

Transaction Management, Concurrency Control and Recovery in Relational Databases

Piyush Ojha
School of Computing and Mathematics
University of Ulster
pc.ojha@ulster.ac.uk

23 November 2005

Abstract

Database transactions are required to be atomic, consistent, isolated and durable. It is the programmer's responsibility to define the transactions appropriately and ensure that they are (individually) consistent. Their atomicity must be ensured by the Database Management System (by rolling back failed transactions). Their concurrent scheduling so that transactions remain effectively isolated and their durability after commit is also the responsibility of the DBMS.

We discuss serialisable, recoverable and cascadeless schedules and concurrency control protocols for generating such schedules. Techniques for recovering from failure are also discussed.

1 Sources

1. A Silberschatz, H F Korth and S Sudarshan, Database System Concepts, 4th Edition, Chapters 15-17.
2. R El Masri and S B Navathe, Fundamentals of Database Systems, 4th Edition, Chapters 17-19
3. Thomas Connolly and Carolyn Begg, Database Systems: A Practical Approach to Design, Implementation and Management, 3rd Edition, Chapter 19.

2 Transactions

Transaction: A database transaction is a collection of database operations that form a **logical** unit of work.

Example: Transfer £100.00 from a savings account to a current account.

This is a single transaction comprising two operations:

O1: Withdraw £100.00 from the savings account.

O2: Credit £100.00 to the current account.

The two operations together constitute a unit of work.

In database transactions the most significant operations are read (i.e. data retrieval) and write (i.e. update).

We will represent the money transfer transaction as follows:

Transaction T_1

```
read(A);
```

```
A:=A-100.00;
```

```
write(A);
```

```
read(B);
```

```
B:= B+100.00;
```

```
write(B);
```

(A is the balance in the savings account and B the balance in the current account.)

2.1 ACID Properties of Database Transaction

Data integrity requires that the database management system maintains the following properties of transactions (collectively known as ACID properties):

Atomicity: Either all operations in a database transaction are completed, or none.

Consistency: Each transactions must maintain the consistency of the database, i.e. take it from one consistent state to another consistent state.

Isolation: Each transaction must be unaware of other transactions that are executing concurrently.

Durability: After a transaction completes successfully (commits), the changes it makes to the database must be permanent.

2.1.1 Need for ACID Properties of the Example Transaction

Atomicity: Clearly, if we withdraw money from the savings account but do not credit it to the current account, the final state of the database would be inconsistent (in the sense that it does not correspond to the state of the real world). Logically, both operations, withdrawl from the savings account and crediting to the current account must be completed.

Atomicity of transactions is ensured by rolling back a transaction which is not going to complete successfully. This is ensured by the recovery-management component of a DBMS.

Consistency: The database is consistent if it reflects the true state of the world. In the example transaction, the sum of the balances of the savings and current accounts must be the same before and after the transactions.

(Note that at an intermediate state while the transaction is executing, for example after the withdrawl has been made but before the current account is credited, the system is temporarily in an inconsistent state. The dabase management system ensures that these temporarily inconsistent states are not 'visible'.)

The programmer has the responsibility of ensuring that an individual transaction is consistent. (The database management system can't ensure consistency if the programmer withdraws £100.00 from the savings account and adds only £50.00 to the current account.)

Isolation: When many transactions are being executed concurrently, efficient utilisation of resources requires that the actual database operations are interleaved. This requires care. For example consider the following **schedule** for the operations belonging to two transactions:

Transaction T_1	Transaction T_2
...	read(A);
...	A:=A+50.00
read(A);	...
A:=A-100.00;	...
...	write(A);
write(A);	...
read(B);	...
B:= B+100.00;	...
write(B);	...

In this *schedule* – the particular interleaving of database operations – the update of A by transaction T_2 is lost. The outcome is not as if T_1 and T_2 were isolated from each other. Clearly, this is undesirable because the database is not consistent.

The concurrency-control component of the database management system ensures isolation.

Durability: Once the money transfer has been completed and the account balances updated, the changes to the database must persist (even if there is a subsequent system failure).

A committed transaction can not be undone by aborting it. Its effects can only be undone by executing another compensating transaction.

The recovery-management component of the DBMS also ensures durability of transactions.

We will examine some of the techniques used by DBMSs to ensure atomicity, isolation and durability of transactions.

