t(T)+ δ. If T aborts, T_c will need partial rollback to this read, and re-read, and request the system to send the new result to T_c after T commits.

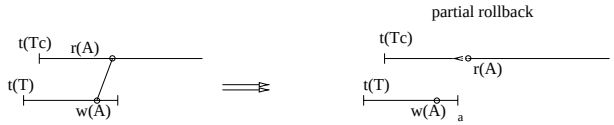

Figure 7. Case (f)

The above six cases covered all possible conflicts between traditional transactions and cooperative transactions. Based on these we develop a novel timestamp ordering approach for dealing with their interactions.

4.2. The new TO algorithm

For each data item A, we will use four timestamps t_r, t_r^c and t_w, t_w^c, where t_r is the largest timestamp of those transactions which have read A , t_r^c is a virtual timestamp of the cooperative transactions which have read A, initialised as the starting time of the cooperative transaction group, t_w is the largest timestamp of those transactions which have written to A, and t_w^c is the virtual timestamp of the cooperative transactions which have written to A.

For the conflicts between the traditional transactions, we will use the basic TO. For the conflicts between traditional and cooperative transactions, we will give some new rules based on the discussion on the above six cases.

New TO Rules: (New Timestamp Ordering Concurrency Control).

1. *Read operation:* When a transaction T requests to read A, observe the following rules:

   (a) if $t(T) > t_w(A)$, T reads A, and sets max(t(T), $t_r(A)$) as the new value of $t_r(A)$.

(b) if $t(T) < t_w(A)$, T cannot read A, and has to be rolled back.

2. *Write operation:* When T requests to write A, observe the following three rules:

   (a) if $t(T) > max(t_r(A), t_w(A))$, the request can be granted and the new value of $t(A)$ is set to $t(T)$.

      i. If $t(T_c) < t(T)$, $r_c(A)$ conflicts with w(A), and $w(A)$ comes later, record the message about the conflict and the write (need to send to T_c when T commits).

      ii. If $t(T_c) > t(T)$, $r_c(A)$ conflicts with w(A), and $w(A)$ comes later. record the message about the conflict and the write.

   (b) if $t_w(A) > t(T) > t_r(A)$, the write operation needs not be performed.

   (c) if $t(T) < t_r(A)$, T cannot write A and it has to be rolled back.

3. *T_c's Read operation:*

   (a) If $t(T_c) < t(T)$, $r_c(A)$ conflicts with w(A), and $r_c(A)$ comes after T is committed. T_c continues and changes the timestamp t(T_c) to a value max(t(T_c, t(T)+ δ).

   (b) If $t(T_c) < t(T)$, $r_c(A)$ conflicts with w(A), and $r_c(A)$ comes later, but before T commits, record the conflict information, ask the system to send the commit or abort message when T commits or aborts. Then T_c continues (may need rollback to re-read).

   (c) If $t(T_c) > t(T)$, $r_c(A)$ conflicts with w(A), and $r(A)$ comes after t commits, T_c will continue.

   (d) If $t(T_c) > t(T)$, $r_c(A)$ conflicts with w(A), and $r_c(A)$ comes later, but before t commits, record the conflict information, ask the system to send the commit or abort message when T commits or aborts. Then T_c continue (may be rollback late to re-read).

4. *T_c's Write operation:* When requests a write, granted, continues the operation.

5. *Commit or abort operation:* this information and/or conflict messages will be sent to T_c.

6. Messages from the write transaction: When T_c receives messages, check

   (a) if T is committed,

      i. If $t(T_c) < t(T)$, $r_c(A)$ conflicts with w(A), and $w(A)$ comes later. T_c may need to read the value again, then some partial rollback to incorporate this new result. If so, change the timestamp t(T_c) to value t(T)+ δ.

      ii. If $t(T_c) > t(T)$, $r_c(A)$ conflicts with w(A), and $w(A)$ comes late, some partial rollback may be needed to incorporate this new result. No change on timestamp t(T_c).

      iii. $r_c(A)$ conflicts with w(A), and $w(A)$ comes earlier. If so, change the timestamp t(T_c) to max(T_c, t(T)+ δ).

   (b) if T is aborted, $r_c(A)$ conflicts with w(A), and $w(A)$ comes earlier, T_c needs some partial rollback to read the old value. No change on the timestamp t(T_c).

