Chapter 19-21 - Transaction Processing Concepts

transaction - logical unit of database processing

- becomes interesting only with multiprogramming - multiuser database
- more than one transaction executing concurrently
- actually - programs are interleaved - part of one trans follows part of another
    - keeps CPU busy when an executing program requires I/O
    - also provides a level of fairness to transactions

transaction - (another def) - the execution of a program that accesses or changes the contents of the database

Operations of a transaction:

read_item(X) - reads database item X into program variable (also called X for convenience)

1) Find disk block address of X
2) Copy block into main memory
3) Copy X from MM to prog variable

write_item(X) - writes value of program variable X to data item X

1) Find disk block address of X
2) Copy block into main memory
3) Copy item X from prog variable to MM
4) Store updated block in MM to disk

Example transactions:

(a)

```
read_item(X);
X:=X-N;
write_item(X);
read_item(Y);
Y:=Y+N;
write_item(Y);
```

(b)

```
read_item(X);
X:=X+M;
write_item(X);
```

Concurrency Control:

- need to control the concurrent access to database by multiple transactions

illustrate the problem with the toy catalog application

- orders come in which must decrement the inventory for a toy
- new shipments from manufacturers come in that must increment the inventory for a toy

example (a) above shows a transaction that removes a toy from inventory (due to an order) and then adds another toy to the inventory (due to a shipment or cancelled order)

example (b) above shows a simpler transaction that adds a toy to inventory

Problems that illustrate the need for concurrency control:

1) Lost update problem:
    - two transactions access the same database items interleave such that values of some updates are lost

    - example: (draw fig 17.3a)

    - final value of X is incorrect because T2 reads the value of X before T1 changes it in the database

2) Temporary Update (dirty read) problem:
    - one trans update database item and then fails - if another trans reads from that update, it has an incorrect value (dirty read)

3) Incorrect summary problem:
    - transaction computing some kind of aggregate function interleaves with other updating transactions
    - aggregate function may calculate some values before update and others after

4) Unrepeatable read:
    - one trans reads an item twice - once before another trans updates, once after
    - gets two different values

Recovery:

- atomic property of transactions: either all operations in the trans are completed successfully, or none

- if trans fails, must recover database to a consistent state (one in which all trans maintain the above property)

Types of failures:

1) System crash: hardware or software error
2) Trans or system error: eg: div by zero

3) Local errors or exception conditions: aborted trans due to some exception within the program
4) Concurrency control enforcement: some CC protocols use aborts to maintain consistency among all transactions (we will see later)
5) Disk failure:
6) Physical problems or catastrophes: power out, fire, theft, etc.

- system must keep sufficient information to recover from failure

Transaction Concepts:

- recovery manager keeps track of the following operations:

  BEGIN_TRANSACTION
  READ WRITE
  END_TRANSACTION (end of reads and writes - must still be committed)
  COMMIT_TRANSACTION: successful end of transaction - updates are safely committed to the database (disk)
  ROLLBACK (ABORT): unsuccessful end of transaction - updates must be undone
  UNDO: applies to single operation rather than a whole trans
  REDO: redo certain trans operations to endure all ops of committed trans are applied successfully to database

(draw Fig 17.4)

- every trans goes through the state transition diagram
  - active - performing read / write ops
  - partially committed - check to make sure it can be committed (some CC may abort depending on state of other trans)
    - also check to make sure system failure will not result in an inability to record changes of the trans
  - committed - passes all tests of above state
  - failed - failed any of above tests
  - terminated - leave system

System log
  - keeps track of all trans ops that affect values of database items

- types of entries in log:

    1) [start_transaction,T]
    2) [write_item,T,X,old_value,new_value]
    3) [read_item,T,X]
    4) [commit,T]
    5) [abort,T]

Commit point
- when all operations that access database have been executed successfully
- and effect of trans has been recorded in log
- beyond commit point - trans is considered committed

Checkpoints
- [checkpoint] written into log when system writes to disk effects of all WRITEs of committed trans
- recovery manager decides when to checkpoint
- performing a checkpoint:
    1) Suspend execution of all trans
    2) write all update ops of committed trans from MM to disk
    3) write [checkpoint] to log
    4) resume execution of trans

ACID properties of transaction (desireable properties):

1) Atomicity - either all of trans is performed, or none at all
2) Consistency - trans must take database from one consistent state to another
    - consistent state satisfies all constraints specified in schema
3) Isolation - trans should not make updates visible to other trans until commit (independence of transactions)
4) Durability - once committed, trans effects should be permanent

Schedule - S of n transaction T1,...,Tn is an ordering of the operations of the transactions such that for each trans Ti that participates in S, the ops of Ti must appear in the same order in S as they do in Ti

