Commitment Ordering Based Distributed Concurrency Control for Bridging Single and Multi Version Resources

Yoav Raz

Digital Equipment Corporation, 151 Taylor St. (TAY1), Littleton, Ma 01460

ABSTRACT

Multi version (MV) based database systems allow queries (read-only transactions) to be executed without blocking updaters (read-write transactions), or being blocked by them. Such systems become more and more common. In a multi database environment, transactions may span several single version (SV) based database systems, as well as MV based ones. The database systems may implement various concurrency control techniques. It is required that a globally correct concurrency control is guaranteed.

Commitment Ordering (CO) is a serializability concept, that allows global serializability to be effectively achieved across multiple autonomous Resource Managers (RMs; e.g., database systems). The RMs may use different (any) concurrency control mechanisms. Thus, CO provides a solution for the long standing global serializability problem ([Raz 90]). For MV resources, the correctness criterion that corresponds to serializability is one-copy serializability (1SER). CO also allows to effectively achieve global 1SER. This work provides a theoretical framework to deal uniformly with both SV and MV resources. Generic distributed CO algorithms, which guarantee the 1SER property over multiple RMs with mixed SV and MV resources are presented. It is also shown, that CO is a necessary condition for guaranteeing 1SER over multiple autonomous RMs with mixed resources.

1 Introduction

Multi version (MV) based database systems allow queries (read-only transactions) to be executed without blocking, or being blocked by updaters (read-write transactions). Such systems become more and more common (e.g., [Ragh 91], [Rdb/VMS], [Oracle], [InterBase]). Various mechanisms for implementing this idea have been proposed (e.g., [Agra 89], [Agra 92], [Bobe 92a], [Bobe 92b], [Chan 85], [Garc 82], [Moha 92], [Weih 87]). These mechanisms, which are inherently based on transient multi versioning (i.e., several last versions of a data item/object are maintained), implement special data access strategies for queries, that do not interact, or loosely interact with the access strategies for updaters, but provide an overall correct concurrency control. A vast literature on concurrency control over multiple single version (SV) based database systems exists (e.g., [Brei 91], [Elma 87], [Geor 91], [Litw 89], [Pu 88], [Raz 90] and [Veij 92]). However, in a multi database environment transactions may span several SV based database systems, as well as MV based systems. The database systems may implement various concurrency control techniques. It is required that a globally correct concurrency control is guaranteed in such environments. This work provides generalizations for multi mixed MV and SV based databases. The work does not concentrate on any specific MV mechanism, but rather shows how to integrate various (any) such techniques, including SV based, in heterogeneous environments of multi databases.

Commitment Ordering (CO; [Raz 90], [Raz 91a], [Raz 91c], [Raz 92]) is a serializability concept¹, that allows global serializability to be effectively achieved across multiple autonomous Resource Managers (RMs; e.g., database systems, recoverable objects, etc.) that may use different (any) concurrency control mechanisms. Thus, CO provides a solution for the long standing global serializability problem (e.g., [Shet 90], [Silb 91]). CO is based on ordering transactions’ commit decision events by the order of respective conflicting resource access operations. In this work we use the notion of transactions in conflicts defined and generalized for MV resources in [Raz 90]. The generalized definition allows to view SV resources as a special case of MV ones, and deal with both within a common theoretical framework. For MV resources, the correctness criterion, that corresponds to serializability (SER), is one-copy serializability (1SER; see for example, [Bern 87]). In this work we show that CO also allows to effectively achieve global 1SER over mixed SV and (non-SV) MV resources, or mixed SV and MV RMs in a multi RM (distributed) environment. Generic distributed CO algorithms, which guarantee the 1SER property over multiple RMs with mixed resources (some RMs may be either SV or MV based), are presented. [Raz 90] ([Raz 92]) shows that CO is a necessary condition for guaranteeing global serializability over multiple autonomous SV based RMs. Similarly, CO is also a necessary condition for guaranteeing 1SER over multiple autonomous RMs with mixed resources.

We define a RM to be (concurrency control) autonomous², if it does not share any resources and concurrency control information (e.g., timestamps) with another entity (external to the RM), and is being coordinated.

¹[Brei 91] redefines CO, naming it strong recoverability (this property was introduced there in the last revision of the paper, April 18, 1991), and uses CO to show (the previously known result; e.g., [Pu 88]) that applying rigorousness (named S-S2PL in [Raz 90]) locally implies global serializability. No algorithm for enforcing CO (beyond S-S2PL) is given there.

²For other definitions/types of autonomy see for example [Garc 88].
(at the nonapplication level) with other RMs solely via Atomic Commitment (AC) protocols ([Raz 90]). This notion of autonomy is useful, since a RM is typically unaware of any resource state dependability with states of resources external to the RM (cross RM integrity constraints), implied by applications. This is also true in the cases where RMs are coordinated by multi-database systems, which provide applications with integrated views of resources. AC protocols are needed to achieve global atomicity. Atomicity means that either a distributed transaction is committed, i.e., its effects on all the resources involved become permanent, or it is aborted (rolled back), i.e., its effects on all the resources are undone. Since atomicity is an inherent property of a transaction (we do not deal here with transaction models that compromise atomicity), AC protocols are essential for multi RM transactions. Thus, by our definition of autonomy, guaranteeing global serializability across autonomous RMs means achieving this goal by using the minimum RM coordination (communication) possible. The most commonly used atomic commitment protocols are variants of the Two Phase Commitment protocol (2PC - [Gray 78], [Lamp 76]).

This work provides generalizations for MV based resources of results in [Raz 90] ([Raz 92]). It is an extension of ideas presented in the multi version based CO scheduling section in [Raz 90]. Correctness is guaranteed by updaters complying with CO (i.e., the generated histories’ projections on updaters are in the class CO, and thus in 1SER) to guarantee consistent results in [Raz 90] ([Raz 92]). Two more event types such as locking and unlocking may be introduced when necessary. The term permanent is relative and depends on a resource’s volatility (e.g., sensitivity to process or media failure).

