Databases - Transactions

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

ACID

ACID is the one acronym universally associated with Databases.

- A tomicity
- C onsistency
- I solation
- D urability

Videos

You may find Jennifer Widom's videos on transactions useful:

```
https://www.youtube.com/watch?v=-NPyRXCysW0
https://www.youtube.com/watch?v=usgUgO8xNDY
https://www.youtube.com/watch?v=zz-Xbqp0g0A
```

Robustness

One of the most important properties of a modern DBMS is that it is *robust* under both normal and unusual operating conditions.

- Normal conditions include multiple users concurrently accessing the database.
- Unusual conditions include computer crashes, connection failures and power outages.

It is important for a DBA to understand the role that *transactions* play in this robustness.

Interleaving

Most databases are being used by more than one client simultaneously.

Suppose that client A tries to run queries S_1, S_2, S_3 and that at about the same time, client B tries to run queries T_1, T_2, T_3.

In order to keep the system responsive for all users, the system will *interleave* the statements from A with the statements from B. So the system might actually run:

$$S_1, T_1, T_2, S_2, S_3, T_3$$

in that order.

What if A and B are trying to work with the *same table* and some of A's statements are altering things that B needs?

System Failure

The “canonical example” of an application where correct treatment of transactions is critical is transferring money in a bank.

For example, suppose that a user at an ATM transfers money:

```
UPDATE Accounts
SET balance = balance - 500
WHERE id = 1;

UPDATE Accounts
SET balance = balance + 500
WHERE id = 2;
```

Suppose the system crashes after the first statement, but before the second?

A database must be able to recover to a consistent state when the system comes back online.

Transactions

A *transaction* is defined to be any *one execution* of a user program.

In this context, a "user program" consists of a number of statements that *read* and *write* database objects (i.e. tables, values etc), before finally *committing* at which point any changes to the state of the DB are made permanent (i.e. written to disk).

A transaction then is a sequence of statements that must be treated on an "all-or-nothing" basis — either all the statements must finish without interference from other users, or none.

Atomicity

The word *atomic* is used in a number of contexts to denote *indivisible*.

In a DB context, transactions are *atomic* if the system ensures that they cannot be “half-done” — in other words, the user is guaranteed that either the entire transaction completes or it fails and has no effect on the database.

The bank transfer example above is one application where users rely on the atomicity of transactions.

Consistency

Transactions must preserve the *consistency* of the database.

More precisely, if the database is in a consistent state, and a transaction is executed to completion on its own (i.e. with no concurrently executing transactions) then the state of the database after the transaction should also be consistent.

This is basically a fancy way of saying that the user's programs should be correct. Transaction consistency is therefore the responsibility of the *user* not the DBMS.

Isolation

Isolation means that the user of the DB should be able to execute a transaction without regard for concurrently executing transactions.

In other words, the user's actions should be *isolated* from the actions of other users — at least for the duration of the transaction.

Durability

Durability means that once the user is informed of the successful completion of a transaction, then its effects on the database are persistent.

Thus the user should be shielded from any possible problems (eg system crashes) that might occur after being notified that the transaction has successfully completed.

Atomicity and Durability

Ensuring *atomicity* requires the DBMS to be able to *undo* the effect of earlier statements if the entire transaction is aborted, either by the DBMS itself (if a later statement fails) or for some external reason (system crash, power cut etc).

The basic mechanism used for this is that the DBMS maintains a *log* of all changes to the database. Every action that causes a change to the state of the database is *first* recorded in the logfile, which is then saved to disk. Finally the new state of the database is written to disk.

Write Ahead Log

The property that changes are logged before they are actually made on disk is called *write ahead log*.

If a transaction is aborted, then the DBMS can consult the log in order to determine which actions need to be undone in order to restore the database to its initial state.

In the case of a system crash, the *recovery manager* uses the log to determine whether there are any completed transactions that still need to be written to disk.

Complete details of the logging process and the recovery manager are complicated and require a detailed understanding of the physical aspects of computers

Interleaving

There would be no problem with *isolation* if the DBMS were able to simply run each transaction to completion at a time before starting the next one.

However in practice, it is vital to *interleave* the actions of transactions in order for the system to be usable in practice.

