3F6 - Software Engineering and Design

Handout 14

Database Systems I
With Markup

Ed Rosten

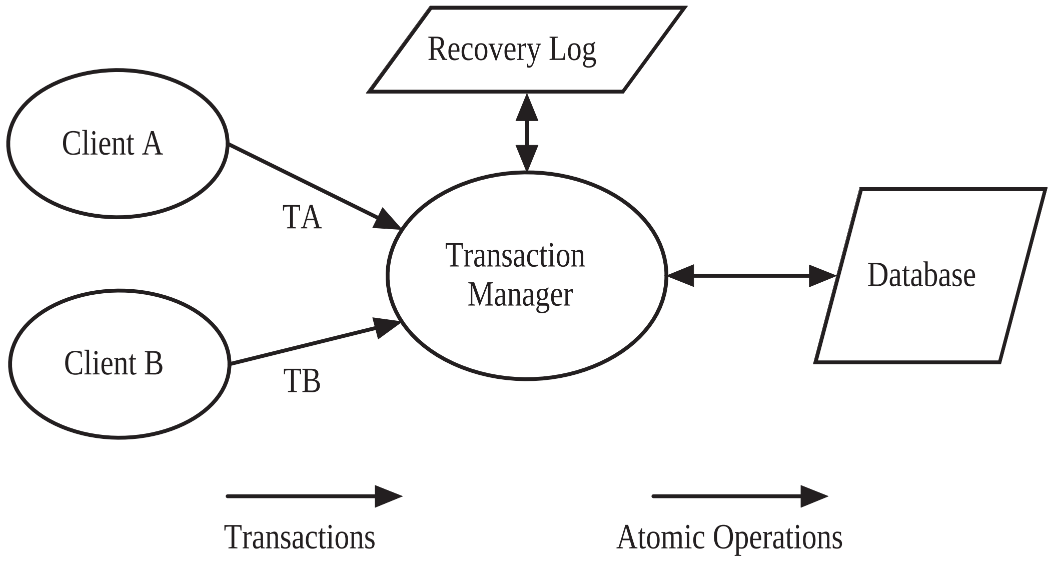
Contents

Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`.

Transaction Processing

Databases are a common component of many distributed systems. They store records for a large number of distinct entities and will typically support a small set of operations to access and manipulate those entities. These operations can be assumed to be *atomic* i.e. they cannot be interrupted.

External clients execute *transactions* which are sequences of operations applied to one or more database entities designed to achieve a single logical affect.



The *transaction manager* ensures that transactions appear atomic to clients. Client receives an acknowledgement of every successful transaction.

Example: Bank transfer

Each account is represented by a different database object, which guarantees that each operation is atomic

```
class account {
   // link to required account records
   DBaseAccessInfo dbinfo;
public:
   // Constructor - open an account
   account(string account_name);

   // Atomic operations
   void debit(float amount);
   void credit(float amount);
   float read_balance();
};
```

A typical transaction will be

```
void transfer(account& A, account& B, float amount)
{

   float balance = A.read_balance();
   if (balance >= amount) {
      A.debit(amount);
      B.credit(amount);
   }

}
```

A key issue is what happens if there is a failure part-way through the transaction.

System Crash

What happens if the system crashes in the middle of a transfer?

```
void transfer(account& A,
              account& B,
              float amount) {
   float balance;
   balance = A.read_balance();
   if(balance >= amount) {
      A.debit(amount);
      <------------------------------CRASH!
```

Account A will have had its money debited, but it will never appear in account B. (Good for the bank, not so good for B!)

The transaction manager (or any *transaction processing system*) must have a means of recovering from errors, and always leaving the system in a valid state.

The ACID Properties of Transactions

A transaction my fail in many different ways (e.g. two clients try to access the same entity at the same time, temporary network failure, software fault, disk crash, etc).