A transaction, informally, is an execution of a set of programs that access shared resources. It is required that a transaction is atomic, i.e., either the transaction completes successfully, and its effects on the resources become permanent, or all its effects on the resources are undone, i.e., related versions disappear or become inaccessible. In the first case, the transaction is committed. In the second, the transaction is aborted.

Formally, we use for the notion of a transaction an abstraction that captures only events and relationships among them, that are necessary for reasoning about concurrency control.

A single RM transaction $T_i$ is a partial order of events (specific events within the above informally defined transaction). The (binary, asymmetric, transitive, irreflexive) relation which comprises the partial order is denoted $\preceq$.

The binary, asymmetric, transitive, irreflexive relation which comprises the partial order is denoted $\preceq$.

Remarks:

- $\text{event}_a \preceq \text{event}_b$ reads: $\text{event}_a \preceq \text{event}_b$ (in $T_i$).
- The subscript $i$ may be omitted when the transaction’s identity is known from the context.

The events of interest are the following:

- The operation of reading a resource; $r[x_i]$ denotes that transaction $T_j$ has retrieved (read) the version (state) of the resource $x$ written by transaction $T_i$.
- The operation of writing a resource; $w_i[x_j]$ means that transaction $T_i$ has modified (written) the state of the resource $x$, generating a new version (state) $x_i$.
- $\text{Ending a transaction}$; $e_i$ means that $T_i$ has ended (has been either committed or aborted) and will not introduce any further operations.

A transaction obeys the following transaction rules (axioms):

- TRI

A transaction $T_i$ has exactly a single event $e_i$.

A value is assigned to $e_i$: $e_i = c$ if the transaction is committed; $e_i = a$ if the transaction is aborted.

Notation: $c_i$ may be denoted $c_i$ or $a_i$ when $e_i = c$ or $e_i = a$, respectively.

2 More event types such as locking and unlocking may be introduced when necessary.

2.1 MV transactions and histories

For a multi-version (MV) resource, a new resource version (state) is generated by each write operation of the resource, and the various versions persist, and can be read independently (e.g., see [Bern 87]). The versions of each resource are totally ordered by the order of their generation.

A multi version resource is single-version (SV), if only the last version that exists can be read (i.e., a SV resource is a special case of a MV one).

It is common that a RM supports either SV or non-SV, MV resources, but not both types. However, it is possible, in principle to have a RM supporting both types. Some RMs that support non-SV resources allow dual resource-access mode: either as SV, or non-SV resources. The first mode is used for updaters (i.e., transactions that modify resources), and the latter to support queries (read-only) transactions (i.e., transactions that do not modify resources; e.g. see snapshots in [Rdb/VMS]).

A transaction is an execution of a set of programs that access shared resources. It is required that a transaction is atomic, i.e., either the transaction completes successfully, and its effects on the resources become permanent, or all its effects on the resources are undone, i.e., related versions disappear or become inaccessible. In the first case, the transaction is committed. In the second, the transaction is aborted.
A transaction is prepared either to be committed or aborted; otherwise it is undecided. An undecided transaction is ready if it has completed its processing, and is prepared either to be committed or aborted; otherwise it is active.

The following diagram defines the possible transitions between states:

![Diagram of transaction states and their transitions]

The notion of conflicting operations plays a central role in serializability theory. Traditionally only write-read conflicts (e.g., [Bern 87]) are defined for MV resources. Here we deviate from the common definitions, and define conflicting operations as follows:

A transaction is decided, if it is either aborted or committed; otherwise, it is undecided.

A transaction is conflict-equivalent with another one-copy serializable (1-serial) transaction if their commit projections (restriction) of a partial order P over a set S (the set of committed transactions) intersect, i.e., if for every two transactions Ti, Tj in S there exists a transaction T such that Ti <H T <H Tj.

A complete (MV) history H over a set T of transactions is a partial order \( \preceq_H \) defined according to the following history rules (axioms):

**HIS1**

If \( T_i \) is in T and \( \text{event}_a \preceq_H \text{event}_b \), then \( \text{event}_a \preceq_H \text{event}_b \).

**HIS2**

If \( T_i \), \( T_j \), and \( T_k \) are in T then the following precedence relationships exist between conflicting operations:

- If \( r_i[x_i], i \neq j \), exists, then \( w_j[x_j] < r_i[x_i] \).
- If \( r_i[x_i], i \neq j \), and \( w_j[x_j], i \neq j \), exist, then either \( w_j[x_j] < r_i[x_i] \) or \( r_i[x_i] < w_j[x_j] \) is true.
- If \( w_i[x_i] \), \( w_j[x_j] \), and \( i \neq j \), exist, then either \( w_i[x_i] < w_j[x_j] \) or \( w_j[x_j] < w_i[x_i] \) is true.

For modeling executions with incomplete transactions, we define a history to be any prefix \( \sigma \) of a complete history.

### 2.2 Correct histories - 1SER and REC

Transaction conflicts ([Raz 90]) are redefined for the MV case. Like in the pure SV case, also here \( \text{ww} \) and \( \text{wr} \) transaction conflicts have a correlation with time-precedence between respective operations. However, for the MV case \( \text{rr} \) conflicts are determined solely by the version read, regardless of the time of reading, i.e., a \( \text{rr} \) conflict can occur even if the read operation takes place after the respective write operation. Conflicts between transactions are redefined as follows:

- If \( r_i[x_i], i \neq j \), exists, or if \( r_i[x_i] \) exists, and \( w_j[x_j] < w_i[x_i], i \neq j \), then \( T_i \) is in a \( \text{wr} \) (write-read) conflict with \( T_j \) (on \( x \)).
- If \( r_i[x_i] \) exists, and \( w_j[x_j] < w_i[x_i], i \neq j \), then \( T_i \) is in a \( \text{ww} \) (write-write) conflict with \( T_j \) (on \( x \)).
- If \( r_j[x_j], i \neq j \), exists, or if \( r_j[x_j] \) exists, and \( w_i[x_i] < w_j[x_j], i \neq j \), then \( T_j \) is in a \( \text{wr} \) (write-read) conflict with \( T_i \) (on \( x \)).