2.2 Transaction State

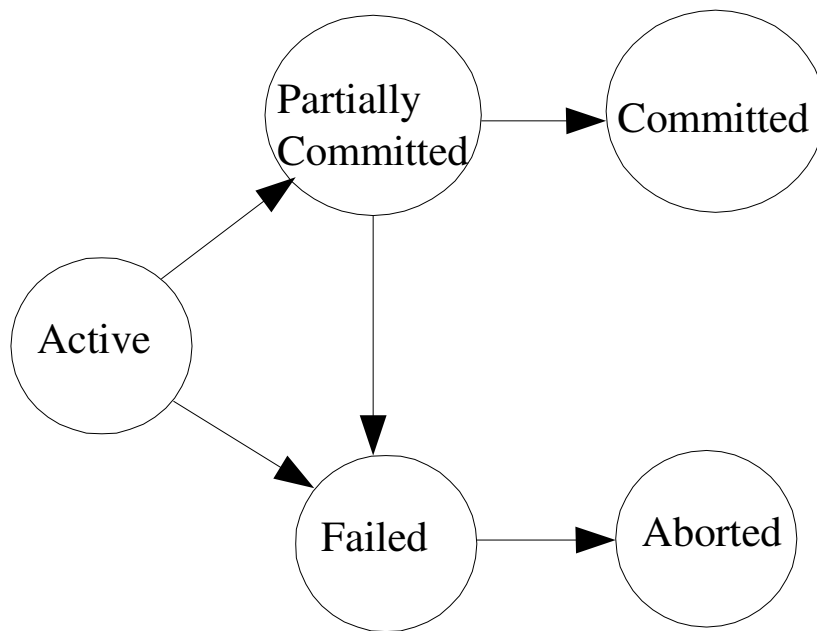


Figure 1: State Transition Diagram for Transaction

A transaction is

Active when it is initiated and remains so while it is executing;

Partially Committed after the final statement has been executed;

Failed after it is discovered that further execution can not proceed;

Aborted after a failed transaction has been rolled back and the database has been restored to an earlier consistent state;

Committed after successful execution.

2.3 Scheduling Concurrent Transactions

If each isolated transaction is consistent – and the programmer has the responsibility for ensuring this – the simplest way of ensuring the consistency of the database is to execute the transactions *serially*, i.e. starting a transaction only when the previous one has finished. However this is not efficient.

Multiple transactions must run concurrently (i.e. with their operations interleaved) so that there is

- better throughput and resource utilisation;
(Transactions require both processing and input-output. If run serially, the CPU would be idle while the transaction is reading from or writing to the backing storage. If run concurrently, one transaction can be processing the data while the other is engaged in I/O.)
- reduced waiting time.
(If run serially, a short transaction may have to wait for a long time for a long transaction to complete. Concurrent scheduling reduces the average response time.)

Ensuring that concurrent transactions do not undermine the consistency of data requires care.

2.3.1 Serial and Concurrent Schedules

Consider the following transactions:

Transaction T_1

```
read(A);  
A:=A-100.00;  
write(A);  
read(B);  
B:= B+100.00;  
write(B);
```

Transaction T_2

```
read(A);  
temp=0.2*A;  
A:=A-temp;  
write(A);  
read(B);  
B:= B+temp;  
write(B);
```

T_1 transfers £100.00 from account A to B whereas T_2 transfers 20% of the balance in A to B. We will assume that the initial balance in each account is £1000.00.

Transaction T_1	Transaction T_2
<pre> read(A); A:=A-100.00; write(A); read(B); B:= B+100.00; write(B); </pre>	<pre> read(A); temp=0.2*A; A:=A-temp; write(A); read(B); B:= B+temp; write(B); </pre>

Table 1: Schedule 1: A serial schedule where T_1 and T_2 are excuted in that order. At the end $A = 720.00$ and $B = 1280.00$, $A + B$ is the same before and after the transaction and the database is in a consistent state.

Transaction T_1	Transaction T_2
<pre> read(A); A:=A-100.00; write(A); read(B); B:= B+100.00; write(B); </pre>	<pre> read(A); temp=0.2*A; A:=A-temp; write(A); read(B); B:= B+temp; write(B); </pre>

Table 2: Schedule 2: An alternative serial schedule where T_2 is executed before T_1 . At the end $A = 700.00$, $B = 1300.00$ and the database is in a consistent state.

Transaction T_1	Transaction T_2
<pre> read(A); A:=A-100.00; write(A); read(B); B:= B+100.00; write(B); </pre>	<pre> read(A); temp=0.2*A; A:=A-temp; write(A); read(B); B:= B+temp; write(B); </pre>

Table 3: Schedule 3: A concurrent schedule equivalent to Schedule 1. After the transactions, $A = 720.00$ and $B = 1280.00$.

