COM106: Introduction to Databases

Database Management – Transaction Management

Database Management – Transaction Management

See Chapter 22 of Connolly & Beggs (6th ed)
(or similar chapters in other textbooks.)

What is a Transaction?

A **transaction** is a **logical unit of work**.

It may consist of a number of commands/operations grouped into a single unit. That is, a number of *related operations necessary to carry out some task* (retrieval and/or insertion and/or deletion and/or update)

For example:

- transfer of money between two bank accounts:

ACCOUNT(account_num, balance, customer)

Transfer of money from one account to another requires:

1. UPDATE query - to subtract money from balance of first account
2. UPDATE query - to add same amount of money to the second account

- order form submission which may involve insertion of records into the customer, order and ordered_part tables

In order to preserve database integrity the transaction must be **atomic** (all parts of transaction must happen or none at all)

Transaction Management

Consider the following ORDERS Schema:

CUSTOMER (cust_num, cust_name, cust_phone_num)

ORDER (order_num, order_date, del_date, cust_num)

ORDEREDPART (order_num, part_num, quantity)

PART (Part_num, p_name, unit_cost)

Order#: 1001	Order Date: 01/05/2015		Delivery Date: 06/05/2015
Cust Number: 21	Cust Name: J Smith		Phone Num: 90345122
Part Number	Part Name	Unit Cost	Quantity
101	Cog	£2.50	5
134	Block	£4.00	3

Adding the new order above requires:

1. INSERT the new customer (cust_num, cust_name, cust_phone_num) into CUSTOMER table (unless customer has already registered)
2. INSERT new order details (order_num, order_date, del_date) into ORDER table
3. Two INSERTS for each order line (order_num, part_num, quantity) into ORDEREDPART table

Transaction Management

Transaction support frees applications from having to deal with:

- inconsistencies from conflicts between concurrent transactions
- partial completion of transactions if system fails
- user-initiated undoing of transactions

ACID properties of database transactions

Atomicity: Transactions complete or fail as one unit

- Recovery System ensures atomicity

Consistency: Transaction must transform database from one consistent state to another.

- DBMS integrity constraint enforcement plus application error handling ensures consistency

Isolation: Transactions should not interfere with each other

- Concurrency control system ensures isolation

Durability: Results of transactions are not lost even on failure

- Recovery System ensures durability

Transactions - Examples

Consider a bank account table database: ACCOUNT (account_num, account_holder, balance)

A **transaction** transfers £100 from account A10 to account A25

Update transaction:

```
UPDATE ACCOUNT
SET balance = balance – 100
WHERE account_num = A10;
```

```
UPDATE ACCOUNT
SET balance = balance + 100
WHERE account_num = A25;
```

The individual updates cannot be done in parallel, so one transaction operation must complete before the other.

However, for the database to maintain **integrity**, the account A10 and A25 records cannot be available for other users to see until **both** operations complete correctly.

Now, suppose both Accounts started with £1000. After a successful **transaction**:

Account A10 should have £900 *(£1000 - £100)*

Account A25 should have £1100 *(£1000 + £100)*

Consider what would happen if the first update worked but the second failed:

Account A10 would have £900

BUT Account A25 would stay at £1000

Result: Database is incorrect

The system must consider these operations as a **single unit - Atomicity**

Transactions – Some Basics

A **transaction** transforms one **consistent** database state to another **consistent** state (without being consistent at all times during the transaction)

Transaction commands:

BEGIN TRANSACTION/END TRANSACTION:

COMMIT: Used at the end of a transaction to signify completion so that updates can be made permanent.

ROLLBACK: Used if any error is detected to undo any update operations.

Rollback (Undo) is achieved by consulting the **systems log** on which details of all updates ('before' and 'after' values of updated items) are recorded.

Transactions are managed by a **transaction manager** which fetches records for processing from the hard disk and returns updated records to the hard disk (permanent storage).

Updated records are first sent to a **database buffer** (part of main memory), then committed to disk to be made permanent.