T1	**Time**	**T2**
Action 1	│	
Action 2	│	
	│	Action 1
	│	Action 2
Action 3	│	
	│	Action 3
End	↓	End

Motivation for Interleaving

Interleaving transactions (properly) allows multiple users of the database to access it at the same time.

While one transaction is performing an I/O task, another one can perform a CPU-intensive task thus maximising the *throughput* of the system.

Strict serial execution of transactions would be impractical because large numbers of short transactions would become “queued up” behind a long running transaction waiting for it to finish.

Thus managing a collection of interleaved transactions is a fundamental task for a DBMS.

Notation

We use the notation $R(O)$ and $W(O)$ to indicate the actions of *reading* a database object O and *writing* a database object O.

Then a transaction can be considered to be a sequence of reading and writing actions ending when the transaction *commits*.

T_1	Time	T_2
$R(A)$	↓	
$W(A)$		
		$R(A)$
		$W(A)$
		Commit
$R(B)$		
$W(B)$		
Commit		

Interleaving Anomalies

There are a variety of *anomalies* that can arise from an unfortunate choice of *schedule* for interleaved transactions.

Each of these anomalies could leave the database in an inconsistent state that could not arise if the two transactions were not interleaved.

- Dirty Reads
- Nonrepeatable Reads
- Phantoms

Dirty Reads

A *dirty read* occurs when one transaction reads a database value that has been altered by a transaction that has not yet committed.

Two major problems can arise from dirty reads

- The database may be in a temporarily inconsistent state due to the partially completed transaction.
- The partially completed transaction may subsequently be aborted restoring the value to its original state.

Dirty Read Example

Suppose that T_1 transfers \$100 from account A to account B, while T_2 adds 5% interest to each account, and the following schedule is used.

T_1	Time	T_2
$R(A)$		
$W(A)$		
		$R(A)$
		$W(A)$
		$R(B)$
		$W(B)$
		Commit
$R(B)$		
$W(B)$		
Commit	$\downarrow$	

Dirty Read Example cont.

If A and B have a \$1000 balance initially then this schedule would proceed as follows:

- T_1 deducts \$100 from A so balance is \$900.
- T_2 adds 5% interest to A so balance is \$945.
- T_2 adds 5% interest to B so balance is \$1050.
- T_1 adds \$100 to B so balance is \$1150.

Neither of the two possible serial schedules (i.e. T_1 first, then T_2 or vice versa) would give these values, and in fact \$5 interest has been lost.

The fundamental problem is that T_1 put the DB into an inconsistent state, and T_2 used the inconsistent values before T_1 could restore the DB.

Unrepeatable Reads

An unrepeatable read is essentially the dirty-read problem in reverse order in that a value gets changed by another transaction *after* it has been read, rather than before.

In this situation, transaction T_1 reads a value which is then changed by T_2. If T_1 subsequently re-reads the value then it gets a different value, even though it hasn't changed it.

This violates the *isolation property* because transaction T_1 should be able to complete as though it is the only transaction currently executing.

Phantoms

A *phantom* is a variant of the unrepeatable read problem that occurs when one transaction performs a `SELECT` statement with some selection criteria, and then subsequently another transaction inserts a new row.

If the first transaction now uses the same criteria again for a subsequent `UPDATE` statement, then a new row will suddenly appear, known as a *phantom* row.

Schedules

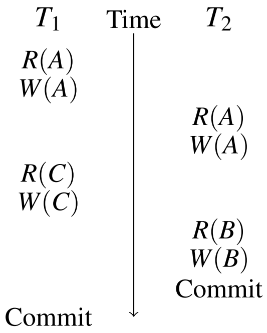
A schedule (of interleaved statements) is called *serializable* if its effect on any consistent database instance is equivalent to running the transactions in *some* serial order.

A schedule is called *recoverable* if a transaction T_2 that reads values changed by T_1 only commits *after* T_1 commits.

The job of the DBMS is to ensure that the only allowed schedules are *serializable* and *recoverable*.

Serializable . . .

This schedule is *serializable* because if both transactions commit as shown, then the effect is the same as running T_1 and then T_2.



