ACID:

Atomicity: All operations of transactions happen or none happen.

Consistency: Satisfies integrity constraints of transaction boundaries

Isolation: Result of execution of concurrent transactions is the same as if transactions were executed independently.

Durability: Effects of completed transactions serially become permanent surviving any subsequent failures.

Conceptual Evaluation Strategy:

1. Compute Cross Product of FROM
2. Discard tuples that fail WHERE
3. Delete attributes not in SELECT
4. If DISTINCT, eliminate duplicate rows

Query Tree:

- Corresponds to a relational algebra expression
- Input relations are leaf nodes
- Relational algebra as internal nodes
- Executes internal node operations

Relational Algebra:

PROJECTION $\pi_{<Attribute\ list>}(R)$: $(SELECT)$

SELECTION $\sigma_{<Selection\ condition>}(R)$: $(WHERE)$

Note:

$$R \bowtie_{condition} S = \sigma_{<Selection\ condition>}(R \times S):$$

NO INDEX + UNSORTED DATA: LINEAR SEARCH
NO INDEX + SORTED DATA: BINARY SEARCH

COST FUNCTIONS:

1. **TUPLE NESTED LOOP JOIN:**

$Cost = B_S + T_s \times B_R$

T_s = Number of tuples

$B_s = Number\ of\ Blocks\ in\ S = \frac{Number\ of\ Tuples}{How\ many\ Tuples\ fit\ in\ one\ block}$

2. **PAGE-ORIENTED NESTED LOOP JOIN:**

$$Cost = B_S + (B_s \times B_R)$$

3. **BLOCK NESTED LOOP JOIN:**

$$Cost = B_s + \left(\frac{B_s}{N_B - 2}\right) \times B_R$$

If foreign Key: Replace $B_s\ with\ B_R\ and\ B_R\ with\ \frac{B_s}{2}$

1. **PAGE-ORIENTED NESTED LOOP JOIN:**

$$Cost = B_R + (B_R \times \frac{B_s}{2})$$

2. **BLOCK NESTED LOOP JOIN:**

$$Cost = B_R + \left(\frac{B_R}{N_B - 2}\right) \times \frac{B_s}{2}$$

No. of Tuples R and S join produces = Max out of{No. of tuples in R or No. of tuples in S}

CONCURRENCY:

Anomalies:

1.Lost update problem: When two transactions send the same data and modify it independently. Last committed transaction overwrites the first transaction.

2.Dirty read problem: If transaction reads data that was modified by another transaction.

3.Unrepeatable Read: When data is sent twice within a transaction and between the reads, it is modified by a different transaction. The 2nd read returns a different value compared to the 1st.

Forms of 2PL locking:

1.BASIC:

GROWING PHASE: Can only acquire locks, can't release them

SHRINKING PHASE: Can only release and not acquire them

Basically: Growing = only locking
Shrinking = only releasing

2. Conservative 2PL: (Dead lock Free)

- Lock all items beforehand, no releasing/unlocking restrictions
- No operations done before acquiring all locks, so no growing phase.

3.Strict 2PL:

Growing phase is similar to normal 2PL. When acquiring all the locks in the first phase, the transaction continues to execute normally. But, strict-2L does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all of them at the same time.

Conditions for a Conflict:

- Belong to 2 different transactions
- Operate on the same data item
- One of them is a write operation

Order: 1. Read(n), 2. Write(w), 3. Abort(a), 4. Commit(c)

LOCK: Synchronisation mechanism

$R(x)\ where\ x\ is\ read-locked\ by\ T_i, R(x)\ is\ a\ shared\ lock$

$W(x)\ where\ x\ is\ write-locked\ by\ T_i.\ W(x)\ is\ an\ exclusive\ lock$

Example:

$S = T_1: W(x), T_2: R(x), T_1: R(Y). T_2: R(x), T_1\ commit, T_2\ commit$

1.Baise 2PL is allowed since no locks that have been dropped have been attempted to be acquired again by the same transaction.

2.Strict 2PL: T_1 releases a lock before it commits and that is not allowed, therefore S is not allowed under Strict 2PL.

3.Conservative 2PL: $T_2: R(x)$ can not happen because T_1 is still holding onto an exclusive lock on x: $T_1: W(x)$.

Shared: Multiple transactions can do read-locks
Exclusive: Only one can hold the lock on x

Locking on its own does not ensure serializability.

TIMEOUT MECHANISM:

1. Long Time: Fewer mistakes made by the scheduler but a deadlock may exist unnoticed for long periods
2. Short Time: Quick deadlock detection but more mistakes are possible (aborting transactions not involved in deadlock).

PROS: Reduces the chance of incorrectly detecting a deadlock

CONS: Increases the time it takes to detect an actual deadlock because of the fact that transactions are blocked due to deadlocks

WRITE-AHEAD LOGGING (WAL) PROTOCOL:

Ensures that no modifications to a database page will be flushed to disk until the associated transaction log record with that modification are written to disk first. We do this to maintain the ACID properties of a transaction.