A **transaction log** keeps transaction details:

- transaction details *(ID, operations)*
- 'before' image *(record(s) involved before update - required where update or delete has occurred)*
- transaction time
- 'after' image *(record(s) involved after update -required where update or append has occurred)*

The transaction log entry is always made permanent before the update is committed to disk (ensures correctness)

Transaction Log

Time	ID	Operation	Before Value	After Value
T1	1	BEGIN TRANSACTION		

T2	1	UPDATE ACCOUNT
		SET balance = balance – 100
		WHERE account_num = A10
			A10, 1000, ‘Smith’	A10, 900, ‘Smith’

T3	1	UPDATE ACCOUNT
		SET balance = balance + 100
		WHERE account_num = A25
			A25, 1000, ‘Jones’	A25, 1100, ‘Jones’

T4	1	END TRANSACTION		

The records held in a transaction log depend on the type of operation:

DELETE operations have an OLD value but NO NEW value

INSERT operations have NO OLD value but only a NEW value

UPDATE operations have an OLD value and a NEW value

RETRIEVAL operations do not require OLD or NEW record values to be stored since they do not change record values

The Log can then allow:

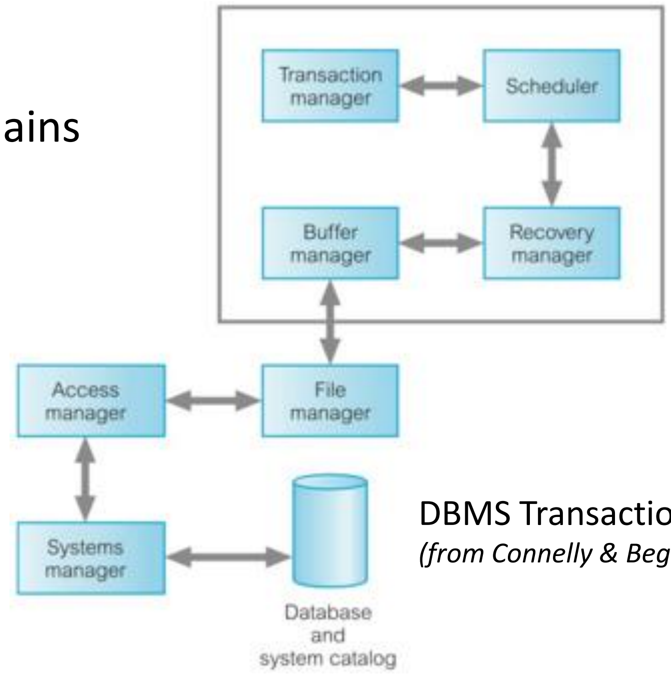
UNDO (ROLLBACK) by putting back the OLD record value

REDO by putting back any NEW values (if available) after ROLLBACK (i.e rerun the operation)

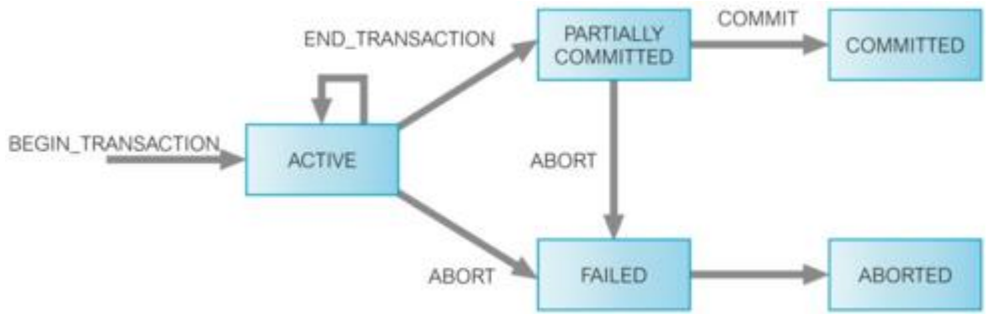
Transaction Management

To ensure that a database is reliable and remains in a consistent state, a DBMS must provide:

- Transaction Support
- Concurrency Control Services
- Recovery Services



DBMS Transaction Subsystem
(from Connelly & Begg (6th ed.) fig 22.3.)



State Transition Diagram for a Transaction
(from Connelly & Begg (6th ed.) fig 22.2.)