A conflict equivalence exists between two histories H and \( H' \) (the two are conflict equivalent) if they are both defined over the same set of transactions T, and consist of the same transaction events (for partially executed transactions), and T is in a (\( \text{rr} \) or \( \text{wr} \) or \( \text{ww} \)) conflict with \( H \), due to operations \( p_i[x_i], q_i[x_i] \), if and only if \( T_i \) is in a conflict with \( H' \) due to the same operations, for any conflicting operations \( p_i[x_i], q_i[x_i] \) of any committed transactions \( T_i, T_{ji} \), respectively, in T (i.e., H and \( H' \) have the same conflicts between operations of committed transactions).

A history H over a transaction set T is serial if for every two transactions \( T_i, T_j \) in T and all the operations and \( e_i \) the end of \( T_i \), precede all the operations and \( e_j \) the end of \( T_j \) (i.e., if \( p_i[x_i] < H e_i \) then for any operations \( s_j[u_m^1], t_j[v_n^1] \in H \), \( s_j[u_m^1] < H t_j[v_n^1] \), and \( e_i < H t_j[v_n^1] \)).

The commit projection of a history H, is its projection (restriction) on its set of committed transactions.

A serial history is one-copy serial (1-serial), if for all \( i, j, x \), if \( T_i \) reads \( x \) from \( T_j \), then \( r_i[x_i] \) is the last transaction preceding \( T_i \) that writes \( x \) ([Bern 87]).

A history is one-copy serializable (ISER), if its commit projection is conflict-equivalent with a 1-serial history.

---

1. A prefix of a partial order P over a set S is a partial order \( P' \) over a set \( S' \subseteq S \), with the following properties:
   - If \( b \in S' \) and \( a <_P b \) then also \( a \in S' \).
   - If \( a, b \in S' \) then \( a <_P b \) if and only if \( a <_P b \).

2. Let P be a partial order over a set S. A projection (restriction) of P on a set \( S' \subseteq S \) is a partial order \( P' \), a subset of P, that consists of all the elements in P, involving elements of \( S' \) only.
In our model Serializability (SER) is defined for SV histories, i.e., histories over transactions that access SV resources only. For SV histories serializability is defined here to be identical to one-copy serializability. This definition properly positions 1SER as a generalization of SER for mixed SV and non-SV resources, and as the correctness criterion for MV histories analogous to SER for SV histories.

The Serializability Graph of a history H, SG(H), is the following directed graph:

\[
SG(H) = (T, C) \quad \text{where}
\]

- \(T\) is the set of all unaborted (i.e., committed and undecided) transactions in H.
- \(C\) is a (subset of \(T \times T\)) set of edges that represent transaction conflicts: Let \(T_1, T_2\) be any two transactions in \(T\). There is an edge from \(T_1\) to \(T_2\) if \(T_2\) is in a conflict with \(T_1\).

The Committed Transaction Serializability Graph of a history H, CSG(H), is the subgraph of SG(H) with all the committed transactions as nodes and all the respective edges.

The Undecided Transaction Serializability Graph of a history H, USG(H), is the subgraph of SG(H) with all the undecided transactions as nodes and all the respective edges.

Theorem 2.1 provides a criterion for checking one-copy serializability.

**Theorem 2.1 - The One Copy Serializability Theorem**

A MV history H is one-copy serializable (1SER) if and only if CSG(H) is cycle-free.

Proof outline:

CSG(H) as defined here is similar to the graph MVSG(H,<<) defined in [Bern 87] (except for transaction conflicts involving unread versions). It is shown there that one-copy serializability of H is a necessary and sufficient condition for the acyclicity of MVSG(H,<<). The same proof applies here.

The same criterion holds true for serializability:

**Corollary 2.1 - The Serializability Condition** (e.g., [Bern 87])

A single version history H is serializable (SER) if and only if CSG(H) is cycle-free.

Recoverability (REC; special cases: cascadelessness (ACA - avoiding cascading aborts), strictness; see [Bern 87], [Hadz 88], [Raz 90]) is an essential property of histories when aborted transactions are present (i.e., in all real situations). Recoverability guarantees that committed transactions read only resource versions (states) written by committed transactions, and hence, no committed transactions read corrupted (aborted) versions, which later become inaccessible.

### 2.3 Updaters vs. queries

Only **updaters** (read-write transactions), which modify resources, determine the resources’ consistency. However, also **queries** (read-only transactions), need to provide consistent results. Non-SV, MV resources allow to achieve 1SER in higher concurrency and performance than with SV resources, by serializing queries before updaters (e.g., [Agra 89], [Agra 92], [Bobe 92a], [Bobe 92b], [Chan 85], [Moha 92], [Weih 87]).

#### 2.3.1 Updaters

Updaters employ any access strategy/mechanism that guarantees 1SER and recoverability. The solutions referenced above implement locking for updaters, and most of them guarantee strictness.

#### 2.3.2 Queries

Performance gain is achieved by enforcing queries to access already committed resource versions, and thus avoiding them from blocking updaters, or being blocked by updaters due to locks. This also implies that no query is involved in a deadlock and needs to be aborted.

The common mechanism for queries (e.g., see the references above), which guarantees 1SER (e.g., [Bern 87]), implements timestamps (members of any totally ordered set). Each **updater** is assigned with a timestamp upon *logging the decision*. This timestamp is used to timestamp all the resource versions committed by that updater. Each **query** is assigned with a timestamp upon *starting*. A query reads committed resource versions, that are timestamped with the latest timestamp that is still earlier than the query’s timestamp. This means that each for resource versions older than the latest version, that is younger (by timestamp) than the oldest running query, can be garbage-collected (*transient multi versioning*).

