Transaction management

Overview

- Why concurrent execution of programs?
- What properties might we wish for?
- What is a transaction?
- What are the problems when interleaving transactions?
- How might we overcome these?

User Programs

- Concurrent execution of user programs is essential for good DBMS performance

- A user’s program may carry out all sorts of operations on the data, but the DBMS is only concerned about what data is read from/ written to the database

Transactions

- Thus a **transaction** is the DBMS’s abstract view of a user program: a series of reads/writes of database objects

Transactions cont.

- A transaction is seen by the DBMS as a series, or list, of actions
  - Includes read and write of objects
  - We'll write this as R(o) and W(o) (sometimes R_T(o) and W_T(o))
- For example

  T1: [R(a), W(a), R(c), W(c)]

  T2: [R(b), W(b)]
- In addition, a transaction should specify as its final action either **commit**, or **abort**

Transactions cont.

- Users submit transactions, and can think of each transaction as executing by itself
  - The concurrency is achieved by the DBMS, which interleaves actions of the various transactions
- Issues:
  - Interleaving transactions (data inconsistency or data loss), and
  - Crashes!

Goal: The ACID properties

- **Atomicity**: Either all actions are carried out, or none are (transaction either executes 0% or 100%)
- **Consistency**: If each transaction is consistent, and the database is initially consistent, then it is left consistent (i.e., a trans. leads to consistent state)
- **Isolation**: Intermediate transaction results must be hidden from other concurrently executed transactions
- **Durability**: If a transaction completes successfully, then its effects persist (the changes it has made to the database should be permanent)

Atomicity

- A transaction can
  - **Commit** after completing its actions, or
  - **Abort** because of
    - Internal DBMS decision: restart
    - System crash: power, disk failure, …
    - Unexpected situation: unable to access disk, data value, …
- A transaction interrupted in the middle could leave the database inconsistent
- DBMS needs to remove the effects of partial transactions to ensure **atomicity**: either all a transaction’s actions are performed or none

Atomicity cont.

- A DBMS ensures atomicity by **undoing** the actions of partial transactions
- To enable this, the DBMS maintains a record, called a **log**, of all writes to the database
- The component of a DBMS responsible for this is called the **recovery manager**

Consistency

- When a transaction run to completion against a consistent input database instance, the transaction leaves the database consistent

- For example, consistency criterion that my inter-account-transfer transaction does not change the total amount of money in the accounts!

Integrity
Constraints!

- **Database consistency** is the property that every transaction sees a consistent database instance. It follows from transaction atomicity, isolation and transaction consistency

Isolation

- Guarantee that even though transactions may be interleaved, the net effect is identical to executing the transactions **serially**
- For example, if transactions T1 and T2 are executed concurrently, the net effect is equivalent to executing
  - T1 followed by T2, **or**
  - T2 followed by T1
- NOTE: The DBMS provides **no guarantee of effective order of execution**

Durability

- DBMS uses the log to ensure durability
- If the system crashed before the changes are committed to disk, the log is used to remember and restore these changes when the system is restarted
- This is handled by the recovery manager

Transactions and schedules

- A transaction is seen by the DBMS as a series, or list, of actions
  - Includes read and write of objects
  - We'll write this as R(o) and W(o) (sometimes R_T(o) and W_T(o))
- For example

  T1: [R(a), W(a), R(c), W(c)]

  T2: [R(b), W(b)]
- In addition, a transaction should specify as its final action either **commit**, or **abort**

Schedules

- A **schedule** is a list of actions from a set of transactions
  - A well-formed schedule is one where the actions of a particular transaction T are in the same order as they appear in T
- For example
  - [R_{T1}(a), W_{T1}(a), R_{T2}(b), W_{T2}(b), R_{T1}(c), W_{T1}(c)] is a well-formed schedule
  - [R_{T1}(c), W_{T1}(c), R_{T2}(b), W_{T2}(b), R_{T1}(a), W_{T1}(a)] is not a well-formed schedule

T1: [R(a), W(a), R(c), W(c)]
T2: [R(b), W(b)]

Schedules cont.

