The Database System

Architectural Overview

Important Aspects

- For permanence, data is on disk.
- To work on data, it must be in main memory.

  (But main memory is volatile!)
- Main memory is thousands of times faster than disk memory.

Primitive Operations

- *Read* a piece of data.
- *Write* a piece of data.

Within the database system, a transaction is just a sequence of reads and writes.

Transaction Management

Certain sets of actions on the database we want to occur together.

Such a set of actions we call a *transaction*.

Properties:

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

Goes hand-in-hand with *concurrency control*. The RDBMS should be able to handle 100,000's transactions a minute.

Some of these will be in conflict.

So a transaction may

- commit or
- abort (a.k.a. rollback)

Atomicity
All or Nothing

- insert into sailors values

  (53, 'dopey', 26, 7);

- insert into sailors values

  (53, 'dopey', 26, 7),

  (54, 'sleepy', 29, 3),

  (55, 'doc', 43, 10);

Consistency

```
create table WorldBank (
    acct# char(12) not null,
    name varchar(50) not null,
    balance decimal(15,2) not null,
    primary key (acct#),
    check (balance >= 0)
);

transfer (from, to, amount) {
    update WorldBank
        set balance = balance - :amount
        where acct# = :from;

    update WorldBank
        set balance = balance + :amount
        where acct# = :to;

    commit;
}
```

Isolation

$\mathbf{T}_1$: transfer(13, 21, 100.00);

$\mathbf{T}_2$: transfer(13, 34, 100.00);

$\mathbf{T}_1$	$\mathbf{T}_2$
$\mathbf{R}$(A)	
	$\mathbf{R}$(A)
$\mathbf{W}$(A)	
	$\mathbf{W}$(A)
$\mathbf{R}$(B)	
$\mathbf{W}$(B)	
	$\mathbf{R}$(C)
	$\mathbf{W}$(C)

How to ensure that X-acts do not "step on" one another?

How do we avoid inconsistencies that could arise due to concurrent X-acts?

Durability

Once a X-act *commits*, its effects on the database are permanent. (But not before then!)

- At what point can a X-act commit?
- Can other concurrent X-acts derail it?
- When will a X-act be *aborted*?

Note: The APP / X-act can decide to abort (rollback) itself at any time (up until a *commit*).

Durabilty and Crashes

What do we do if the DB crashes while some X-acts are still active?

- All uncommitted X-acts are effectivelt aborted on reboot.
- By durability, all committed X-acts must be reflected in the DB. (But they may not have been written to disk yet at the time of the crash!)

The RDBMS *logs* all actions so that it can *undo* the actions of all uncommitted transactions, and it can *redo* all committed transactions that did not make it to disk.

Serializability

```
inflate (percent) {
        update WorldBank
              set balance = balance * (1.0 + :percent)
        commit;
}
```

T_1: transfer(34, 13, 100.00);

T_2: inflate(13, 0.06);

We will accept any *equivalent schedule* such that the end effect is equivalent to *some serial schedule*.

Such a schedule is called *serializable*.

That X-acts can abort greatly complicates things!

What could go wrong if we just picked *any* schedule?

Anomalies

"Dirty Reads" / WR Conflicts

$\mathbf{T}_1$	$\mathbf{T}_2$
R(A)	
W(A)	
	R(A)
	W(A)
	commit
R(B)	
W(B)	
abort	

Anomalies

Unrepeatable Reads / RW Conflicts

$\mathbf{T}_1$	$\mathbf{T}_2$
R(A)	
	R(A)
	W(A)
	commit
R(A)	
W(A)	
commit	

Anomalies

Overwriting / WW Conflicts

$\mathbf{T}_1$	$\mathbf{T}_2$
W(A)	
	W(A)
	W(B)
	commit
W(B)	
abort	

Locks

How can we avoid such anomalies / conflicts? Locks!

Types of locks:

- **S**(A): Shared lock on A.

  Fine if X-act only needs to read A.
- **X**(A): Exclusive lock on A.

  Necessary if X-act needs to write A.

Granularity

What is A? What do we lock?

- table
- page
- row (tuple)
- cell (attribute in a tuple)
- index

Smaller granularity allows more concurrency, but is harder to manage.