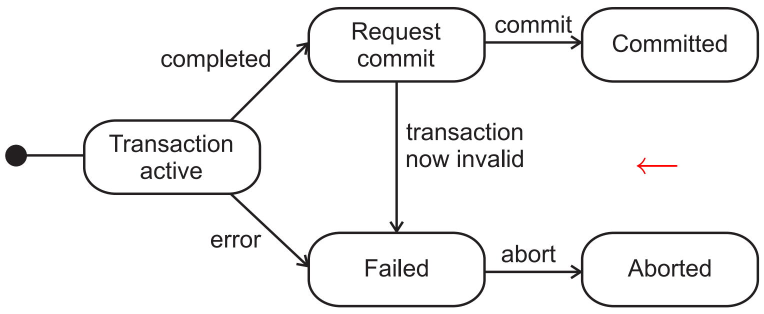
The transaction processor tries to ensure that transactions have the following properties:

- **Atomicity**
  Either all or none of the transaction's operations are performed.
- **Consistency**
  Transactions transform the system from one consistent state to another.
- **Isolation**
  An incomplete transaction cannot reveal its result to other transactions before it is complete.
- **Durability**
  Once the transaction is committed, the system must guarantee that the results of its operations will persist, even if there are subsequent system failures.

Recovery

In order to maintain the ACID properties, a transaction processor must be able to recover from errors by restoring the system to a consistent state.

To achieve this, transactions are modelled on the following state machine:



← Transaction processor might invalidate this transaction (see last slide)

Example: the transfer transaction

```
void transfer(account& A, account& B, float amount)
{
   try {
      int id = BeginTransaction();  // Record transaction start
      float balance = A.read_balance();
      if (balance >= amount) {
         A.debit(amount);
         B.credit(amount);
      }

      Commit(id);                  // success so commit

   }
   catch (...){

      Abort(id);                   // failure so undo

   }
}
```

Recovery Mechanisms

Recovery relies on two sources of information being stored in *safe, persistent storage* (e.g. a RAID array):

- A log of every database operation
- Frequent snapshots which record the currently active transactions (called *checkpoints*).

Recovery involves re-doing any transactions which had been committed since the last checkpoint, and undoing any uncommitted ones.

In the case of catastrophic failures, the database is restored from backup tape and all logged transactions are re-done from the point of failure. With modern RAID array technology and protected power supplies, this type of failure is rare.

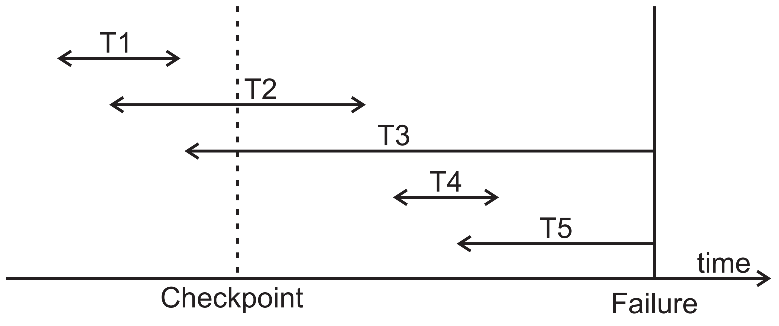
Log Files

The log file is the core of the recovery process. Each database action is added to the end of the log file immediately *before* it occurs (otherwise a crash could result in a committed transaction not being recovered). A typical log format is as in this example:

```
<checkpoint>          - A snapshot was taken here
<T1 start>            - Transaction 1 has started
<T1, A, 1000, 950>    - The value of object A was
<T1, B, 2000, 2050>     changed from 1000 to 950
<T2, start>
<T1, commit>          - Transaction 1 committed
<T2, C, 700, 600>
<T2, abort>           - Transaction 2 aborted
```

The presence of old and new values in the object lines `<T, A, old, new>` allows the log file to be used to UNDO transactions when an abort occurs (as happens to T2 above). In the case of a rebuild, the log file is also used to REDO all the operations since the reloaded `<checkpoint>`.

Recovery Algorithm



In the example above, the snapshot taken at the checkpoint would record that T2 and T3 are active.

At the point of failure, T2 and T4 had been committed, so they need REDOing. T3 and T5 had started, but were incomplete, so the best we can do is to UNDO them to ensure a consistent database state.

An algorithm for working out which to UNDO and REDO is as follows:

1. Add all transactions active at the checkpoint to `undo_list`. The `redo_list` is initially empty.
2. Work forwards from the checkpoint:
   - If you find a `<Ti start>`, add that transaction to `undo_list`.
   - If you find a `<Ti end>`, move that transaction to `redo_list`.
3. UNDO transactions on `undo_list`, working backwards from the end of the log.
4. REDO transactions on `redo_list`, working forwards from the checkpoint.