During processing, a transaction can be in one of a number of different **states**:

- Active
- Committed
- Aborted
- Partially Committed
- Failed

Recovery

Larger DBMSs (SQL Server, MySQL, ORACLE) provide automatic recovery mechanisms for transactions.

There are two broad categories of failures:

System Failure	which affects all current transactions but does not damage the stored database (e.g. power failure).
Media Failure	which affects the current transactions and damages the database (e.g. disk head crash).

System failure involves loss of main storage contents. The precise state of transactions in progress is unknown

- some transactions are completed and no action is required
- some transactions must be undone (rolled back) then redone
- some transactions must be redone

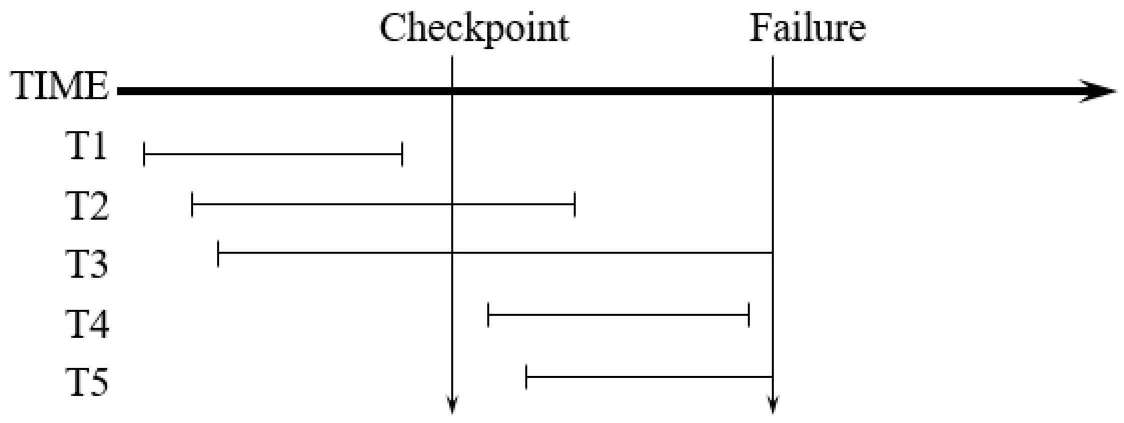
Recovery components

Transaction Log	details of all transactions with before and after values. On system failure indicates which transactions are complete, which need to be undone and/or redone.
Checkpoint	periodic storage of all transactions to disk and writing of details to **system log**
	Forces the contents of the database buffer and the transaction log to disk
	Makes an entry (checkpoint time) in transaction log of the database state and identifies all currently active transactions

Recovery – Transaction Classes

On a systems failure, the class (type) of transaction can be decided by running the log backwards from the time of failure to the time of the checkpoint.

Consider the diagram below:



When the system is restarted:

Transaction T1:	is complete and updates written to disk - no action necessary
Transactions T2 & T3:	were active over last checkpoint - must be first undone (rolled back) then redone
Transactions T4 & T5:	had not started at last checkpoint and thus no part is saved to disk - must be redone

After this has happened recovery is complete and new transactions can be accepted.

Recovery – Transaction Classes

To decide what to do with the transactions found while running the transaction log backwards:

- If a transaction has completed before the last checkpoint **(Class 1)**

  the updated records must have been written to disk at that checkpoint

  no action is required

- If a transaction is active over the last checkpoint **(Class 2)**

  the transaction is incomplete AND only some of the updates have been written to disk at the last checkpoint

  the transaction must be undone (using before images) THEN redone

- If a transaction started after the last checkpoint **(Class 3)**

  no updated records have been written to disk

  the transaction must be redone again

Recovery – Media Failure

It is necessary that regular **backup copies** of the database are taken at quiet periods.

Transaction Log is also required to show what transactions have occurred since last backup and before media failure

Recovery from Media Failure:

- Requires the database to be restored (reloaded) from a previous backup .
- Transaction Log shows transactions that must be redone (started after last backup but completed before media failure).
- System can then accept new transactions.