Undo: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM), its BFIM must be written to the log and the log must be saved on a stable store.

Redo: Before a transaction is committed, all it's AFIM must be written to the log and the log must be saved on a stable store.

Heuristics:

1. Selection operation performed as early as possible to reduce number of tuples

2. Projection operation performed as early as possible to reduce num of attributes. This can be done by moving selection and projection operators as far down the tree as possible

3. Selection and joining operations that are most restrictive are executed first. Done by rearranging leaf nodes and readjusting the tree accordingly.

Consider the following two transaction:

$$T_1: read(A);\ read(B); if\ A = 0\ then\ B = B + 1;\ write(B)$$

$$T_2: read(B);\ read(A); if\ B = 0\ then\ A = A + 1;\ write(A)$$

Write out all the lock and unlock instructions to transaction T1, T2, so that they obey the two-phase locking protocol:

T_1	T_2
$T_1: read\_lock(A)$	$T_2: read\_lock(B)$
$read(A)$	$read(B)$
$read\_lock(B)$	$read\_lock(A)$
$read(B)$	$read(A)$
$if\ A = 0\ then\ B = B + 1$	$if\ B = 0\ then\ A = A + 1$
$write\_lock(B)$	$write\_lock(A)$
$write(B)$	$write(A)$
$unlock(A)$	$unlock(B)$
$unlock(B)$	$unlock(A)$

Q24: Can the execution of these transactions result in a deadlock?

Yes, because $T_1: read\_lock(A)$ needs to wait for T_2 to release $write\_lock(A)$ and $T_2: read\_lock(B)$ needs to wait for T_1 to release $write\_lock(B)$.

Cache Flushing:

Steal: A cache page can be flushed before a transaction is committed.

Force: Cache is flushed to disk when a transaction is committed.

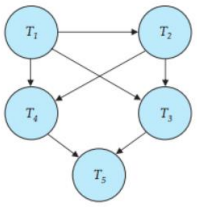
No Steal and Force: Do not need to redo or undo because if you have no steal, can't flush until it is completed. Then transactions as soon as they are committed, they are flushed to the disk.

General Facts:

Dead locks do not occur in serial executions

If a system supports ACID, then you can treat the transactions as if they were processed serially.

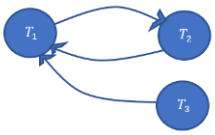
If the graph is acyclic there can be no conflicts so it's serializable. If it's cyclic then there can be deadlocks so it can't be serializable. The following schedule conflict is serializable because there are no cycles.



Schedule:

time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
T_1	R(A)	W(A)	W(E)	R(B)		W(B)	R(C)	W(C)	R(D)	W(D)
T_2			R(A)		W(A)	R(D)	W(D)	W(E)		W(A)
T_3				R(C)	W(C)			R(F)	W(F)	

Precedency Graph: For the above schedule



Recovery Techniques:

- Atomicity and durability
- Brin database into last consistent state

Recovery Goals:

When transaction T commits, make updates permanent so it survives subsequent failures. When transaction T aborts, get rid of any updates on data item by aborted transaction in the database. Get rid of the effects of T in other transactions

Recovery Protocols:

- Undo for Atomicity: All uncommitted transactions from beg -> end, use BFIM
- Redo for Durability: All committed transactions beg-> end us AFIM
- **Logs:** Sequence of records representing modifications in order, physical and logical.

Buffer Manager/Cache Manager

- Maximise the likelihood that a block of data needed by a transaction is in main memory.

Relational Algebra:

Question 13 [10 marks]:

You are given the following tables, where the primary keys are underlined:

branch (branch_name, branch_city, assets)

account (account_number, branch_name, balance)

Now, consider the following SQL query:

```
Select T.branch_name
From branch T, branch S
Where T.assets > S.assets and S.branch_city = "Brisbane"
```

Write the most optimized relational-algebra expression that is equivalent to that query. Justify your choice.

$$\pi_{Branch\_name}(T)(\pi_{assets}(S)\left(\sigma_{Branch_{city}="Brisbane"}(S)\right)\bowtie_{T.assests>S.assest}(\pi_{T.assets,\ T.branch\_name}(T))$$

Question 4

You are given the following tables:

Student(StudId, Name, Addr, Status)

Transcript(Id, CrsCode, Semester, Grade)

Consider a query that outputs the names of all students who took INFS2200 in 2013. An execution plan for that query can be expressed as follows:

4.2) [3 marks] Give the most **efficient** relational algebra expression that is equivalent to the one above.

$$\pi_{name}(\pi_{studid,\ name}(Student))\bowtie_{Studid=id}(\sigma_{CrsCode=\text{INFS2200 } AND\ Semester="2013"}(\pi_{CrsCode,\ Semester,\ Id}(Transcript))$$