T_1	Time	T_2
$R(A)$		
$W(A)$		
		$R(A)$
		$W(A)$
$R(C)$		
$W(C)$		
		$R(B)$
		$W(B)$
		Commit
Commit	$\downarrow$	

Locking

The main way in which a DBMS ensures that only serializable, recoverable schedules are allowed is through *locking protocols*.

A *lock* is a flag, or indicator, that can be attached to a database object indicating that it is in use by a transaction; a second transaction wishing to use the same DB object may have to wait until the first transaction has finished it.

A locking protocol is a set of rules that determine what types of lock to use in particular situations.

Strict Two-Phase Locking

The most widely used locking protocol is *Strict Two-Phase Locking* (Strict 2PL) which uses two rules.

- A transaction that wishes to *read* an object first requests a *shared lock* on that object, while a transaction that wishes to *modify* an object first requests an *exclusive lock* on that object.
- All locks held by a transaction are released when the transaction completes (commits or aborts).

InnoDB

The InnoDB database engine, which currently (in MySQL 5.7) is the default one that is used if the user does not specify otherwise is an ACID-compliant engine.

A transaction can be initiated by a user with the statement

```
START TRANSACTION;
```

All statements after that will be deemed to form part of the same transaction until one of the statements

```
COMMIT;
```

or

```
ROLLBACK;
```

occurs.

Example

Suppose students are forming project *groups* and being entered into the following database table.

```
CREATE TABLE groups(
  id INT,
  sNum INT,
  UNIQUE(sNum));
```

The `UNIQUE` keyword will ensure that no student accidentally ends up allocated to more than one group.

Initial data

Suppose that the first group has been entered:

```
SELECT * FROM groups;
+------+------+
| id   | sNum |
+------+------+
|    1 | 1537 |
|    1 | 1433 |
+------+------+
```

and that the second proposed group comprises students 1010 and 1537; of course 1537 is already in a group and so adding this group should fail.

Use a transaction

```
START TRANSACTION;
INSERT INTO groups VALUES(2,1010);
INSERT INTO groups VALUES(2,1537);
ERROR 1062 (23000): Duplicate entry ’1537’ for key 1
```

The first student was correctly added, but the second violates the key constraint — now the entire transaction can be aborted by issuing the `ROLLBACK` statement.

```
ROLLBACK;
```

```
SELECT * FROM groups;
+------+------+
| id   | sNum |
+------+------+
|    1 | 1537 |
|    1 | 1433 |
+------+------+
```

Rolling Back

In addition to explicitly issuing a `ROLLBACK` statement, this will happen *automatically* if the client becomes disconnected before the transaction is committed. Therefore critical transactions will never get left half-done.

There is also the facility to *partially* roll-back a transaction to an intermediate “checkpoint” that is declared inside a transaction.

```
SAVEPOINT doneOne;
```

If something goes wrong in the next few statements, the user can

```
ROLLBACK TO SAVEPOINT doneOne;
```

rather than rolling back the entire transaction.

Isolation Levels

MySQL permits the user to choose how “isolated” they wish each transaction to be by choosing between

- `READ UNCOMMITTED`
- `READ COMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`

The user can set the isolation level on a per-session or even per-transaction basis, using statements such as:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
```

Isolation levels

The `READ UNCOMMITTED` and `READ COMMITTED` isolation levels determine when a transaction will see a value that has been changed by another transaction — either immediately or only after the other transaction has committed.

The default isolation level of `REPEATABLE READ` guarantees that a transaction will get the same results each time it issues the same `SELECT` statement. (MySQL also guarantees that `REPEATABLE READ` prevents phantoms.)

The top level of isolation, `SERIALIZABLE` is essentially like `REPEATABLE READ` with minor differences to exactly which statements the guarantee applies to.

Isolation Level Summary

This table summarizes the anomalies that can or cannot arise at the different isolation levels.

Level	Dirty Read	Unrepeatable Read	Phantom
`READ UNCOMMITTED`	Yes	Yes	Yes
`READ COMMITTED`	No	Yes	Yes
`REPEATABLE READ`	No	No	No[1]
`SERIALIZABLE`	No	No	No

[1]MySQL-specific