Concurrency in Databases

Larger DBMSs are multi-user (allow **concurrent transactions**)

Concurrent transactions are **interleaved** by operating system

- elements of the transaction (e.g. read, update) may be performed independently and during other transactions to allow each transaction an equal share of the processor and thus maximise throughput.
- high levels of interleaving increases throughput but also increases the chance of **inconsistencies** occurring.

Schedules and Serializability

A **schedule** of concurrent transactions is a set of (possibly interleaved) operations preserving the relative order in each individual transaction

If two transactions occur one after the other (with no interleaving) the schedule is **serial**

If two concurrent transactions are interleaved the schedule is **nonserial**

If a nonserial schedule is correct is it is **serializable** (produces the same result as you would get if the transactions are performed serially - i.e. as if they were run completely independently)

Concurrency - Schedules and Serializability

Example of a Serial Schedule

Transaction 1 and Transaction 2 are **serial** (they occur separately – one after the other).

This **guarantees correctness**, but ……. in an application processing many transactions it is an inefficient use of a fast processing resource

Transaction 1	Time	Transaction 2
Begin Transaction	t1	
Read X	t2	
Write X	t3	
Commit	t4	
	t5	Begin Transaction
	t6	Read Y
	t7	Write Y
	t8	Commit

Transaction 1	Time	Transaction 2
Begin Transaction	t1	
Read X	t2	
	t3	Begin Transaction
	t4	Read Y
Write X	t5	
Commit	t6	
	t7	Write Y
	t8	Commit

Example of an Interleaved Schedule

In order to use resources better, transactions are swapped in and out of the processor so that each is given processing time.

This increases performance but there is a **risk** that *two interleaved transactions operating on the same record may interfere with each others processing* and give rise to incorrect data results

Concurrency – The Lost Update Problem

An example of transaction interference

Consider two transactions (**T_A and T_B**) each involving an update of the **same record** (R)

- At time t1 - **T_A** reads the initial value of R.
- At time t2 - **T_B** also reads the same initial value of R
- At time t3 - **T_A** changes record R (then written to disk)
- At time t4 - **T_B** changes record R (then writes to disk ignorant of the change made by **T_A**)

Transaction A	Time	Transaction B
Read R	t1	
	t2	Read R
Update R	t3	
	t4	Update R

Since both transactions are updating the **same record** (R), the result is

T_A update is overwritten and lost.

An Example: Consider two transactions updating Stock Level (SL) for the same item of Stock – record R

T_A records sale of **3 items**

T_B records sale of **5 items**

Initial Stock Level is **10 items**

Transaction A	Time	Transaction B
Read R (SL = 10)	t1	
	t2	Read R (SL =10)
Update R(SL = 10-3 = 7)	t3	
	t4	Update R (SL = 10 - 5 = 5)

- At time t3 - **T_A** writes the Stock level as **7**
- At time t4 - **T_B** overwrites the previous Stock level as **5**
- The final Stock Level should have been **10 - 3 - 5 = 2**, so A's update is **lost**

Concurrency – Uncommitted Dependency

An **uncommitted dependency** arises when one transaction is allowed to retrieve or update a record that has been updated but is later **rolled back** (undone).

As in the previous example, consider two transactions updating Stock Level for the same record (R).

T_A records sale of **3 items** T_B records sale of **5 items**

Initial Stock Level is **10 items**

- At time t1 - Stock Level is **5**
- At time t3 - Stock Level is **2** (A has used Stock Level updated by B)
- But at time t4 - T_B is rolled back (undone) and Stock Level is reset to 10

The effect of T_A is thus lost (T_B is rerun)

Transaction A	Time	Transaction B
	t1	Update R
Read R	t2	
Update R	t3	
	t4	ROLLBACK

Concurrency Control Problem Summary

Lost Update Problem

When two transactions wish to update the same record simultaneously, both cannot be allowed to read the initial value before they perform updates otherwise, when the updates are written back to disk, the second update will overwrite the first, which will be lost

Uncommitted Dependency (aka - Dirty Read)

When two transaction are updating the same record the second transaction cannot be allowed to use the result of the first transaction until that result has finally been **committed** to disk.