[Bobe 92b] and [Garc 82] describe correctness criteria (*consistency levels, which define different history properties*) for queries, that generalize (and may violate) 1SER. However, 1SER is maintained for updaters (i.e., the histories’ projections on updaters are 1SER). [Bobe 92b] presents mechanisms for implementing the generalized criteria, and shows that being less constraining than 1SER, they provide additional performance gain.

### 3 Commitment Ordering (CO)

Commitment Ordering (CO) is a property of histories that guarantees 1SER. A history is CO if the order induced by the conflicts between any two committed transactions matches the order of their commit events.

CO allows access by conflicting operations, while using any access scheduling strategy. This allows CO to be implemented also in a nonblocking manner (*optimistic*; see [Kung 81]), which guarantees deadlock-freedom. The price for this, however, is the possibility of *cascading aborts* when recoverability is applied.

**Definition 3.1 (Raz 90)**

A history is in CO, if for any committed transactions \(T_1, T_2\), whenever \(T_2\) is in a conflict with \(T_1\), then \(e_1 < e_2\).

Formally: \(e_1 = c\) and \(e_2 = c\) and \((T_2\) is in a conflict with \(T_1\)) implies \(e_1 < e_2\).

CO implies 1SER (proving this is almost identical to proving that CO implies SER; see [Raz 90]):

**Theorem 3.1**

\(1SER \supseteq CO\) (strict containment).

---

1 [Bern 87] ignores what we define as conflicts involving unread versions. Ignoring such conflicts is analogous, in the single-version case, to ignoring conflicts involving unread resource states.
Note that the interpretation of the commit event is flexible. The commit event is typically considered the commit decision logging event. In a multi RM environment this event usually occurs in a system component that executes an atomic commitment (AC) protocol (typically named transaction manager, TM; see more in section 5 below).

4 Commitment Ordering scheduling

This section describes a generic CO algorithm which provides algorithmic characterizations for the property CO. Though this algorithm is applicable to all transactions involved, in a MV environment it is possible to apply CO to updaters only, while allowing queries to use different scheduling strategies (as described in section 2.3). In this case only the histories’ projections on updaters are in CO.

The Commitment Order Coordinator (COCO) is a component of a RM’s scheduler that guarantees generating CO histories. The generated histories are not necessarily recoverable. Recoverability, if required, can be applied by enhancing the COCO or by an external mechanism (see more below).

A COCO maintains a serializability graph, the USG, of all undecided transactions. Every new transaction processed by the RM is reflected as a new node in the USG; every conflict between transactions in the USG is reflected by a directed edge (an edge between two transactions may represent several conflicts).

\[ \text{USG}(H) = (UT, C) \]

- \( UT \) is the set of all undecided transactions in a history \( H \)
- \( C \) is a subset of \( UT \times UT \) is the set of directed edges between transactions in \( UT \) where an edge is from \( T_1 \) to \( T_2 \), if \( T_2 \) is in conflict with \( T_1 \).

The set of transactions in the USG, aborted as a result of committing a transaction \( T \) (to prevent any future commitment ordering violation) is defined as follows:

\[ \text{ABORT}_{C}(T) = \{ T' \mid \text{The edge } T' \rightarrow T \text{ is in } C \} \]

These aborts cannot be compromised, as stated by lemma 4.1:

Lemma 4.1 ([Raz 90])

Aborting all the members of \( \text{ABORT}_{C}(T) \), after \( T \) is committed, is necessary for guaranteeing CO (assuming that every transaction is eventually decided).

Proof:

Suppose that \( T \) is committed. Let \( T' \) be some transaction in \( \text{ABORT}_{C}(T) \). Thus \( T' \) is undecided when \( T \) is committed. If \( T' \) is later committed, then \( c < c' \), where \( c \) and \( c' \) are the commit events of \( T \) and \( T' \) respectively. However, \( T \) is in a conflict with \( T' \), and thus, CO is violated.

\[ \text{Lemma 4.1 is the key for the CO algorithm, executed by the COCO.} \]

Algorithm 4.1 - The CO Algorithm ([Raz 90])

Repeat the following steps:

- Select any transaction in the \text{ready} state (i.e., a transaction that has completed processing) \( T \) in the USG (using any criteria, such as by priorities assigned to each transaction; a priority can be changed dynamically as long as the transaction is in the USG), and commit it.
- Abort all the transactions in the set \( \text{ABORT}_{C}(T) \), i.e., all the transactions (both \text{ready and active}) in the USG that have an edge going to \( T \).
- Remove any \text{decided} transaction (\( T \) and the aborted transactions) from the graph (they do not belong in the USG by definition).

Remark: During each iteration the USG should reflect all operations’ conflicts until commit.

For MV resources incoming edges can be generated in the serializability graph for ready and committed transactions (this is impossible for the SV case). Thus, for MV resources the condition in lemma 4.1, implemented by the CO algorithm, is necessary, but not sufficient for guaranteeing CO. The following condition provides the required additional restriction:

Definition 4.1 - The Multi Version Commitment Ordering condition

- $$\text{MVCO}$$

No committed version older than the last committed version is read. (However, uncommitted versions may be read without any restriction.)

Note that MVCO is always true for SV resources.

The next theorem defines under what conditions the CO algorithm is correct.

Theorem 4.1

The CO algorithm (algorithm 4.1) generates CO, MV histories only (guarantees CO), if and only if MVCO is maintained.

Proof:

(i) If MVCO is maintained, then all the histories generated are proven to be in CO, i.e., CO is guaranteed:

This claim is proven by induction on the number of iterations by the algorithm, starting from an empty history \( H_0 \), and an empty graph \( \text{USG}_0 = \text{USG}(H_0) \). \( H_0 \) is in CO.