Cascading Aborts

$\mathbf{T}_1$	$\mathbf{T}_2$	$\mathbf{T}_3$
$\mathbf{X}(\mathsf{A})$		
$\mathbf{R}(\mathsf{A})$		
$\mathbf{W}(\mathsf{A})$		
$\overline{\mathbf{X}}(\mathsf{A})$		
	$\mathbf{X}(\mathsf{A})$	
	$\mathbf{R}(\mathsf{A})$	
	$\mathbf{W}(\mathsf{A})$	
	$\overline{\mathbf{X}}(\mathsf{A})$	
		$\mathbf{X}(\mathsf{A})$
		$\mathbf{R}(\mathsf{A})$
		$\mathbf{W}(\mathsf{A})$
		$\overline{\mathbf{X}}(\mathsf{A})$
abort		

Purchase X-act

```
purchase (acct, merchant, state, amount) {
    select percent into :percent
        from TaxRate
        where state = :state
    update WorldBank
        set balance = balance - (:amount * (1.0 + :percent))
        where acct# = :acct;
    update WorldBank
        set balance = balance + :amount
        where acct# = :merchant;
    update WorldBank
        set balance = balance + (:amount * :percent)
        where acct# = (
                select acct#
                    from TaxRate
                    where state = :state
            )
    commit;
}
```

Deadlocks

A *deadlock* occurs when two (or more!) X-acts are mutually waiting on locks to be released that the others hold.

- Can deadlocks be avoided?
- Is it worth avoiding them?
- How do we resolve deadlocks (if they are "allowed" to occur)?

For that matter, can we avoid cascading aborts?

Two-phase Locking

- Each X-act must obtain a shared lock on each object before reading, and an exclusive lock on each obhect before writing.
- All locks are released at the completion of the X-act (*strict 2PL*).
- If any X-act holds an exclusive lock on A, no other X-act can have a shared or exclusive lock on A.

Strict 2PL

- allows only serializable schedules, and
- makes cascading aborts unnecessary.

It does not prevent deadlocks.

Transaction Modes (p. 539)

- Serializable
- Repeatable Read
- Read Committed
- Read Uncommitted

Serializable is just as advertised.

Repeatable Read avoids all the anomalies we discussed, except *phantoms*!

Read Committed releases a shared lock after reading. So unrepeatable read anomalies are possible.

Read Uncommitted obtains no locks! (Must be of type *read only*.)

Aborting

- If $\mathbf{T}_i$ is aborted, all its actions must be undone.

  If $\mathbf{T}_j$ read an object after $\mathbf{T}_i$ wrote it, $\mathbf{T}_j$ must be aborted too.

- *Cascading aborts* can be avoided by only releasing a X-act's locks at completion (commit / abort) time.

  So if $\mathbf{T}_i$ writes an object, $\mathbf{T}_j$ can only read this object *after* $\mathbf{T}_i$ is done.

- To *undo* actions, the RDBMS must maintain a *log* which records every write.

  The log mechanism is also used in *crash recovery.* All X-acts active at the time of the crash are aborted when the database system reboots.

The Log

Actions recorded in the log:

- $\mathbf{T}_i$ writes an object.
  - the old value
  - the new value

  The log record must go to disk *before* the changed page.
- $\mathbf{T}_i$ commit or $\mathbf{T}_i$ abort.

Log records are chained together by a X-act ID, so it is easy to *undo* a specific X-act.

The log is often duplexed and archived on stable storage for crash recovery.

All concurrency control (CC) activities—logging, locking, and deadlock control—are handled by the RDBMS transparently!

Crash Recovery (ARIES)

The three phases of the ARIES recovery algorithm:

- *Analysis:* Scan the log forward and find all X-acts that were active (committed, aborted, and continuing) since the last checkpoint.
- *Redo:* Redoes all writes (updates to dirty pages) in the buffer pool (as needed) to ensure all logged updates are carried out and (eventually) written to disk.
- *Undo:* Undoes the writes of all X-acts active at the crash, working backwards through the log.

Care must be taken to handle the case of a crash *during* the recovery itself!