Concurrency

In practice, a database transaction processor will be receiving a stream of transaction requests, and will need to execute transactions in parallel in order to provide acceptable response times.

When two transactions reference the same account, uncontrolled interleaving of operations can produce an incorrect result. There are three classes of concurrency problem:

- **The uncommitted dependency problem**

Time	Transaction 1	Transaction 2
t_1	–	A.write()
t_2	A.read()	–
t_3	–	abort()

In this case, transaction 1 reads an updated account value, but transaction 2 aborts undoing the effect of the update. Transaction 1 is then left holding an incorrect account value.

Note: `A.read()` indicates any operation which reads a value from account A but does not change it (eg `A.read_balance()`), `A.write()` indicates any operation which changes account A (eg `A.credit()` or `A.debit()`) .

- **The lost update problem**

Time	Transaction 1	Transaction 2
t_1	A.read()	–
t_2	–	A.read()
t_3	A.write()	–
t_4	–	A.write()

In this case, the change made to account A at t_3 by transaction 1 is lost because it is overwritten at time t_4 by transaction 2.

- **The inconsistent analysis problem**

Time	Transaction 1	Transaction 2
t_1	A.read()	–
t_2	–	A.read()
t_3	–	A.write()
t_4	–	commit()

In this case, transaction 2 updates account A after transaction 1 has read its value. Hence, transaction 1 is left holding an incorrect value for account A.

Locks

The 3 problems noted above can be prevented by associating *locks* with each account.

When a transaction wishes to access an account it first *secures* a lock on that account, when it has finished it *releases* the lock. If a lock is already taken, the transaction must wait until it is released.

It is more efficient, however, to allow 2 levels of locking

- **Shared (S)** - allows read only access
- **Exclusive (X)** - allows read/write access

Several transactions can hold an S lock on an account, but only 1 transaction can hold an X lock.

When a transaction requires a lock, the following protocol applies

New request from transaction T	Currently allocated Locks			
	None	S (owned by T)	S (other)	X
S	ok	ok	ok	wait
X	ok	ok	wait	wait

Note that in practice locking is implicit. Any read access automatically acquires an S lock, any write access automatically acquires an X lock. All locks are held until the transaction commits or aborts.

Concurrency with Locking

With 2-stage locking as described above, the problem cases described early are avoided

- **The uncommitted dependency problem**

Time	Transaction 1	Transaction 2
t_1	–	A.write()
	–	A.Xlock()
	–	(A.X granted)
t_2	A.read()	–
	A.Slock()	–
	wait	–
	wait	–
t_3	wait	abort()
	wait	(A.X released)
t_4	(A.S granted)	
	–	

Now the read by transaction 1 is delayed until transaction 2 has aborted and the database state has been restored.

- **The lost update problem**

Time	Transaction 1	Transaction 2
t_1	A.read()	–
	A.Slock()	–
	(A.S granted)	
	–	–
t_2	–	A.read()
	–	A.Slock()
	–	(A.S granted)
	–	–
t_3	A.write()	–
	A.Xlock()	–
	wait	–
t_4	wait	A.write()
	wait	A.Xlock()
	wait	wait
	wait	wait
	wait	wait

In this case, the incorrect database state is avoided, but both transactions end up waiting for each other.

This is called *deadlock*.

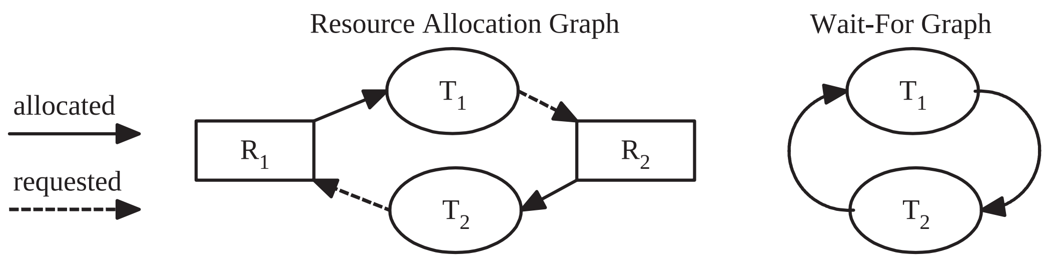
Deadlocks

Deadlock can occur whenever two transactions T_1 and T_2 require two resources R_1 and R_2 to proceed:

- T_1 holds R_1 and is waiting for R_2
- T_2 holds R_2 and is waiting for R_1

Deadlock detection

Deadlocks can be detected by building a *resource allocation graph* from which a *wait-for-graph* can be constructed. A cycle in a wait-for-graph indicates a deadlock.