Transaction T_1	Transaction T_2
<pre> read(A); A:=A-100.00; write(A); read(B); B:= B+100.00; write(B); </pre>	<pre> read(A); temp=0.2*A; A:=A-temp; write(A); read(B); B:= B+temp; write(B); </pre>

Table 4: Schedule 4: A concurrent schedule that leaves the database in an inconsistent state. At the end $A = 900.00$ and $B = 1200.00$ and $A + B$ has been changed.

Conclusion: Not all concurrent schedules leave the database in a consistent state. The concurrency-control component of a DBMS ensures that only concurrent schedules that leave the database in a consistent state are used.

2.4 Serialisability

We can ensure that a concurrent schedule leaves the database in a consistent state by requiring that it is equivalent to a serial schedule. (Serial schedules always leave the database in a consistent state because the consistency of individual transactions is ensured by the programmer.) Such concurrent schedules are called **serialisable**.

From the point of view of scheduling, the only significant database operations are read and write.

There are two kinds of serialisability:

- conflict serialisability
- view serialisability.

2.4.1 Conflict Serialisability

Consider a schedule S and two **consecutive** operations I_i and I_j which belong to transactions T_i and T_j respectively.

If I_i and I_j refer to different data, they can be swapped without affecting the results produced by the schedule. However, if they refer to the same data, the order of the operations may matter.

Consider the following four possibilities:

- $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$
The order doesn't matter because both transactions read the same value of Q .
- $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$
Order matters; the value of Q read by T_i depends on whether I_i is executed before I_j or after.
- $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$
Order matter; this is really the same as the previous case with roles of I_i and I_j reversed.
- $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$
Order matters; it does not affect T_i or T_j but affects the value of Q read by subsequent operations in S , or if there are no subsequent reads, the final value of Q written to the datanase.

Thus the relative order of I_i and I_j matters if they operate on the same data item and at least one operation is a write. In this case, I_i and I_j are said to **conflict**.

Conflict Equivalence

If a schedule S can be transformed into anther schedule S' by swapping non-conflicting operations, S and S' are said to be conflict equivalent.

Conflict Serialisability

A schedule S is **conflict serialisable** if it is conflict equivalent to a serial schedule.

Clearly a conflict-serialisable schedule leaves the database in a consistent state.

Examples

Example 1

Transaction T_1	Transaction T_2
<code>read(A);</code> <code>write(A);</code> <code>read(B);</code> <code>write(B);</code>	<code>read(A);</code> <code>write(A);</code> <code>read(B);</code> <code>write(B);</code>

Table 5: Schedule 1 with only read and write statements shown.

Transaction T_1	Transaction T_2
<code>read(A);</code> <code>write(A);</code> <code>read(B);</code> <code>write(B);</code>	<code>read(A);</code> <code>write(A);</code> <code>read(B);</code> <code>write(B);</code>

Table 6: Schedule 3 showing only read and write operations. `read(A)` and `write(A)` operations in T_2 can be swapped with `read(B)` and `write(B)` operations in T_1 . The schedule is conflict equivalent to Schedule 1 and therefore conflict serialisable.

Example 2

Transaction T_3	Transaction T_4
read(A);	
write(A);	write(A);

Table 7: Schedule 5: A non-conflict-serialisable schedule. The two write operations can not be swapped, so the schedule can not be transformed into the serial schedule $(T_3; T_4)$. Similarly, because T_3 : read(A) can not be swapped with T_4 : write(A), it can not be transformed into the serial schedule $(T_4; T_3)$.

Notice the blind write (i.e. a write without a read) in T_3 .

Example 3

Transaction T_1	Transaction T_5
read(A);	
A:=A-100.00;	
write(A);	
	read(B);
	B:=B-50.0;
	write(B);
read(B);	
B:= B+100.00;	
write(B);	
	read(A);
	A:= A+50.0;
	write(A);

Table 8: Schedule 6: A concurrent schedule that is not conflict serialisable (Exercise: Why?) but still leaves the database in a consistent state

The last example shows that conflict serialisability is **sufficient but not necessary** for ensuring consistency of the database. Nevertheless, it is simpler to require all concurrent schedules to be conflict-serialisable.

2.4.2 View Serialisability

View serialisability is also based only on read and write operations but it is less stringent than conflict serialisability.

