COSC265 — Relational Database Systems

Neville Churcher

Department of Computer Science & Software Engineering
University of Canterbury

2021

Database & State

- ☆ Consider database as collection of named data items
- ☆ Granularity (table, block, attribute, ...)
- ☆ Consistency & correctness essential
- ☆ Database $\mathcal{D}$ has with schemas $\mathcal{R}$ and dependencies $\mathcal{F}$
- ☆ If we perform operations $\mathcal{O}_\rangle$ then we expect
- ☆ $|\mathcal{D}'(\mathcal{R}', \mathcal{F}') >= \mathcal{O}_n \ldots \mathcal{O}_2\mathcal{O}_1|\mathcal{D}(\mathcal{R}, \mathcal{F}) >$

Transactions

- Atomic unit of work
- Analogy with multiprocessing OS: single-CPU multi-user DBMS can only execute single process at a time
- Fundamental operations *interleaved* to approximate parallel processing
- Transactions may access shared data without interfering with each other
- "All or nothing" (commit/abort)

Basic Data Access Operations

$r(x)$ Read a data item (typically into program variable of same name)

- ✰ Locate disc block containing x
- ✰ Copy block to buffer (if not already present)
- ✰ Extract data item and assign to variable

$w(x)$ Write program variable to corresponding database item

- ✰ Locate disc block containing x
- ✰ Copy block to buffer (if not already present)
- ✰ Copy x from variable to correct buffer location
- ✰ Store updated block (in-place, shadow, . . .)

Representing Transactions

- Transaction identifier T_i, T_j
- Operations (read, write, commit, abort, ...) and their order
- $r_i[x]r_i[y]r_j[x]w_i[x]c_i$
- Order (ignoring other operations) $r_i[x] \to w_i[x] \to c_i$
- Data items accessed. Read-set (write-set) is set of all data items read (written) by transaction

Transactions & Recovery

- ☆ System log records all updates
- ☆ Transaction details
- ☆ Before/after images
- ☆ Can be large
- ☆ Used for recovery/restart
- ☆ Undo operation must be idempotent

Transactions & State

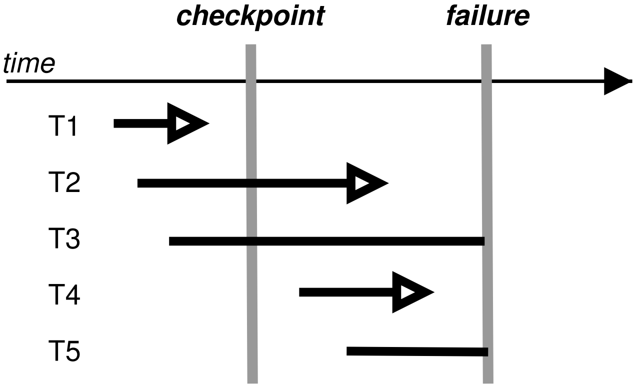
- ☆ Read-set (write-set) is set of all data items read (written) by transaction
- ☆ Begin/end transaction operations
- ☆ Commit transaction — changes committed to database and can not be undone
- ☆ Rollback/abort — changes made by unsuccessful transaction are undone (pretend they never happened)
- ☆ Transaction states

  Active: Has started, r_i, w_i possible
  Partially committed: Operations completed
  Committed: Changes permanent
  Failed: Aborted or couldn't be committed
  Terminated: Left active mix

Checkpoints & Logging

- Taken periodically e.g. after
  - n transactions
  - t seconds
  - b bytes of log data
- Suspend (temporarily) active transactions
- Force-write log buffers to physical log files
- Force-write checkpoint record to log
  - IDs of active transactions
  - Address of last log record for each
- Force-write modified memory buffers to disc
- Place address of checkpoint record in restart file

Recovery Management (simplified)



T1: No action required

T2, T4 Redo

T3, T5: Undo

☞ See text for more detail

ACID Properties

Ideally, DBMS should ensure that transactions are:

Atomic: All (commit) or nothing (rollback)

Consistent: DBMS enforces integrity constraints on initial & final states; programmers responsible for logic.