Assume that the history \( H_0 \), generated after iteration 0, is in CO. \( \text{USG}_0 \) (in its \text{UT} component) includes all the undecided transactions in \( H_0 \). Now perform an additional iteration, number \( n+1 \), and commit transaction \( T_n \) (without loss of generality - wlg) in \( \text{USG}_n \). \( H_{n+1} \) includes all the transactions in \( H_n \) and the new (undecided) transactions that have been generated after completing step \( n \) (and are in \( \text{USG}_{n+1} \)).

Examine the following cases after completing iteration \( n+1 \):

- Let \( T_2, T_3 \) (wlg) be two committed transactions in \( H_n \). If \( T_3 \) is in conflict with \( T_2 \), then \( c_2 < c_3 \) since \( H_n \) is in CO by the induction hypothesis.
- Obviously, \( c_2 < c_3 \) for every (previously) committed transaction \( T_2 \) in \( H_n \) with which \( T_1 \) is in a conflict.
- Suppose that a committed transaction \( T_3 \) is in a conflict with \( T_1 \).
  If the conflict is \( \text{ww} \) or \( \text{wr} \) then \( T_1 \) is in \( \text{ABORT}_{C}(T_3) \), and thus aborted when \( T_2 \) was committed earlier. A contradiction.
  If the conflict is \( \text{rw} \), assume (wlg) that the conflict is on resource \( x \). If \( r_{1}[x_j] < w_{2}[x_j] \) then the case of \( \text{ww} \) and \( \text{wr} \) conflicts applies, and we reach the same contradiction. Suppose that \( w_{2}[x_j] < r_{1}[x_j] \). Since for having a \( \text{rw} \) conflict \( w_{2}[x_j] < w_{2}[x_j] \) is true (see the definition of a \( \text{rw} \) conflict in section 2.2), the operation \( r_{1}[x_j] \) does not represent reading the last committed version of \( x \), and thus MVCO is violated. A contradiction.
The cases above exhaust all possible pairs of conflicting committed transactions in \( H_{n+1} \). Hence, \( H_{n+1} \) is in CO.

(ii) Suppose that the condition above is not always maintained. Then, the following scenario is possible:

Suppose that the USG has a single node \( T_k \). Let \( x_i \) be the last committed version of \( x \), and \( x_i \) the previous committed version. Suppose that \( T_k \) reads versions \( x_i \) and \( x_i \) and then is being committed. Let \( H \) be the history at that stage. By the definitions of conflicts (section 2.3.1) the following conflicts exist:

- \( T_k \) is in wr conflicts with \( T_i \) (reading \( x_i \)) and \( T_j \) (reading \( x_j \)).
- \( T_i \) is in a rw conflict with \( T_k \), since \( r_k[x_j] \) exists, and \( w_j[x_j] < w_k[x_i] \).

Since \( T_i \) is a committed transaction as well, \( CSG(H) \) has the edges \( T_k \rightarrow T_i \) and \( T_i \rightarrow T_k \). Hence, \( CSG(H) \) has a cycle, and by theorem 2.1 \( H \) is not in 1SER. Thus, \( H \) is not in CO by theorem 3.1, and CO is not guaranteed.

The version of the CO algorithm (algorithm 4.1), applied to transactions that obey MVCO, is referred to as the MVCO algorithm. By theorems 3.1 and 4.1 we conclude corollary 4.1:

**Corollary 4.1**

The MVCO algorithm generates 1SER histories only (guarantees 1SER).

Note that aborting the transactions in \( \text{ABORT}_{\text{CO}}(T') \) when committing \( T \) prevents any cycle involving \( T \) being generated in the CSG in the future. This observation is a direct way to show that the algorithm 4.1 guarantees 1SER. If a transaction \( T \) exists, that does not reside on any cycle in the USG, then a transaction \( T \) exists with no incoming edges from any other transaction. \( T \) can be committed without aborting any other transaction since \( \text{ABORT}_{\text{CO}}(T') \) is empty. If all the transactions in the USG are on cycles, at least one transaction has to be aborted when committing another one. If the COCO is combined with a scheduler that guarantees (local) 1SER, cycles in the USG are either prevented, or eliminated by the scheduler aborting transactions.

The COCO is completely passive regarding the access strategy implemented by the scheduler. Thus, any concurrency control can be utilized as long as the COCO is updated with conflict information. For implementation oriented aspects of the COCO see [Raz 91a].

**Recoverability** (or cascadelessness (ACA) or strictness; see [Bern 87], [Hadz 88], [Raz 90]) can be enforced either by the local concurrency control mechanism or by an enhancement of algorithm 4.1 ([Raz 90]). If recoverability (cascadelessness, strictness) is enforced by the local concurrency control mechanism, also the combined mechanism (i.e., the concurrency control mechanism together with the CO algorithm) guarantees it (see the recoverability inheritance theorem in [Raz 90]). Like algorithm 4.1, also the enhanced algorithm is correct for MV resources, provided that the MVCO condition is maintained.

Cascadelessness is guaranteed if no write-read conflicts (wr; section 2.2) are reflected in the USG. The following condition, a special case of MVCO, guarantees cascadelessness when applied with the CO algorithm:

**Definition 4.2 - The Multi Version Cascadelessness condition**

- **MVACA**
  
  Whenever a resource is read, its last committed version is read.

Note that the MVACA condition applied to queries also guarantees that they are serialized before updaters, exploiting the benefits of multi-versioning for CO.

### 5 Multi RM environment

A multi RM transaction consists of one or more *local subtransactions*. A local subtransaction is the portion of a transaction within a single RM. A local subtransaction obeys the transaction rules and has states as defined in section 2. It is assumed that an *atomic commitment (AC)* protocol is applied to guarantee *atomicity* across the local subtransactions.