Schedules S and S' comprising the same set of transactions are said to be **view equivalent** if the following conditions are satisfied:

1. For each data item Q , if transaction T_i reads the initial value of Q in the schedule S , it must also read the initial value of Q in S' .
2. For every data item Q , if the value of Q read by T_i in S was previously written by T_j , the value of Q read by T_i in S' must also have been previously written by T_j .
3. For every data item Q , the final value must be written to the database by the same transaction in both schedules.

Conditions 1 and 2 ensure that every transaction reads the same data in the two schedules. Likewise, the third condition ensures that both schedules make the same updates to the database.

A concurrent schedule is **view serialisable** if it is **view equivalent** to a serial schedule.

Example

Transaction T_3	Transaction T_4	Transaction T_6
read(A);		
	write(A);	
write(A);		
		write(A);

Table 9: Schedule 7: A non-conflict-serialisable schedule that is view-serialisable. The schedule is view-equivalent to $(T_3; T_4; T_6)$.

Every conflict-serialisable schedule is also view-serialisable.

Every view-serialisable schedule that is not conflict-serialisable has blind writes.

2.4.3 Testing for Conflict Serialisability

Concurrency control modules of DBMSs are designed to generate conflict-serialisable schedules but it is still useful to be able to test a given schedule for conflict serialisability.

This can be done easily by constructing a precedence graph for a given schedule S . The precedence graph contains a vertex (or a node) for each transactions and a number of directed edges as follows.

For each pair of transactions T_i and T_j , there is a directed edge from T_i to T_j if any of the following conditions holds:

- T_i executes $\text{read}(Q)$ before T_j executes $\text{write}(Q)$;
- T_i executes $\text{write}(Q)$ before T_j executes $\text{read}(Q)$;
- T_i executes $\text{write}(Q)$ before T_j executes $\text{write}(Q)$;

If there is a directed edge from T_i to T_j , then T_i must come before T_j in any equivalent serial schedule.

It follows that the schedule is conflict serialisable if and only if its precedence graph does not contain any cycles.

Examples

Schedule 3

$T_1 : r_1(A); w_1(A); r_1(B); w_1(B);$

$T_2 : r_2(A); w_2(A); r_2(B); w_2(B);$

(Here $r_i(A)$ and $w_i(A)$ are read and write operation respectively on A by transaction T_i .)

$S : r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B).$

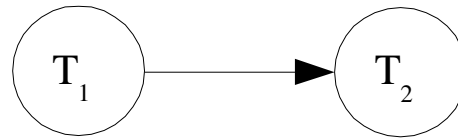


Figure 2: Precedence graph for Schedule 3

The precedence graph has no cycles, so the schedule is conflict serialisable.

Schedule 4

$T_1 : r_1(A); w_1(A); r_1(B); w_1(B);$

$T_2 : r_2(A); w_2(A); r_2(B); w_2(B);$

$S : r_1(A); r_2(A); w_2(A); r_2(B); w_1(A); r_1(B); w_1(B); w_2(B).$

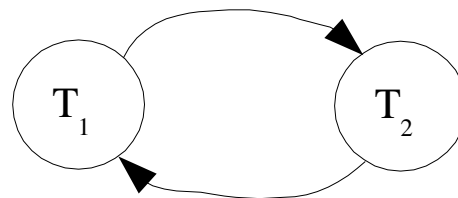


Figure 3: Precedence graph for Schedule 4

2.5 Recoverability

When S is conflict-serialisable and all transactions commit, the database is left in a consistent state.

What if one of the transactions (say T_i) fails?

We must rollback T_i as well as all other transactions that have read data written by T_i .

Recoverable Schedules

Transaction T_1	Transaction T_2
read(A); write(A);	
	read(A); write(A); commit;
read(B); write(B);	

Table 10: A non-recoverable schedule.

In the above schedule, T_2 commits immediately after writing A. If T_1 fails subsequently, we must roll back T_1 as well as T_2 (because it reads the value of A written by T_1). However, T_2 can't be rolled back because it has already committed. The schedule is **non-recoverable**.

A schedule is **recoverable** if, for every pair of transaction T_i and T_j where T_j reads a data item previously written by T_i , T_i commits before T_j .

Clearly concurrent schedules should be conflict-serialisable and recoverable.

