COMP9311 Week 11 Lecture

Transaction Processing

Transactions, Concurrency, Recovery

DBMSs provide access to valuable information resources in an environment that is:

- *shared* - concurrent access by multiple users
- *unstable* - potential for hardware/software failure

Each user should see the system as:

- unshared - their work is not inadvertantly affected by others
- stable - the data survives in the face of system failures

Ultimate goal: data integrity is maintained at all times.

... Transactions, Concurrency, Recovery

Transaction processing

- techniques for managing "logical units of work" which may require multiple DB operations

Concurrency control

- techniques for ensuring that multiple concurrent transactions do not interfere with each other

Recovery mechanisms

- techniques to restore information to a consistent state, even after major hardware shutdowns/failures

Transactions

A *transaction* is

- an atomic "unit of work" in an application
- which may require multiple database changes

Transactions happen in a multi-user, unreliable environment.

To maintain integrity of data, transactions must be:

- *A*tomic - either fully completed or totaly rolled-back
- *C*onsistent - map DB between consistent states
- *I*solated - transactions do not interfere with each other
- *D*urable - persistent, restorable after system failures

Example Transaction

Bank funds transfer

- move *N* dollars from account *X* to account *Y*
- `Accounts(id,name,balance,heldAt, ...)`
- `Branches(id,name,address,assets, ...)`
- maintain `Branches.assets` as sum of balances via triggers
- transfer implemented by function which
    - has three parameters: amount, source acct, dest acct
    - checks validity of supplied accounts
    - checks sufficient available funds
    - returns a unique transaction ID on success

... Example Transaction


```
create or replace function
   transfer(N integer, X text, Y text) returns integer
declare
   xID integer; yID integer; avail integer;
begin
   select id,balance into xID,avail
   from   Accounts where name=X;
   if (xID is null) then
      raise exception 'Invalid source account %',X;
   end if;
   select id into yID
   from Accounts where name=Y;
   if (yID is null) then
      raise exception 'Invalid dest account %',Y;
   end if;
...
```

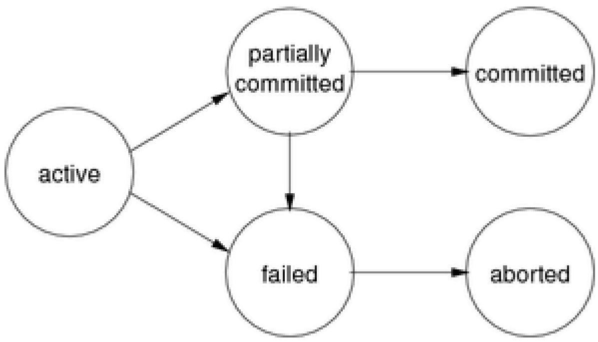
... Example Transaction


```
...
   if (avail < N) then
      raise exception 'Insufficient funds in %',X;
   end if;
   -- total funds in system = NNNN
   update Accounts set balance = balance-N
   where  id = xID;
   -- funds temporarily "lost" from system
   update Accounts set balance = balance+N
   where  id = yID;
   -- funds restored to system; total funds = NNNN
   return nextval('tx_id_seq');
end;
```

Transaction Concepts

A transaction must always terminate, either:

- successfully (`COMMIT`), with all changes preserved
- unsuccessfully (`ABORT`), with database unchanged



... Transaction Concepts

To describe transaction effects, we consider:

- `READ` - transfer data from disk to memory
- `WRITE` - transfer data from memory to disk
- `ABORT` - terminate transaction, unsuccessfully
- `COMMIT` - terminate transaction, successfully

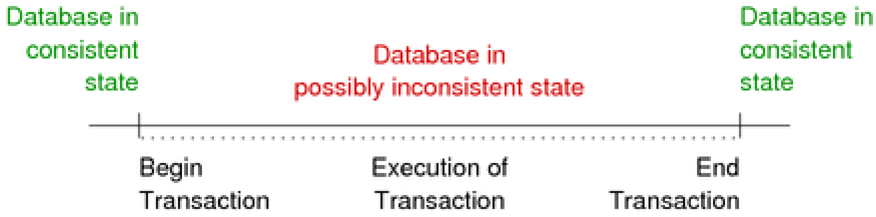
Normally abbreviated to `R(X)`, `W(X)`, `A`, `C`

`SELECT` produces `READ` operations on the database.
`INSERT` produces `WRITE` operations.
`UPDATE`, `DELETE` produce both `READ` + `WRITE` operations.