An AC protocol implements the following general scheme each time a transaction is decided:

- **AC**

  Each participating RM votes either YES or NO (also absence of a vote within a specified time limit may be considered NO) after the respective *local subtransaction* has reached the ready state, or votes NO if unable to reach the ready state. The transaction is committed by all RMs if and only if all have voted YES. Otherwise it is aborted by all the RMs.

The YES vote is an obligation to end the local subtransaction (commit or abort) as decided by the AC protocol. After voting YES the RM cannot affect the decision anymore. The fact that AC is used allows one to assume that like in the single RM case, also a distributed (committed) transaction has a single commit event, interpreted as *logging the commit decision*, which is executed by the AC protocol (in a distributed Transaction Manager, TM, component) for global transactions (i.e., transactions that span two or more RMs). Each RM applies by itself the commit decisions to its *local* transactions. This means that all participating RMs, that enforce a certain commit decision order (which can be applied also for global transactions; see below), view (portions of) the same commitment (partial) order of commit decision events. By theorem 3.1 this implies global 1SER.

### 5.1 Distributed CO algorithms

In a multi RM environment, the COCO described in section 4 also votes on behalf of its RM in AC protocols. The COCO typically receives a request, via an AC protocol to commit some transaction T in the USG. If the COCO can commit the transaction, it votes YES via AC, which is an obligation to either commit or abort according to the decision reached by the AC protocol. When the COCO commits T, all transactions in \( \text{ABORT}_{\text{CO}}(T) \) need to be aborted (by lemma 4.1). Thus the COCO has to delay its YES vote on T, if it has voted YES on any transaction in \( \text{ABORT}_{\text{CO}}(T) \), since not delaying the YES vote can result in a contradiction if the AC protocol decides to commit both (Note that committing first T’ in \( \text{ABORT}_{\text{CO}}(T’) \), and then committing T is not sufficient, since T can be in \( \text{ABORT}_{\text{CO}}(T’) \) in another RM, and committing both will result in a global serializability violation; see also CD³C in [Raz 90]). By similar ar-

---

1. Local subtransactions reflect transaction partitioning over RMs, and are independent of possible transaction partitioning to nested subtransactions by an application.
guments the COCO cannot vote YES on T, if T is in \( \text{ABORT}_{\text{CO}}(T') \) for some T’ on which it has already voted YES. If YES vote on T is possible, the COCO may either choose to do so immediately upon being requested (the "nonblocking without delays" approach), or to delay the voting for a given, predetermined amount of time ("nonblocking with delays"). During the delay the set \( \text{ABORT}_{\text{CO}}(T) \) may become smaller or empty, since its members may be decided and removed from the USG. For SV resources \( \text{ABORT}_{\text{CO}}(T) \) cannot increase after T has reached the ready state. Instead of immediately voting, or delaying the voting for a given amount of time, which may result in aborts, the COCO can block the voting on T until all transactions in \( \text{ABORT}_{\text{CO}}(T) \) are decided. However, if another RM in the environment also blocks, this may result in a global deadlock (e.g., if T’ is in \( \text{ABORT}_{\text{CO}}(T) \) for one RM, and T is in \( \text{ABORT}_{\text{CO}}(T') \) for another RM). Aborting transactions by timeout is a common mechanism for resolving such deadlocks. Controlling the timeout by the AC protocol, rather than aborting independently by the RMs, is preferable for preventing unnecessary aborts. Note that aborting transactions by the COCO is necessary only if a local cycle in its USG is not eliminated by some external entity (e.g., a scheduler that generates a cycle-free USG or one that uses aborts to resolve cycles), or if a global cycle (across two or more local USGs) is generated. The cycles are generated exclusively by the way resource access is scheduled and are independent of the commit order. Thus the COCO does not have to abort more transactions that need to be aborted for serializability violation prevention. The following is an AC based CO algorithm (CO-AC), which combines algorithm 4.1 with a generic AC protocol. By the arguments given above, the algorithm enforces CO (and thus 1SER) globally if executed by each RM in the environment, and if the condition MVCO is maintained:

**Algorithm 5.1 - CO-AC, The distributed CO algorithm ([Raz 90])**

Repeat the following steps:

- Select any transaction T in the USG, that meets the following conditions (using any criteria; selecting T that minimizes the cost of aborting the transactions in the set \( \text{ABORT}_{\text{CO}}(T) \), when T is later committed, is desirable):
  - T is in the ready state (i.e., has completed processing).
  - T is not in \( \text{ABORT}_{\text{CO}}(T') \) of any T’ on which a YES vote has been issued (see discussion above).
  - No YES vote has been issued on any transaction in \( \text{ABORT}_{\text{CO}}(T) \) (see discussion above).
- If such a T is found, then vote YES on T.
- Later, asynchronously (after receiving a notification about the decision on T), do the following:
  - If T is committed by the AC protocol, commit T and abort all the transactions in the set \( \text{ABORT}_{\text{CO}}(T) \); otherwise (T is aborted by the AC protocol) abort T.
  - Remove T and the (possibly) other aborted transactions from the graph (they do not belong in the USG by definition).

The distributed CO algorithm (algorithm 5.1) above assumes that all the transactions are decided by the AC protocol. If a RM’s scheduler “knows” that a ready transaction is local, i.e., does not span other RMs (vs. global which spans two or more), the (local) COCO itself decides whether to commit or abort it. Algorithm 5.2 is an enhancement of the distributed CO algorithm, where local transactions are decided locally, rather than being decided by the AC protocol. A knowledge about a transaction being local is usually acquired by a RM by an explicit notification from an application (program) that interacts with the RM.