Even when a schedule is recoverable, the failure of one transaction may cause rollback of several transactions (**cascading rollback**) and therefore wasted work. For example

Transaction T_1	Transaction T_2	Transaction T_3
read(A); read(B); write(A);		
	read(A); write(A);	
		read(A); write(A);
write(B)		

Table 11: A schedule with a **cascading rollback**. Failure of T_1 causes T_1 , T_2 and T_3 to be rolled back.

Cascadeless Schedule

A concurrent schedule is cascadeless if, for every pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , T_i commits before the read operation of T_j .

Every cascadeless schedule is also recoverable.

3 Concurrency Control

In this section we consider protocols for generating serialisable concurrent schedules. This is sufficient to ensure effective isolation of transactions.

We will consider two kinds of protocols:

- Lock-based protocols
- Timestamp-based protocols

3.1 Lock-Based Protocols

In these protocols, a transaction is given access to a data item only if it holds a lock on it.

There are two kinds of locks:

- **Shared:** A transaction T_i holding a shared lock on a data item Q can read it but not write it. Other transactions can simultaneously hold a shared lock on the data item.
- **Exclusive:** A transaction holding an exclusive lock on a data item can read as well as write it. Only one transaction can hold an exclusive lock on a data item at a time.

A transaction

- **requests** an appropriate lock from the lock manager when it needs access to a data item,
- **proceeds** with the operation when the lock is **granted**, and
- **releases** the lock when access is no longer required.

A lock is granted only if no other transaction holds an **incompatible** lock on the data item.

(A shared lock is compatible with a shared lock but incompatible with an exclusive lock. An exclusive lock is incompatible with both shared and exclusive locks.)

A transaction has to wait till a lock it has requested can be granted.

In the following, we include lock and unlock requests as part of the transaction (lock_X(A) is a request for an exclusive lock on A, lock_S(A) is a request for a shared lock and unlock(A) releases the lock).

Transaction T_1

```
lock_X(A);
read(A);
A:=A-100.00;
write(A);
unlock(A);
lock_X(B);
read(B);
B:= B+100.00;
write(B);
unlock(B);
```

Transaction T_2

```
lock_S(A);
read(A);
unlock(A);
lock_S(B);
read(B);
```

```
unlock(B);
print(A+B);
```

T_1 transfers £100.00 from the savings account (A) to the current account; T_2 prints the value of $A + B$.

Notice that both transactions release a lock as soon as they have finished using a data item.

This is not a good idea because it allows concurrent schedules that are not serialisable. For example, Schedule 1 below.

Transaction T_1	Transaction T_2	Lock Manager
lock_X(A)		grant_X(A, T_1);
read(A);		
A:=A-100.00;		
write(A);		
unlock(A);		
htb!	lock_S(A);	
	read(A);	grant_S(A, T_2);
	unlock(A);	
	lock_S(B);	
		grant_S(B, T_2);
	read(B);	
	unlock(B);	
	print(A+B);	
lock_X(B)		grant_X(B, T_1);
read(B);		
B:= B+100.00;		
write(B);		
unlock(B)		

Table 12: Schedule 1: A concurrent schedule for transactions T_1 , T_2 and the lock manager. If $A = B = 1000.00$ at the start of the transactions, T_2 prints 1900.00 instead of 2000.00 printed by either of the serial schedules (T_1, T_2) or (T_2, T_1) . This schedule is not serialisable and leaves the database in an inconsistent state because T_1 releases the lock on A too early.

We now change the transactions T_1 and T_2 slightly and alter the order of lock requests and lock releases. In particular, all lock requests are made before any locks are released.

Transaction T_3

```
lock_X(A);  
read(A);  
A:=A-100.00;  
write(A);  
lock_X(B);  
unlock(A);  
read(B);  
B:= B+100.00;  
write(B);  
unlock(B);
```

Transaction T_4

```
lock_S(A);  
read(A);  
lock_S(B);  
unlock(A);  
read(B);  
unlock(B);  
print(A+B);
```

This seemingly small change has a profound effect on the concurrent schedules. Any concurrent schedule for T_3 and T_4 is guaranteed to be serialisable.

Transaction T_3	Transaction T_4	Lock Manager
lock_X(A)		grant_X(A, T_1);
read(A);		
A:=A-100.00;		
write(A);	lock_S(A);	
lock_X(B)		grant_X(B, T_1);
unlock(A);		grant_S(A, T_2);
htb!	read(A);	
	lock_S(B);	
read(B);		
B:= B+100.00;		
write(B);		
unlock(B)		grant_S(B, T_2);
	unlock(A);	
	read(B);	
	unlock(B);	
	print(A+B);	

Table 13: Schedule 2: A concurrent schedule for transactions T_3 , T_4 and the lock manager. T_4 has to wait for the locks it requests. The schedule is actually equivalent to the serial schedule (T_1, T_2) and T_2 prints the correct result $A + B = 2000.0$. In fact, every concurrent schedule for T_3 and T_4 is serialisable. The equivalent serial order is the order in which the transactions make their last lock request.