Isolated: Concurrent transactions should not interfere with each other in any way

Durable: Changes must be persistent (after commit) and failures should not cause data loss.

How Big is a Transaction?

- ☆ Part of SQL statement?
- ☆ Single SQL statement?
- ☆ Several consecutive SQL statements with no intervening host language code?
- ☆ Several consecutive SQL statements and application code & system calls?
- ☆ Arbitrary program?

Transactions in SQL

- ☆ SQL views queries as part of transaction
- ☆ Changes not permanent until COMMIT or ROLLBACK issued
- ☆ Most DBMS include AUTOCOMMIT as user-settable option
- ☆ If autocommit is ON then each SQL transaction is treated as separate transaction (i.e. `\g` ≡ `\g COMMIT`)
- ☆ Multi-statement transactions

```
BEGIN
 INSERT INTO foo ...
 ...
COMMIT
```

Lost Update Problem

Inconsistent Analysis

Isolation: Undesirable Phenomena

Dirty Read: T_i reads data written by uncommitted T_j

Non-Repeatable Read: Successive reads $r_i(d)$, $r_i'(d)$ of data by T_i produce different results because data modified by transaction T_j which has committed after the first read.

$$\ldots r_i'(d) \ldots c_j \ldots r_i(d) \in \mathcal{H}$$

Phantom Read: Record sets S, S' returned from successive select query in T_i differ because of records added/deleted by T_j

Isolation Levels

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
Read Uncommitted	✔	✔	✔
Read Committed	✘	✔	✔
Repeatable Read	✘	✘	✔
Serializable	✘	✘	✘

- ☆ Read Uncommitted only for the brave/foolhardy?
- ☆ Repeatable Read really only adds protection against phantom reads—which tend to be rare.
- ☆ Individual DBMS products may not offer all four levels.

Isolation Levels in Practice

- ☆ Oracle implements *Read Committed*, *Serializable*, *read-only*
- ☆ Default is `READ COMMITTED`
- ☆ Set for current transaction (before first DML query)

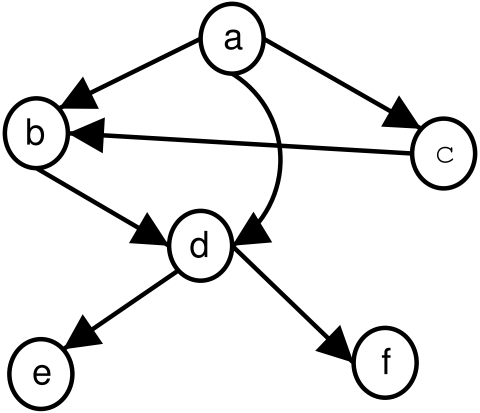
  ```
  set transaction isolation level serializable
  ```
- ☆ Set at session level (DBMS dependent)

  ```
  set session characteristics as transaction
      isolation level read committed
  ```
- ☆ set as variable (DBMS dependent))

  ```
  set default_transaction_isolation = 'value'
  ```
- ☆ Set system default via configuration file

Topological Sort

- Sequence of nodes of DAG G such that if a appears before b in the sequence then there is no path from b to a in G



- a, c, b, d, e, f
- a, c, b, d, f, e

Partial Orders

- ☆ Partial order $\mathcal{P} = (\Sigma, <)$
- ☆ Σ is the domain of $\mathcal{P}$
- ☆ $<$ is binary operation on Σ

  irreflexive: $a \not< a \quad \forall a \in \Sigma$

  transitive: $\{a<b, b<c\} \models a<c$
- ☆ a precedes b in $\mathcal{P}$ if $a<b$
- ☆ a, b incomparable if $a \not< b, b \not< a$
- ☆ Partial orders equivalent to corresponding DAG