**Algorithm 5.2 - Distributed CO algorithm that differentiates local transactions**

Repeat the following steps:

- Select any ready transaction (i.e., a transaction that has completed processing) T in the USG, that is not in \( \text{ABORT}_{\text{CO}}(T') \) of any T’ on which a YES vote has been issued, and no YES vote has been issued on any transaction in \( \text{ABORT}_{\text{CO}}(T) \) (using any criteria, possibly by priorities assigned to each transaction; a priority can be changed dynamically as long as the transaction is in USG).
- If T is global then vote YES on T, and later, asynchronously, do the following:
  - If T is committed by the AC protocol, commit T and abort all the transactions in the set \( \text{ABORT}_{\text{CO}}(T) \); otherwise abort T.
  - Remove T and other aborted transactions (if any) from the graph (they do not belong in the USG by definition).
- Otherwise, if T is local, do the following:
  - Commit T and abort all the transactions in the set \( \text{ABORT}_{\text{CO}}(T) \).
  - Remove T and other aborted transactions (if any) from the graph (they do not belong in the USG by definition).

Remark: If all the transactions are assumed to be global, i.e., decided by the AC protocol (even local), this algorithm reduces to algorithm 5.1.

**Example 5.1 ([Raz 90])**

The behavior of algorithms 5.1, 5.2 is demonstrated by the two scenarios below:

![Figure 5.1](image1)

To avoid aborting T1, RM1 first votes YES on T1, attempting to have it committed before voting YES on T3. Similarly, RM2 ends T1 and T2 before voting on T3. After both T1 and T2 are removed from the USGs, T3 can be voted on and be committed.

![Figure 5.2](image2)

RM1 votes YES on T1 and avoids voting on T2 until T1 is decided. RM2 votes YES on T2 and avoids voting on T1 until T2 is decided. This situation results in a deadlock, which can be resolved by the AC protocol aborting either T1 or T2 (e.g., by timeout).
Theorem 5.1
Algorithms 5.1 and 5.2 guarantee global MV CO histories, if and only if the MVCO condition is maintained.

When a RM has the knowledge about transactions being local, the CO condition can be relaxed, and global 1SER can be achieved by enforcing locally in each RM the Extended Commitment Ordering (ECO) property together with (local) 1SER. ECO is defined in [Raz 91b], where a distributed ECO algorithm, a generalization of algorithm 5.2 above, is presented. Unlike CO, ECO enforces a (partial) order on the commit decision events of global transactions only. Like the CO algorithms, also the ECO algorithm is applicable to MV resources, provided that the MVCO condition is maintained.

We can now summarize some relationships between local CO histories (of individual RMs) and the properties induced by them on respective global histories. Note that the commit events in the definition of CO are interpreted as the commit decision events, occurring at an AC protocol system component (distributed Transaction Manager, TM).

Definition 5.1
For any history property X a (global) history is in Local-X, if the respective (local) history of every RM in the environment is in X.

Note that like for serializability (SER), also Local-1SER ⊃ 1SER, i.e., a global history is not necessarily 1SER even if each respective local history is 1SER. However, since AC guarantees a single commit (decision) event for each committed transaction, the theorem below follows by definitions 3.1 and 5.1:

Theorem 5.2
Local-CO = CO
i.e., a (global) history is in CO if and only if it is in Local-CO.

Proof:
(i) Assume that a history H is in Local-CO. By the arguments given in the discussion above, for each RM that enforces CO and for any two committed transactions T1 and T2, T2 is in a conflict with T1 in the RM (reflected by the RM's operations), if and only if c1 < c2 (the commit decision events). Thus, for all the RM participating in both T1 and T2, either T2 is in a conflict with T1, or no conflict exists (the third possibility, T2 being in a conflict with T2, implies that c2 < c1, a contradiction). Thus, the condition above is globally true, and H is in CO.

(ii) If H is in CO, then T2 being in a conflict with T1 implies c1 < c2. This is true for all the RM that view this conflict (i.e., where T2 is locally in a conflict with T1). Thus every RM is in CO, and H is in Local-CO.

The following theorem is a consequence of theorems 3.1 and 5.2:

Theorem 5.3
1SER ⊃ Local-CO
i.e., if a (global) history is in Local-CO then it is 1SER.

Theorem 5.3 means that if each RM provides CO, then global 1SER is guaranteed.

5.2 Distributed queries
As was described in section 2.3, the main motivation for transient multi versioning is serializing queries before all the running updaters. If the MVACA condition (definition 4.2) is applied (at least for queries) with the distributed CO algorithm, then this goal is achieved, and no specialized access strategy for queries (other than CO) is required. Otherwise, for non-SV, MV resources, queries may not participate in CO algorithms, but rather use other access strategies. For SV resources, however, it is usually unavoidable that queries may serialize after some running updaters. Thus, to guarantee 1SER, a query accessing SV resources should be treated as an updater, i.e., participate in a CO algorithm (unless query serialization before the updaters is somehow guaranteed).

The timestamping mechanism described for queries in section 2.3 can be used for multi RM queries as follows:

• Upon logging the commit decision of an updater, the AC protocol assigns a global timestamp (i.e., unique in the multi RM environment) to the updater. This timestamp is propagated with the updater’s commit messages to all the participating RMs. The updater’s modified resources (its respective new versions) are timestamped upon locally being committed (by respective RMs).

• A global timestamp is assigned to each query upon starting, and it propagates with the query invocations over the RMs spanned by the query. For any resource read, the query reads the latest (committed) version timestamped with a timestamp smaller than the query’s timestamp.

Remarks:
• The timestamp mechanism is unnecessary, if the MVACA condition (definition 4.2) is enforced for queries with the CO algorithm.

• Generating globally unique timestamps requires synchronizing the distributed AC protocol system components (transaction managers, TMs), i.e., additional synchronization messages are required.

• These timestamps do not impose any blocking on updaters’ operations.

• The timestamps piggy-backed on commit messages violate RM autonomy as defined in section 1.

6 Necessary and sufficient conditions for guaranteeing 1SER globally

Definition 6.1 ([Raz 90])
A system (any collection of interacting components/objects) guarantees a property P, if every reachable state of the system has property P.

Thus, a property P1 is necessary (a necessary condition) to guarantee property P2, if when all the reachable states have property P2, then they also have property P1. Equivalently, the existence of a reachable state that does not have property P1 implies the existence of a (possibly different) reachable state that does not have property P2.