Transaction Consistency

Transactions typically have intermediate states that are inconsistent.

However, states *before* and *after* transaction must be consistent.



Reminder: "consistent" = satisfying all of the specified constraints

... Transaction Consistency

Transaction descriptions can be abstracted

- consider only *R*ead and *W*rite operations on shared data
- e.g. T1: R(X) W(X) R(Y) W(Y),   T2: R(X) R(Y) W(X) W(Y)

A *schedule* defines

- a specific execution of one or more transactions
- typically concurrent, with interleaved operations

Abribtrary interleaving of operations causes *anomalies*

- two consistency-preserving transactions
- produce a final state which is not consistent

Serial Schedules

Serial execution: `T1` then `T2` or `T2` then `T1`

```
T1: R(X) W(X) R(Y) W(Y)
T2:                     R(X) W(X)
```

or

```
T1:           R(X) W(X) R(Y) W(Y)
T2: R(X) W(X)
```

Serial execution guarantees a consistent final state if

- the initial state of the database is consistent
- `T1` and `T2` are consistency-preserving

Concurrent Schedules

Concurrent schedules interleave `T1`,`T2`,... operations

Some concurrent schedules are ok, e.g.

```
T1: R(X) W(X)      R(Y)      W(Y)
T2:           R(X)      W(X)
```

Other concurrent schedules cause anomalies, e.g.

```
T1: R(X)      W(X)      R(Y) W(Y)
T2:      R(X)      W(X)
```

Want the system to ensure that only valid schedules occur.

Serializability

Serializable schedule:

- concurrent schedule for $T_1 .. T_n$ with final state S
- S is also a final state of one of the possible serial schedules for $T_1 .. T_n$

Abstracting this needs a notion of *schedule equivalence*.

Two common formulations of *serializability*:

- *conflict serializibility* (read/write operations occur in the "right" order)
- *view serializibility* (read operations *see* the correct version of data)

Conflict Serializability

Consider two transactions T_1 and T_2 acting on data item X.

Possible orders for read/write operations by T_1 and T_2:

T_1 first	**T_2 first**	Equiv?
$R_1(X)\ R_2(X)$	$R_2(X)\ R_1(X)$	yes
$R_1(X)\ W_2(X)$	$W_2(X)\ R_1(X)$	no
$W_1(X)\ R_2(X)$	$R_2(X)\ W_1(X)$	no
$W_1(X)\ W_2(X)$	$W_2(X)\ W_1(X)$	no

If T_1 and T_2 act on different data items, result is always equivalent.

... Conflict Serializability

Two transactions have a potential *conflict* if

- they perform operations on the same data item
- at least one of the operations is a write operation

In such cases, the order of operations affects the result.

If no conflict, can swap order without affecting the result.

If we can transform a schedule

- by swapping the order of non-conflicting operations
- such that the result is a serial schedule

then we say that the schedule is *conflict serializible*.

... Conflict Serializability

Example: transform a concurrent schedule to serial schedule

```
T1: R(A) W(A)      R(B)      W(B)
T2:           R(A)      W(A)      R(B) W(B)
swap
T1: R(A) W(A) R(B)           W(B)
T2:                R(A) W(A)      R(B) W(B)
swap
T1: R(A) W(A) R(B)      W(B)
T2:                R(A)      W(A) R(B) W(B)
swap
T1: R(A) W(A) R(B) W(B)
T2:                     R(A) W(A) R(B) W(B)
```

... Conflict Serializability

Checking for conflict-serializability:

- show that ordering in concurrent schedule
- cannot be achieved in any serial schedule

Method for doing this:

- build a *precedence-graph*
- nodes represent transactions
- arcs represent order of action on shared data
- arc from $T_1 \rightarrow T_2$ means T_1 acts on X before T_2
- cycles indicate *not* conflict-serializable.

Concurrency Control

Concurrency Control

Serializability tests are useful theoretically ...

But don't provide a mechanism for organising schedules

- they can only be done "after the event"
- they are computationally very expensive *O(n!)*

What is required are methods that ...

- can be applied to each transaction individually
- guarantee that overall schedule is serializable

... Concurrency Control

Approaches to ensuring ACID transactions:

- lock-based

  Synchronise transaction execution via locks on some portion of the database.