  $\mathcal{P}(\Sigma, <) \Rrightarrow G(N, E)$ where $N = \Sigma$ and $(a, b) \in E$ iff $a<b$

  $G(N, E) \Rrightarrow \mathcal{P}(\Sigma, <)$ where $\Sigma = N$ and $a<b$ iff $(a, b) \in E^+$

Transactions as Partial Orders

- ☆ $T_i \equiv T_i(\Sigma_i, <_i)$
- ☆ Transactions terminated by t_i commit or abort
- ☆ $a_i \in T_i$ iff $c_i \notin T_i$
- ☆ $\{\forall o_i \in T_i, o_i \neq t | o_i <_i t_i\}$
- ☆ If $r_i[x], w_i[x] \in T_i$ then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$

Schedules/Histories

- $S(T_1, \ldots, T_n)$ contains (interleaved) operations of transactions
- Operations of T_i (including c_i, a_i) appear in order of occurrence
- $r_i[x] \ldots r_i[y] \ldots w_i[x] \ldots c_i$
- History specifies order of conflicting operations
- $o_i[x]$ conflicts with $o_j[x']$ if
  - $i \neq j$
  - $x = x'$
  - At least one of o_i, o_j is a write operation
- In general, a history is a partial order. Non-conflicting operations can occur in either order.

Histories & Equivalence

$H_i \equiv H_j$ if

- ☆ defined over same set of transactions
- ☆ have same operations
- ☆ order conflicting operations of non-aborted transactions in the same way
- ☆ For conflicting operations $p_i \in T_i, q_j \in T_j$ where $a_i, a_j \notin H$ then if $p_i <_H q_j$ then $p_i <_{H'} q_j$

Serialisation & Histories

Lost update revisited

- ☆ $H_1 = r_1[x]r_2[x]w_1[x]c_1w_2[x]c_2$
- ☆ $H_2 = r_1[x]w_1[x]c_1r_2[x]w_2[x]c_2$
- ☆ $H_2 \equiv T_1T_2$
- ☆ $H_1 \not\equiv H_2$ — order of conflicting operations $r_2[x], w_1[x]$ differ
- ☆ $H_1 \not\equiv T_1T_2,\ H_1 \not\equiv T_2T_1$

- ☆ Serial execution corresponds to total order of transactions in history. One transaction completes before another begins
- ☆ Serialisable history is equivalent to some *serial history* of same transactions

Testing for Serialisability

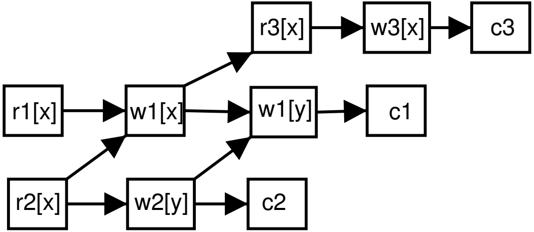
- ☆ Serialisation graph
- ☆ Nodes represent transactions in H
- ☆ Each (directed) edge $T_i \rightarrow T_j$ indicates that at least one operation of T_i precedes and conflicts with one of T_j's
- ☆ Edge $T_i \rightarrow T_j (i \neq j)$ where $\exists p_i \in T_i, q_j \in T_j : p_i <_H q_j$ and p_i conflicts with q_j
- ☆ T_i should precede T_j in any serial history $\equiv H$

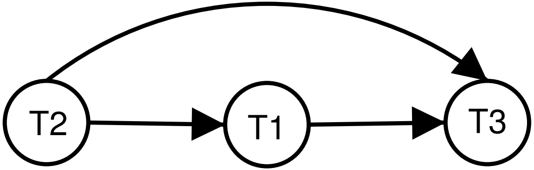
Serialisability Theorem

History H is serialisable iff $SG(H)$ acyclic

Serialisation Graph Example

History & serialisation graph

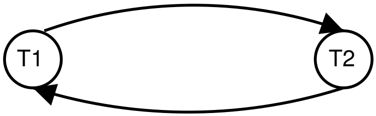




- ☆ $SG(H)$ acyclic: $H \equiv H_S = T_2 T_1 T_3$
- ☆ If $w_3[x]$ replaced by $w_3[z]$ then $T_2 \rightarrow T_3 \notin SG(H)$

