Data Modelling and Databases - Chapter 19 (Book)

- Author: Ruben Schenk
- Date: 27.05.2021
- Contact: ruben.schenk@inf.ethz.ch

19. More About Transaction Management

19.1 Serializability and Recoverability

19.1.1 The Dirty-Data Problem

Recall that data is `dirty` if it has been written by a transaction that is not yet committed. The dirty data could appear either in the buffers, or on disk, or both. Either can cause trouble.

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$ **Denied**		
$r_1(B)$;			
Abort; $u_1(B)$;			
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50

Example: Let us consider the serializable schedule from Fig. 18.13, but suppose that after reading B, T_1 has to abort for some reason.
T_2 has now read data that does not represent a consistent state of the database. That is, T_2 read the value of A that T_1 changed, but read the value of B that existed prior to T_1's actions.

19.1.2 Cascading Rollback

As we see from the example above, if dirty data is available to transactions, then we sometimes have to perform a `cascading rollback`. That is, when a transaction T aborts, we must determine which transactions have read data written by T, abort them, and recursively abort any transactions that have read data written by an aborted transaction.

As we have noted, a timestamp-based scheduler with a commit bit prevents a transaction that may have read dirty data from proceeding, so there is no possibility of cascading rollback with such a scheduler.

19.1.3 Recoverable Schedules

For any of the logging methods we have discussed in Chapter 17 to allow recovery, the set of transactions that are regarded as committed after recovery must be consistent. That is, if a transaction T_1 is, after recovery, regarded as committed, and T_1 used a value written by T_2, then T_2 must also remain committed, after recovery. Thus, we define:

- A schedule is `recoverable` if each transaction commits only after each transaction from which it has read has committed.

19.1.4 Schedules That Avoid Cascading Rollback

Recoverable schedules sometimes require cascading rollback. To guarantee the absence of cascading rollback, we need a stronger condition than recoverability. We say that:

- A schedule `avoids cascading rollback` (or is an `ACR schedule`) if transactions may read only values written by committed transactions.

Put another way, an ACR schedule forbids the reading of dirty data. As for recoverable schedules, we assume that "committed" means that the log's commit record has reached disk.

19.1.5 Managing Rollbacks Using Locking

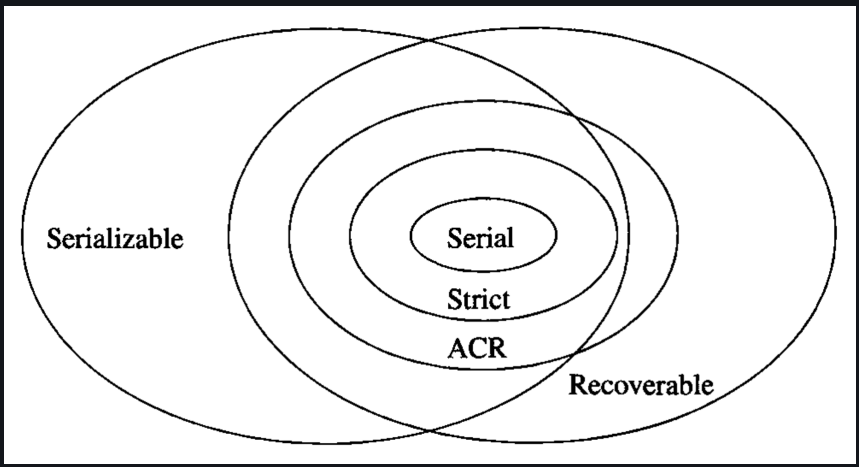
In the common case that the scheduler is lock-based, there is a simple and commonly used way to guarantee that there are no cascading rollbacks:

- `Strict Locking:` A transaction must not release any exclusive locks until the transaction has either committed or aborted, and the commit or abort log record has been flushed to disk.

A schedule of transactions that follow the strict-locking rule is called a `strict schedule. Two important properties of these schedules are:

1. *Every strict schedule is ACR.*
2. *Every strict schedule is serializable.*

With these observations, we can now picture the relationships among the different kinds of schedules we have seen so far. The containments are suggested in Fig. 19.3 below:



19.1.6 Group Commit

Under some circumstances, we can avoid reading dirty data even if we do not flush every commit record on the log to disk immediately. As long as we flush log records in the order that they are written, we can release locks as soon as the commit record is written to the log in a buffer.

This policy, often called `group commit`, is:

- Do not release locks until the transaction finished, and the commit log record at least appear in a buffer.
- Flush log blocks in the order that they were created.

Group commit, like the policy of requiring "recoverable schedules" as discussed in Section 19.1.3, guarantees that there is never a read of dirty data.

19.1.7 Logical Logging

Left out.

19.1.8 Recovery From Logical Logs

Left out.

19.2 Deadlocks

A `deadlock` occurs when each of several transactions is waiting for a resource held by one of the others, and none can make progress.

19.2.1 Deadlock Detection by Timeout

When a deadlock exists, it is generally impossible to repair the situation so that all transactions involved can proceed. Thus, at least one of the transactions will have to be aborted and restarted.
The simplest way to detect and resolve deadlocks is with a `timeout`. Put a limit on how long a transaction may be active, and if a transaction exceeds this time, roll it back.

19.2.2 The Waits-For Graph

Deadlocks that are caused by transactions waiting for locks held by another can be detected by a `waits-for graph`, indicating which transactions are waiting for locks held by another transaction. This graph can be used either to detect deadlocks after they have formed or to prevent deadlocks from ever forming.

If there are no cycles in the waits-for graph, then each transaction can complete eventually.
However, if there is a cycle, then no transaction in the cycle can ever make progress, so there is a deadlock.

19.2.3 Deadlock Prevention by Ordering Elements

Left out.

19.2.4 Detecting Deadlocks by Timestamps

Left out.

19.2.5 Comparison of Deadlock-Management Methods

Left out.

19.3 Long-Duration Transactions

Left out.

19.4 Summary of Chapter 19

Dirty Data

Data that has been written, either into main-memory buffers or on disk, by a transaction that has not yet committed is called `dirty`.

Cascading Rollback

A combination of logging and concurrency control that allows a transaction to read dirty data may have to `roll back` transactions that read such data from a transaction that alter aborts.

Strict Locking

The `strict locking` policy requires transactions to hold their locks until not only have they committed, but the commit record on the log has been flushed to disk.

Group Commit

We can relax the strict-locking condition that requires commit record to reach disk if we assure that log records are written to disk in the order that they are written.

Restoring Database State After an Abort

If a transaction aborts but has written values to buffers, then we can restore old values either from the log or from the disk copy of the database.

Logical Logging

For large database elements such as fisk blocks, it saves much space if we record old and new values of the log incrementally, that is, by indicating only the changes.

Deadlocks

`Deadlocks` occur when each of a set of transactions is waiting for a resource, such as a lock, currently held by another transaction in the set.

Waits-For Graphs

Create a node for each waiting transaction, with an arc to the transaction it is waiting for. The existence of a deadlock is the same as the existence of one or more cycles in the waits-for graph.