- version-based

  Allow multiple consistent versions of the data to exist, and allow each transaction exclusive access to one version.

- timestamp-based

  Organise transaction execution in advance by assigning timestamps to operations.

- validation-based (optimistic concurrency control)

  Exploit typical execution-sequence properties of transactions to determine safety dynamically.

Lock-based Concurrency Control

Synchronise access to shared data items via following rules:

- before reading X, get shared (read) lock on X
- before writing X, get exclusive (write) lock on X
- an attempt to get a shared lock on X is blocked if another transaction already has exclusive lock on X
- an attempt to get an exclusive lock on X is blocked if another transaction has any kind of lock on X

These rules alone do not guarantee serializability.

Locking also introduces potential for deadlock and starvation.

Locking and Performance

Locking reduces concurrency ⇒ lower throughput.

Granularity of locking can impact performance:

+ lock a small item ⇒ more of database accessible

+ lock a small item ⇒ quick update ⇒ quick lock release

- lock small items ⇒ more locks ⇒ more lock management

Granularity levels: field, row (tuple), table, whole database

Many DBMSs support multiple lock-granularities.

Multi-version Concurrency Control

One approach to reducing the requirement for locks is to

- provide multiple (consistent) versions of the database
- give each transaction access to an "appropriate" version
  (i.e. a version that maintains the serializability of the transaction)

This approach is called *Multi-Version Concurrency Control*.

Differences between MVCC and standard locking models:

- writing never blocks reading (make new version of tuple)
- reading never blocks writing (read old version of tuple)

Concurrency Control in SQL

Transactions in SQL are specified by

- `BEGIN` ... start a transaction
- `COMMIT` ... successfully complete a transaction
- `ROLLBACK` ... undo changes made by transaction + abort

In PostgreSQL, other actions that cause rollback:

- **raise exception** during execution of a function
- returning null from a **before** trigger

... Concurrency Control in SQL

Concurrent access can be controlled via SQL:

- table-level locking: apply lock to entire table
- row-level locking: apply lock to just some rows

`LOCK TABLE` explicitly acquires lock on an entire table.

Other SQL commands implicitly acquire appropriate locks, e.g.

- `ALTER TABLE` acquires an exclusive lock on table
- `UPDATE`, `DELETE` acquire locks on affected rows

All locks are released at end of transaction (no explicit unlock)

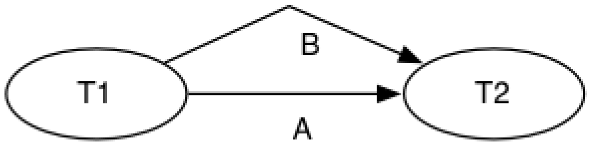
Examples

```
Schedule Q1 T1: R(A) W(A)      R(B)      W(B)
            T2:           R(A)      W(A)      R(B) W(B)
```

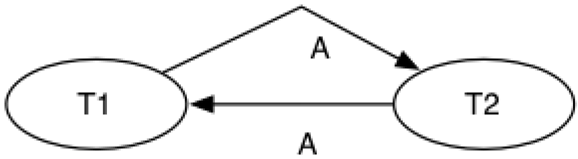
```
Schedule Q2 T1: R(A)      W(A)      R(B)      W(B)
            T2:      R(A)      W(A)      R(B)      W(B)


Schedule Q3 T3:      R(B)                W(B)
            T4: R(A)      W(A) R(B)                W(B)
            T5:                     R(A)      W(A)
```

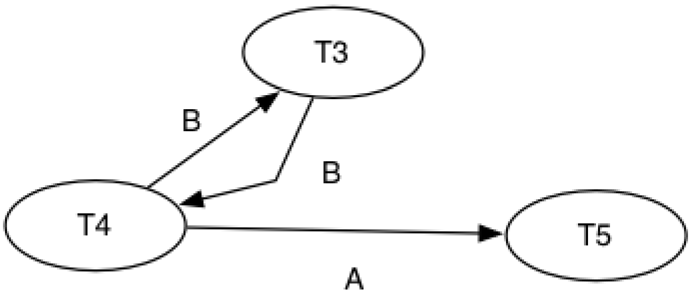
Q1



Q2



Q3



[Detailed Solutions]

Produced: 02 Oct 2017