As we assume that in cooperative transactions for each write operation there must be a read operation before hand, we do not need to consider the conflicts between the traditional write operations and cooperative write operations, since the earlier conflict treatment between cooperative read operations and traditional write operations will cover the treatment on their write - write conflicts.

Based on the interactive features of some cooperative work, the rollback techniques will be different from the rollback in the traditional database recovery. In traditional database, rollback is used to recover transactions from failure[16]. In database recovery, a write-ahead log is used to keep the records of recent updates. When recovery from failure, a transaction may need rollback to a point or to the beginning of the transaction, that means undoing all the operations from a transaction to that point. Consequently these operations will be redone. In database, these undo and redo can be done automatically in the system, as all the transaction semantics are fixed before submission. In cooperative work environments, many transactions are on-line and dynamic. When roll back to a point, one may need to undo only some affected operations to that point, then perform some new operations.

5. Consistency and serialisability

In the proposed new TO algorithm, both traditional (short) transactions and long cooperative transactions coexist. In addition, it has the following properties:

Theorem 3 *Short transactions can be processed in the traditional way as that in the basic TO algorithm, as if there were no cooperative transactions, therefore they will not be blocked by cooperative transactions. The serialisability among these transactions is preserved.*

Proof: This can be shown from Rules 1 and 2, the process of read and write operations from traditional transactions is the same as they are in basic TO. The only extra booking work is to record the conflicting messages with cooperative transactions as shown in Rule 2(a) i and ii, and to send messages to T_c when commit as shown in Rule 4. So they will not be blocked by cooperative transactions. As basic TO guarantees the serialisability of the transactions, the new TO will also preserve the serialisability among traditional transactions.

Theorem 4 *In new TO, cooperative transactions will not be aborted when there is a conflict with short transactions, rather, it will incorporate the recent updates into its own processing;*

Proof: Again this can be shown directly from Rules 3, 4 and 6.

For cooperative work, the correctness criteria will depend on the application and cooperative transaction processing models employed. In the REDUCE project[22], the correctness criteria are formalised through a CCI consistency model and these criteria are achieved through a set of synchronisation and transformation algorithms. As shown in [24], the schedule produced from REDUCE system is f-conflict serialisable. Furthermore, the following general correctness of the novel TO algorithm can be shown.

Theorem 5 *In the new TO algorithm, a schedule of a cooperative transaction T_c (group) and other short transactions committed earlier than T_c commits, produced from new TO, is f-conflict serialisable.*

Proof: To prove this theorem, we need to consider the f-conflict graph of the schedule, and then show that the f-conflict graph is acyclic, since that a schedule is f-conflict serialisable if and only if its f-conflict graph is acyclic. We now construct the conflict graph in two steps:

Step 1.

The conflict graph among traditional transactions is constructed as following: for each transaction T_i, there is a node in the graph (we also name the node T_i). For any pair of conflicting operations (o_i, o_j), where o_i from T_i and o_j from T_j, respectively, and o_i comes earlier than o_j, add an arc from T_j to T_i in the conflict graph.

As the schedule among the traditional transactions is serialisable, there will be no cycle among them, and we could assume their serialisation order as $T_1, T_2, ..., T_i, ..., T_n$ (which can be done by rearrange transactions' subscripts).

Step 2.

Now add a node T_c for cooperative transaction group. Let us see how to add and maintain the arcs between T_c and other transactions.

We only need to consider conflicting cases a, b, c and d shown in Figures 2 - 5, as T in cases 6 and 7 are aborted and the arcs from and to T will be deleted. For the conflicts in cases a and d, there will be only arcs from T to T_c, and $t(T_c)$ is changed to a value bigger than $t(T)$, it indicates that T_c can be be serialised after T.

For the conflicts in cases b and c, there might be some arcs from T_c to T initially as some operations from T_c come later than the conflicting operations from T.

But when T commits, the conflicting messages will be sent to T_c. According to Rule 6, T_c will re-read the value written by T, and redo the necessary operations to incorporate this update. After this T_c is different from the initial T_c, as the new T_c has read the value from T, it is now serialisable after T, that's the reason to change T_c's timestamp. In the final conflict graph, the arcs from T_c to T now need to be reversed. Hence, there will be new arcs from T_c to any T.

Therefore, there will be no cycle, and T_c is serialisable after all the short transactions which are committed earlier than T_c. □

The f-conflict serialisability is a more generalised form of the traditional conflict serialisability. The theorem can be used for advanced transactions such as long transactions, cooperative transactions and interactive transactions when there are more than one read and write on the same data in the transaction.