Deadlock recovery

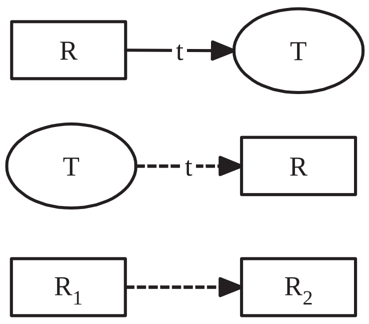
Select one of the deadlocked transactions, called the *victim* and abort it. Any changes made by the victim are undone and then all locks held by it are released.

Criteria for choosing a victim are

- avoid transactions which have been running a long time
- avoid transactions which have made many updates
- prefer transactions which are blocking access to many resources

Resource Allocation and Wait-For Graphs

A resource allocation graph shows the state of both acquired and requested resources along with the time at which each request or allocation was made.



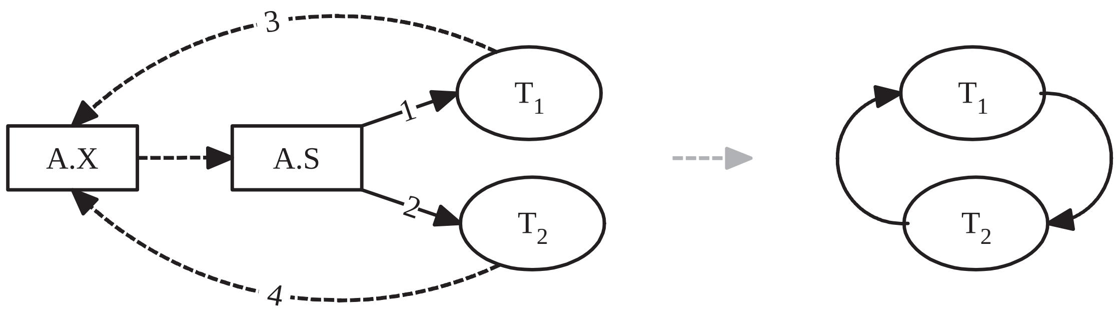
R has been allocated to T since time t

T has been waiting for R since time t

R_2 is blocking the allocation of R_1

Example: the lost update problem again

Time	Transaction 1	Transaction 2
t_1	A.read()	–
t_2	–	A.read()
t_3	A.write()	–
t_4	wait	A.write()



Lock-free Concurrency Control

Locking mechanisms carry a significant computational overhead. In very large systems with many accounts, the probability of two transactions conflicting might be very small. In such cases, locking is very inefficient. PESSIMISTIC

An alternative strategy is to allow uncontrolled access to accounts, and then simply abort any transactions which might have suffered a conflict.OPTIMISTIC

When the transaction starts, *shadow copies* of the accounts are taken and changes are only made to these copies. When the transaction asks to commit, the transaction undergoes *validation* to ensure that there could not have been any conflicts with transactions already accepted.

The validation stage looks at shadow accounts, and their timestamps, and compares them with the real accounts to ensure that:

- no other transactions have committed operations on the same accounts
- or, if they have, that those operations did not conflict with the operations this transaction performed.

If both these tests fail, the shadow is discarded and the transaction starts again. Otherwise, the real account is *updated* with the new information.

Which is best? *Lock* if conflicts are frequent; *Lockfree* otherwise

Summary

- A *transaction* is a collection of operations which perform a single logical function.
- Transactions should provide Atomicity, Consistency, Isolation and Durability (the ACID properties).
- A log file, and regular checkpoints (backups) provide durability in the event of a system crash, and allow transactions to be undone.
- Concurrent execution conflicts can lead to an inconsistent database state.
- Locking prevents conflicts but leads to deadlocks.
- Deadlocks can be detected by looking for cycles in allocation graphs.
- Locking can be inefficient, optimistic "lock nothing, and fix later" strategies may be more efficient in some applications.