Two operations of a schedule are said to conflict if:
1) they belong to different transactions
AND 2) one is a write

(draw compatibility table)

A schedule S of trans T1...Tn is complete if:

1) ops in S are exactly the ops in T1,...,Tn
2) order of ops from each trans is maintained
3) for any two conflicting operations, one of the two must occur before the other in the schedule

- thus we have a partial order of all operations in S
- but a total order on conflicting operations and on ops in the same trans

- committed projection C(S) of S includes only the operations in S that belong to committed transactions

A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed

example of a recoverable schedule:

r1(X); r2(X);w1(X);r1(Y);w2(X);w1(Y);c1

example of a non-recoverable schedule:

r1(X);w1(X);r2(X);r1(Y);w2(X);c2;a1;

- because T2 reads X from T1 and then T2 commits before T1 commits

cascading rollback: an uncommitted trans has to be rolled back because it read a trans that failed - ex:

r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1

Deadlock prevention:

- use transaction timestamp TS(T) - ordered based on when trans started
    - older trans gets lower timestamp
- trans Ti tries to lock X, but can't because Tj holds a conflicting lock on X

- wait-die: if TS(Ti) < TS(Tj) (Ti older than Tj)
  then Ti is allowed to wait
  else abort Ti (Ti dies) - restart later with same timestamp

- wound-wait: if TS(Ti) < TS(Tj)
  then abort Tj (Ti wounds Tj) and restart later with same timestamp
  else Ti is allowed to wait

- both schemes abort younger trans that may be involved in deadlock
- will prevent deadlock - but can be overly conservative (abort trans that may never be involved in deadlock)

- cautious waiting: if Tj is not blocked
  then Ti is blocked and allowed to wait
  else abort Ti

- timeout: system aborts a trans after a certain set time limit expires - assuming that it is stuck in deadlock

deadlock detection: periodically check system for deadlock using a wait-for graph (represents which trans are waiting for which others in a graph)

Livelock: trans is prevented from proceeding for an indefinite period of time
- may be due to an unfair timesharing scheme - or due to priority assignment

Timestamp ordering:

- produces a schedule that is equivalent to the particular serial schedule that corresponds to the order of the transaction timestamps

- checks whenever two conflicting operations occur in the incorrect order and rejects the later of the two operations by aborting the trans

- associates two timestamp values:

    - read_TS(X) - largest ts of all trans that have successfully read X
    - write_TS(X) - lartest ts of all trans that have successfully written X

- trans T issues write_item(X)
    - if TS(T) < read_TS(X) { some trans with TS > TS(T) already
      then abort and rollback T read X - violating TS order }

      else execute write_item(X) and set write_TS(X) := TS(T)

- trans T issues read_item(X)
    - if TS(T) < write_TS(X)

then abort and roll back T

else execute read_item(X) and set
read_TS(X) := max(TS(T),read_TS(X))

Multiversion Concurrency Control - read on your own

Optimistic CC:

- basic idea - let each trans execute as if it will never conflict (optimistic), then before committing, check to find out if any conflicts occurred and abort the necessary transactions

- Three phases:

  read phase - do all reads from database - writes done to local copy

  validation phase - check to ensure that serializability will not be violated of updates are applied

  write phase - if validation successful then apply updates and commit - otherwise abort and restart trans

Recovery Techniques:

- recovery from transaction failure - restore database to some earlier state (close to time of failure)

- two approaches to updates:
  1) deferred update - trans update local copies of data and don't write to db until after commit - no undo necessary (recovery is minimal) - redo may be necessary after commit

  2) immediate update - trans update db before commit - write all ops to system log so they can be undone and redone

- recovery techniques based on these two approaches:

Deferred Update -

- multiple user with concurrent transactions
  - assuming CC holds all locks until commit point
  - keep two lists of trans:
    1) committed since last checkpoint
    2) active transactions

  - redo all writes of committed trans in same order as they appear in log
  - all active trans are cancelled and restarted (never touched db so no undo's necessary)

Immediate Update -

- multiple user with concurrent transactions
    - assume strict 2PL for CC
    - again, use two lists of trans, committed and active
    - undo all writes of active trans in reverse order as in log
    - redo all write ops of committed trans

Shadow paging -

- database made up of a fixed number of disk pages
- page table with n entries is kept - ith entry points to ith db page on disk
- when trans begins, current page table copied to shadow page table
    - trans uses current page table leaving shadow on disk
- when a trans does a write, a copy of the page written on is created and modified
- in the event of failure, recovery involves freeing the modified db pages and discarding the current page table

- advantages - no undo or redo
- disadvantages - db pages can change location of disk - making it difficult to keep related data close together