Lost Update & Serialisation

- $H = r_1[x]r_2[x]w_1[x]c_1\,w_2[x]c_2$



- $SG(H)$ cyclic
- H not serialisable
- $H_X = r_1[x]w_1[x]c_1\,r_2[x]w_2[x]c_2$
- $SG(H_X) = \{T_1 \rightarrow T_2\}$
- $SG(H_X) \equiv T_1\,T_2$

Concurrency Control

- Requested operations may be:
  - scheduled immediately
  - delayed (queued)
  - rejected (abort transaction)
- *Aggressive schedulers* avoid delays. Risk having to reject operations later to produce serialisable execution history.
- *Conservative schedulers* use delays to re-order operations & reduce risk of rejection.
  - Extreme case is *serial* scheduler
  - Needs knowledge of read-sets & write-sets

Locking

- Extend transaction operations to include $rl_i[x], wl_i[x]$ read, write locks placed on x by T_i
- Read locks also called *shared* locks; write locks also called *exclusive* locks
- Also add $ru_i[x], wu_i[x]$ operations to release locks
- Various lock types ...
  - exclusive, share, intention, ...
- ... and granularities ...
  - Database, table, tuple, logical record, predicate, ...

Resolving Lock Conflicts

☆ Conflicts: $pl_i[x], ql_j[y]$ conflict if $x = y$ and $i \neq j$ and p, q conflicting operations

		T_i **has**		
		Exclusive	**Shared**	–
T_j wants	**Exclusive**	✘	✘	✔
	Shared	✘	✔	✔
	–	✔	✔	✔

Two Phase Locking (2PL)

1. All data items locked before use. If requested lock conflicts with existing lock then lock request is delayed and requesting transaction delayed
2. Lock pl_i set by scheduler not released until (at least) after DM (data manager) acknowledges processing of corresponding operation p_i
3. Once *any* $pu_i[x]$ has been scheduled T_i may not obtain any more locks

- ☆ Rule 1 prevents transactions from concurrently accessing a data item in conflicting modes. Conflicting operations scheduled in order that corresponding locks are obtained.
- ☆ Rule 2 forces DM to process operations in order scheduler sends them & helps prevent setting of conflicting locks
- ☆ Rule 3 divides transaction into 2 phases

2PL (continued)

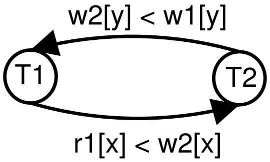
Growing phase: locks may be obtained

Shrinking phase: locks released

- Consider $T_1 : r_1[x] \rightarrow w_1[y] \rightarrow c_1 \quad T_2 : w_2[x] \rightarrow w_2[y] \rightarrow c_2$

$$H_\alpha = rl_1[x]r_1[x]ru_1[x]wl_2[x]w_2[x]wl_2[y] \\ w_2[y]wu_2[x]wu_2[y]c_2wl_1[y]w_1[y]wu_1[y]c_1$$

- Not 2PL since $ru_1[x] <_{H_\alpha} wl_1[y]$
- Between $ru_1[x]$ and $wl_1[y]$ T_2 writes both x and y
- $SG(H_\alpha)$ cyclic



- T_2 appears to follow T_1 wrt x and precede it wrt y
- *Any 2PL schedule is serialisable*

2PL & Deadlocks

$$T_1: \; r_1[x]w_1[y]c_1$$
$$T_2: \; w_2[y]w_2[x]c_2$$
$$H_{DL}: \; rl_1[x]r_1[x]wl_2[y]w_2[y]$$