- A **complete schedule** is one that contains an abort or commit action for every transaction that occurs in the schedule

- A **serial schedule** is one where the actions of different transactions are not interleaved

Serialisability

- A **serialisable schedule** is a schedule whose effect on any consistent database instance is identical to that of some complete serial schedule

Anomalies with interleaved execution

- Two actions on the same data object **conflict** if at least one of them is a write
- We’ll now consider three ways in which a schedule involving two consistency-preserving transactions can leave a consistent database inconsistent

WR conflicts

- Transaction T2 reads a database object that has been modified by T1 which has not committed

Debit Rs100 from a

Credit Rs100 to b

T1: R(a),W(a), R(b),W(b),C

T2: R(a),W(a),R(b),W(b),C

Read a and b and add 6% interest

"Dirty read"

RW conflicts

- Transaction T2 could change the value of an object that has been read by a transaction T1, while T1 is still in progress

T1: R(a), R(a), W(a), C
T2: R(a),W(a),C

"Unrepeatable Read"

Read A (5) Write 5+1=6

T1: R(a), W(a),C
T2: R(a), W(a),C

A is 4 ☹

Read A (5) Write 5-1=4

WW conflicts

- Transaction T2 could overwrite the value of an object which has already been modified by T1, while T1 is still in progress

T1: [W(Britney), W(gmb)]   "Set both salaries at £1m"

T2: [W(gmb), W(Britney)]   "Set both salaries at $1m"

- But:

"Blind Write"

T1: W(Britney), W(gmb)

T2: W(gmb), W(Britney)

gmb gets £1m
Britney gets $1m
☺

Serialisability and aborts

- Things are more complicated when transactions can abort

Deduct €100
from a

T1:R(a), W(a), **Abort**

T2: R(a),W(a),R(b),W(b),C

Add 6% interest
to a and b

Can’t undo T2
It’s committed ☹

Strict two-phase locking

- DBMS enforces the following locking protocol:
  - Each transaction must obtain an S (**shared**) lock before reading, and an X (**exclusive**) lock before writing
  - All locks held by a transaction are released when the transaction completes
  - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object
- Strict 2PL allows only serialisable schedules

More refined locks

- Some updates that seem at first sight to require a **write (X)** lock, can be given something weaker
  - Example: Consider a seat count object in a flights database
  - There are two transactions that wish to book a flight – get **X** lock on seat count
  - *Does it matter in what order they decrement the count?*
    - They are **commutative actions!**
    - Do they need a write lock?

Aborting

- If a transaction Ti is aborted, then all actions must be undone
  - Also, if Tj reads object last written by Ti, then Tj must be aborted!
- Most systems avoid **cascading aborts** by releasing locks only at commit time (strict protocols)
  - If Ti writes an object, then Tj can only read this after Ti finishes
- In order to undo changes, the DBMS maintains a **log** which records every write

The log

- The following facts are recorded in the log
  - "Ti writes an object": store new and old values
  - "Ti commits/aborts": store just a record
- Log records are chained together by transaction id, so it's easy to undo a specific transaction
- Log is often duplexed and archived on stable storage (it's important!)

Connection to Normalization

- The more redundancy in a database, the more locking is required for (update) transactions.
  - Extreme case: so much redundancy that, all update transactions, are forced to execute serially.

- In general, less redundancy allows for greater concurrency and greater transaction throughput.

!!! This is what normalization is all about !!!

The Fundamental Tradeoff of Database Performance Tuning

- De-normalized data can often result in faster query response
- Normalized data leads to better transaction throughput

Yes, indexing data can speed up query response time, but an index is redundant data. General rule of thumb: indexing will slow down transactions!

What is more important in your database --- query response or transaction throughput? The answer will vary.

Summary

You now understand:

- Transactions and the ACID properties
- Schedules and serialisable schedules
- Potential anomalies with interleaving
- Strict 2-phase locking
- Problems with transactions that can abort
- Logs