Deadlock

Lock-based protocols can lead to a deadlock.

	Transaction T_3	Transaction T_5	Lock Manager
htb!	lock_X(A)		grant_X(A, T_1);
	read(A);		
	A:=A-100.00;		
	write(A);	lock_S(B)	grant_S(B, T_2);
	lock_X(B)	read(B); lock_S(A);	

Table 14: Schedule 2: A deadlocked concurrent schedule. T_5 is waiting for T_3 to release its exclusive lock of A and T_3 is waiting for T_5 to release its shared lock on B. One of the transactions must be rolled back.

When a deadlock occurs, one of the deadlocked transactions must be rolled back. Its locks are released and other transactions can proceed.

Locking Protocol

A **locking protocol** is a set of rules which determine when a transaction may lock and unlock data items. A locking protocol ensures conflict serialisability if every concurrent schedule generated under the locking protocol is conflict serialisable.

3.1.1 The Two-Phase Locking Protocol

Every transaction requests locks and releases locks in two phases, a **growing phase** and a **shrinking phase**.

In the growing phase, a transaction may request locks but may not release any locks.

In the shrinking phase, a transaction may release locks but may not request any new locks.

Transactions T_3 and T_4 in the previous example conform to the two-phase locking protocol.

The two-phase locking protocol has the following properties:

- It ensures serialisability.
- The equivalent serial order is the order in which the transactions make their last request for a lock (i.e. the order in which their growing phase ends).
- It does not guarantee freedom from deadlock. Transactions T_3 and T_5 in the previous example conform to the two-phase locking protocol, yet end up deadlocked.
- Cascading rollbacks may occur.

Strict Two-Phase Locking

In addition to two-phase locking, all exclusive locks are held till the transaction commits.

Therefore data written by an uncommitted transaction is not read by other transactions till the transaction commits.

Therefore, if a transaction fails, no other transaction has to be rolled back because no other transaction has read any data written by the failed transaction.

The corresponding concurrent schedules are therefore cascadeless.

Rigorous Two-Phase Locking

In this variant of two-phase locking, all locks are held till a transaction commits.

The equivalent serial order is the order in which the transactions commit.

3.2 Timestamp-Based protocols

These protocols effectively determine the serialisability order of transactions in advance.

With each transaction T_i is associated a unique timestamp, $TS(T_i)$.

The timestamp is assigned before the transaction starts execution.

The timestamp may be taken from the system clock or from a counter which is incremented every time a new timestamp is assigned.

The timestamp order determines the serialisability order of transactions.

The system also assigns TWO timestamps to each data item. a write-timestamp and a read-timestamp (denoted WTS and RTS respectively).

WTS(A) is the timestamp of the last transaction that wrote A.

RTS(A) is the largest timestamp of any transactions which read A successfully.

3.2.1 The Timestamp-Ordering Protocol

The timestamp-ordering protocol ensures that read and write operations are executed in timestamp order.

The rules of this protocol are as follows:

1. When T_i wants to read(A)
 - If $TS(T_i) < WTS(A)$, A has already been written by a later transaction
 T_i is rolled back
 - If $TS(T_i) \geq WTS(A)$, T_i is allowed to read A and $RTS(A)$ is updated to the larger of $RTS(A)$ and $TS(T_i)$
2. When T_i wants to write(A)
 - If $TS(T_i) < RTS(A)$, A has been read by a later transaction
 T_i is rolled back
 - If $TS(T_i) < WTS(A)$, A has been written by a later transaction.
 T_i is rolled back
 - If $TS(T_i) \geq RTS(A)$ and $TS(T_i) \geq WTS(A)$, T_i is allowed to write A and $WTS(A)$ is updated to $TS(T_i)$.

When a transaction is rolled back, it is restarted with a new timestamp.

The timestamp-ordering protocol guarantees conflict serialisability because conflicting operations are executed in timestamp order.

It is free from deadlocks because no transaction ever has to wait for a data item.

Long transactions may be **starved** if they are repeatedly rolled back because of conflicting short transactions.