1. 2PL scheduler receives $r_1[x]$ from TM, sets $rl_1[x]$ and passes $r_1[x]$ to DM
2. $w_2[y]$ received from TM, sets $wl_2[y]$ and passes $w_2[y]$ to DM
3. $w_2[x]$ received from TM but can't set $wl_2[x]$ as conflicts with existing $rl_1[x]$ so $w_2[x]$ delayed
4. $w_1[y]$ received but can't set $wl_1[y]$ as conflicts with existing $wl_2[y]$ so $w_1[y]$ delayed
5. Neither transaction can continue without violating 2PL

Deadlocks & Lock Promotion

- ☆ Deadlock can also arise in lock promotion
- ☆ If $\exists r_i[x], w_i[x] \in T_i$ then $rl_i[x]$ must be promoted to $wl_i[x]$
- ☆ Problem occurs if other transactions have been granted read locks on x before write lock request received
- ☆ i.e. $rl_i[x] < rl_j[x] < wl_i[x]$

Resolving Deadlocks

- Deadlock avoidance or deadlock detection
- Hard to insist that all required data items be locked in advance
  - Infeasible to order objects (and hence lock requests)
  - Lockable objects often addressed by content rather than name (can't tell *a priori* whether distinct requests are for same object)
  - Scope of locking may be determined dynamically
- Avoidance techniques often use transaction timestamps — unique transaction ID based on order in which transactions started

Wait-For Graph (WFG)

- ☆ Nodes represent active transactions
- ☆ Edge $T_i \rightarrow T_j$ when T_i is waiting for a lock on some x currently locked by T_j
- ☆ Edges removed when locks released
- ☆ Deadlocks correspond to cycles in WFG — avoid adding them or search for them
- ☆ If $T_i \rightarrow T_j$ is in the WFG and both transactions commit then $T_j \rightarrow T_i$ will appear in the SG
- ☆ If either transaction aborts then no SG edge will be contributed

WFG Example

T_1: $r_1[x]w_1[y]c_1$

T_2: $w_2[y]w_2[x]c_2$

H_{DL}: $rl_1[x]r_1[x]wl_2[y]w_2[y]$

- ☆ WFG null
- ☆ $w_2[x] > rl_1[x]$ so T_2 waiting on T_1 and edge $T_2 \to T_1$ added to WFG
- ☆ $w_1[y] > wl_2[y]$ so T_1 waiting on T_2 and edge $T_1 \to T_2$ added to WFG to form cycle $\Rightarrow$ deadlock

Using the WFG

When T_A requests object already locked by T_B (adding $T_A \rightarrow T_B$ to WFG)

Wait-Die:

- ☆ T_A waits if it is older than T_B
- ☆ T_A dies if it is younger than T_B (abort and restart with same timestamp)
- ☆ WFG consists of older transactions waiting on younger ones (hoping they will finish soon)
- ☆ Favours young transactions (waits for each of them)
- ☆ As transaction ages it is likely to wait on more and more young 'uns

Using the WFG (continued)

Wound-Wait:

- ☆ T_A wounds T_B if T_A is older (try to abort and restart T_B with same timestamp)
- ☆ T_A waits if it is younger than T_B
- ☆ WFG has only younger transactions waiting on older ones
- ☆ Stroppy old transactions push themselves through the mix, damaging younger ones irrespective of how close to completion they are or how much work they have done
- ☆ Both wait-die and wound-wait only kill off younger transactions

Using the WFG (concluded)

Timeout: Alternative is to check periodically for cycles

- ☆ Choose victim
  - ☆ rollback (releasing locks)
  - ☆ restart
- ☆ Victim selection based on
  - ☆ age (youngest?)
  - ☆ locks held (fewest?)
  - ☆ work done (fewest updates?)
  - ☆ ...