6. Conclusions and future work

In traditional database systems, a transaction can be aborted due to the conflict operations. Aborting a long transaction often means the increased cost. In order to support both traditional short transaction and long cooperative transactions, we have proposed a novel timestamp ordering approach in this paper. With our new timestamp ordering (TO) method, short transactions can be processed in the traditional way, as if there are no cooperative transactions, therefore not be blocked by long transactions; cooperative transactions will not be aborted when there is a conflict with short transactions, rather, it will incorporate the recent updates into its own processing; and the serialisabilities, among short transactions, and between a cooperative transaction (group) and other short transactions, are all preserved.

In developing the new TO method, we emphasise the interactive features of many cooperative transactions, and take the advantages from user interaction, which makes our method different from other concurrency control approaches.

References

[1] K. Aberer et al., Transaction Models Supporting Cooperative Work – TransCoop Experiences. in Y Kambayash and K. Yokota, editors, Cooperative databases and applications, World Scientific, 1997.

[2] P. A. Bernstein and N. Goodman. Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems. in Proc. 6th Int. Conf. on VLDB, 1980.

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

[4] A. Buchmann, M.T. Ozsu and M. Hornick. A transaction model for active distributed object systems. *in A. Elmagarmid, editor, Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.

[5] J. Cao. Transaction Management in Multidatabase Systems. PhD thesis, Department of Mathematics and Computing, University of Southern Queensland, 1997.

[6] U. Dayal, M. Hsu and R. Latin. A transactional model for long running activities. In *Proc. 17th VLDB*, pp.113-122, 1991.

[7] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[8] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proc. ACM SIGMOD*, pages 399–407, 1989.

[9] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: some issues and experiences. *Communication of the ACM*, Vol.34, No.1, pp.39–58, Jan. 1991.

[10] J. N. Gray. The transactions concept: Virtues and limitations. In *Proc. of 7th Int. Conf.on VLDB*, pp.144–154, Sept. 1981.

[11] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD*, 1987.

[12] D. Jean, A. Cichock and M. Rusinkiewicz. A database environment for workflow specification and execution. Proc. Intl Symposium on Cooperative Database Systems Kyoto, Dec. 1996.

[13] G. Heiler, et al., A flexible framework for transaction management in engineering environments. *in A. Elmagarmid, editor, Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.

[14] G. Weikum and H.J. Shek. Concepts and applications of multilevel transactions and open nested transactions. *in A. Elmagarmid, editor, Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.

[15] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. In *Proc. Int. Conf. on VLDB*, 1979.

[16] C. Mohan, et al., ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, ACM TODS, Vol.17, No.1, March, 1992.

[17] M. Nodine and S. Zdonik, Cooperative transaction hierarchies: a transaction model to support design applications, in Proc. 10th Int. Conf. on VLDB, pp83-94, 1984.

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

[19] C. Pu, G. Kaiser and N. Huchinson. Split transactions for open-ended activities. *Proc. 14th Int. Conf. on VLDB*, Los Angeles, August 1988.

[20] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. *in W Kim, editor, Modern Database Systems*, Addition-Wesley, 1994.

[21] C. Sun, X. Jia, Y. Zhang, and Y. Yang: A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems. In Proc. of ACM Group97, pp.425-434, Phoenix, USA, Nov. 1997.

[22] C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen, Achieving convergency, causality-preservation and intention preservation in real-time cooperative editing systems. ACM Transactions on Computer-Human Interaction, Vol. 5, No. 1, pp.63-108, March, 1998.

[23] Y. Zhang and Y. Yang, On operation synchronisation in cooperative editing environments. In IFIP Trans. A-54 on Business Process Re-engineering, pp.635-644, 1994.

[24] Y. Zhang, Y. Kambayashi, C. Sun, Y. Yang and X. Jia, F-conflict serializability: A correctness criterion for advanced transaction processing, Technical Report SC-MC-9803, Dept. of Math and Computing, University of Southern Queensland, 1998.

[25] X. Zhong and Y. Kambayash. Timestamp Ordering Concurrency Mechanisms for Transactions of Various length. *Proceedings of 3rd International Conference on Foundations of Data Organisation and Algorithms (FODO'89)*, Paris, 1989. Lecture Notes in Computer Science 367, Springer-verlag, 1989.