Footnote:
1Follows by theorem 2.1, by examining a global CSG and its respective local CSGs.
We concentrate on the case where systems’ states are histories generated by the system\(^1\).

By theorem 3.1 Local-CO is a sufficient condition for a global history to be in 1SER. In this section it is shown that under the requirements of RM autonomy (as defined in section 1), guaranteeing Local-CO is also a necessary condition for guaranteeing 1SER globally in a multi-RM environment. By theorem 3.1 and definition 6.1 we immediately conclude the following:

**Theorem 6.1**

Guaranteeing Local-CO is a sufficient condition for guaranteeing global 1SER.

In what follows we show, using (informally) knowledge theoretic arguments (e.g., see [Halp 87], [Hadz 87]), under what conditions guaranteeing Local-CO is also a necessary condition for guaranteeing global 1SER:

**Theorem 6.2**

If all the RMs in the environment are autonomous (i.e., the RMs are coordinated solely via AC protocols), then guaranteeing Local-CO is a necessary condition for guaranteeing global 1SER.

Proof:

Let the environment guarantee (global) 1SER. Since guaranteeing 1SER implies guaranteeing Local-1SER (by theorem 2.1, considering the global CSG vs. the respective local CSGs), the environment guarantees Local-1SER as well.

Suppose that guaranteeing Local-CO is not necessary for guaranteeing global 1SER under the conditions of RM autonomy. Let Y be the (most general) history property (class) such that Local-Y is necessary for guaranteeing global 1SER under RM autonomy conditions. Since Local-CO is a sufficient condition for 1SER (theorem 5.3), the following is true:

(i) \( Y \supseteq CO \) (strict containment; if \( CO \supseteq Y \), then CO is the necessary property, contrary to our assumption above; if Y and CO are incomparable, Y cannot be the necessary property).

and thus:

(ii) \( 1SER \supset Local-Y \supset Local-CO \).

Suppose that the environment includes two RMs (with local 1SER histories; see above). Assume that RM 1 generates histories in Y, and RM 2 generates histories in CO, which are also in Y (by (i)). Hence, the global histories are in Local-Y (by definition of Local-Y), and thus also in 1SER (by (ii)). We now schedule transactions in the environment above in such a way that the generated global history H includes the following elements and relationships among them:

Since RM1 guarantees Y, we can schedule two global transactions (without loss of generality) \( T_1 \), \( T_2 \) such that \( T_2 \) is in a conflict with \( T_1 \) (for RM1), both are committed, and \( c_2 < c_1 \) (otherwise, if we cannot, \( Y \) collapses to CO, contrary to (i)).

The above two transactions can be scheduled in RM2 in such a way that \( T_1 \) is in a conflict with \( T_2 \) (for RM2). The commitment order above, i.e., \( c_2 < c_1 \) is in agreement with CO for RM2, and thus consistent with the operation of RM2.

Since no RM has any knowledge about the conflicts of transactions in the other RM, the situation described above is possible in normal operation.

However, the conflicts above imply that CSG(H) has a cycle through \( T_1 \) and \( T_2 \), i.e., H is not in 1SER (theorem 2.1), and we have reached a contradiction.

Hence, no such property Y exists, and CO is a necessary condition.

Note that when the RMs have the knowledge of autonomous RMs, and in addition the knowledge about transactions being local, i.e., they can distinguish between local and global transactions (defined to have the extended knowledge autonomy (EKA) property), then the more general property, Extended Commitment Ordering (ECO), is the necessary condition for guaranteeing global 1SER (see [Raz 91b]; the proof presented there is applicable to MV histories as well).

By theorems 6.1 and 6.2 we conclude:

**Corollary 6.1**

Guaranteeing CO locally (Local-CO) is a necessary and sufficient condition for guaranteeing (global) 1SER in an environment of autonomous RMs.

### 7 Conclusion

This work provides a theoretic framework to deal with mixed, single and multi version resources. It demonstrates the Commitment Ordering (CO) property as an integrated solution for a correct global concurrency control in a heterogeneous environment of multi autonomous RMs. The RMs may use different (any) concurrency control mechanisms. The correctness criteria utilized are one-copy serializability (1SER) and recoverability (REC). Guaranteeing CO locally is shown to be a necessary and sufficient condition for guaranteeing 1SER globally over autonomous RMs. Atomic commitment (AC) based distributed algorithms for enforcing CO (and thus 1SER) globally are presented. The simplicity of the CO algorithms and the lack of any communication overhead (i.e., no messages beyond AC) in the distributed case, make CO an efficient and practical basis for concurrency control in a distributed, multi RM, high performance transaction processing environment. The MVCO and MVACA conditions (see section 4) allow the CO algorithm to exploit the transient multi-versioning benefits for queries. However, it is also possible to apply the CO algorithms to updaters only, for preserving resource consistency. This can be done without interfering with specialized MV resource access strategies applied for queries, while query result consistency is maintained.

### References


\(^1\)The related state transition function has a history and an event set as arguments. Its values on a given history and its prefixes, and on a given event set and its subsets, are compatible. Such a function is neither formalized nor explicitly used here.


[Geor 91] Dimitrios Georgakopoulos, Marek Rusinkiewicz, Amit Sheth, "On serializability of Multi database Transactions Through Forced Local Conflicts", in Proc. of the Seventh Int. Conf. on Data Engineering, Kobe, Japan, April 1991.


[Raz 92] Yoav Raz, "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers, Using Atomic Commitment", in the Proc. of the Eighteenth Int. Conf. on Very Large Data Bases, pp. 292-312, Vancouver, Canada, August, 1992. (An abridged version of [Raz 90]).


[Veij 92] J. Veijalainen, A. Wolski, "Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases", in Proc. of the Eighth Int. Conf. on Data Engineering, Tempe, Arizona, February 1992.