Concurrency – Locking

Overall objective of concurrency control is to:

- prevent transaction interference (**enforce serialisability**)
- allow maximum possible concurrency (i.e. **maximise interleaving**)

To ensure that records will not change during a transaction, it acquires a **lock** (to lock other transactions out)

Locking Level (Granularity) - Locking can be applied at record, page, table or even database level.

- Locking more of the database at once reduces the concurrency that is allowed, but the easier it is to apply (and vice versa).

The **transaction manager** maintains a **lock table**:

- keeps information on which transactions have been granted permission to access which records
- when a transaction wishes to access a record it requests permission from the transaction manager, which checks the lock table
- if no other transaction has a lock, a lock is granted
- if another transaction has a lock permission is denied
- when a transaction ends the lock is released (i.e. message sent to transaction manager which amends lock table)

Concurrency – Locking Solution to Lost Update

Recall the Lost Update Problem - two transactions wish to update the same record simultaneously

Transaction A	Time	Transaction B
Read R	t1	
	t2	Read R
Update R	t3	
	t4	Update R

Now, consider a solution using **locking**:

Transaction A	Time	Transaction B
Read R (Lock granted)	t1	-
	t2	Read R (Lock denied)
Update R	t3	wait (Lock denied)
(Lock released)	t4	wait (Lock denied)
	t5	Read R (Lock granted)
	t6	Update R

Using locking, T_A completes before T_B is allowed to start

- no interference can occur
- effectively, T_A and T_B have been serialised

Note:- If T_A and T_B were reading different records (the most common situation)

- both would have been granted locks and no wait for locks would occur

Concurrency - Locking Solution to Uncommitted Dependency

Recall the uncommitted dependency Problem - two transactions wish to update the same record, but one transaction is **rolled back**

Transaction A	Time	Transaction B
	t1	Update R
Read R	t2	
Update R	t3	
	t4	ROLLBACK

Now, consider a solution using **locking**:

Transaction A	Time	Transaction B
	t1	Update R (Lock granted)
Read R (Lock denied)	t2	
wait (Lock denied)	t3	
wait (Lock denied)	t4	ROLLBACK
wait (Lock denied)	t5	Read R (Lock released)
Read R (Lock granted)	t6	
Update R	t7	

T_A never sees the record value updated by **T_B** since the original value of record R is restored *before* the **lock** is released by **T_B**

T_B will then rerun later after **T_A** successfully completes and releases its **lock** on R

Concurrency – Problem of Deadlock

Consider two transactions (T_A and T_B) updating records R and S

Transaction A	Time	Transaction B
Read R (Lock on R granted)	t1	
	t2	Read S (Lock on S granted)
Read S (Lock on S denied)	t3	
wait (Lock on S denied)	t4	Read R (Lock on R denied)
wait (Lock on S denied)	t5	wait (Lock on R denied)
wait (Lock on S denied	t6	wait (Lock on R denied)

Deadlock occurs at time t4 - unless the database intervenes no further processing can occur

Handling Deadlock

Once detected, **Victim Rollback** is the only means of **deadlock resolution**.

- A victim transaction is selected (normally on basis of transaction start time) and rolled back, releasing locks and allowing other transactions to proceed
- The rolled back transaction is then redone after the other has completed

Deadlock Detection (Note:- Deadlock **prevention** is possible, but is expensive and rarely used)

- May use deadlock detection based on an algorithm detecting **cycles** in a **wait-for graph**
- **Transaction Timeout** - A less sophisticated way is to define a maximum time on no transaction activity – when this is exceeded then assume deadlock and rollback a random victim (problem of **livelock**)

Concurrency – Two-Phase Locking Protocol

With a two-phase locking protocol transaction are divided into two phases:

- **Growing Phase:** locks acquired but none released
- **Shrinking Phase:** locks released but no more acquired

Two phase locking **guarantees serialisability** (correctness)

Problems:

Deadlock -	mutual wait for state with other transaction(s)
Livelock -	continually waiting for lock release while other newer transactions are granted locks ahead of them - requires priority system based on transaction start time

Commonly, commercial DBMSs use two-phase locking which guarantees serialisability at the expense of concurrency. This involves:

- Gaining **ALL** locks on all records required for a transaction **BEFORE** processing begins
- Running all transaction commands
- Releasing **ALL** locks together at the end of transaction