Transaction Scheduling and Preemption

Problem Set #4 V1.0 is posted.

Expect V1.1 this weekend with a Problem 7 and more clarifications.

Database Transactions

- A *transaction* is the DBMS's abstract view of a *user program*: a sequence of database commands; disk reads and writes.
- *Concurrent execution* of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu busy by working on several *transactions* concurrently (like an OS).
- A user's program may carry out many consecutive operations on the data retrieved from the database, but the DBMS is only concerned about what data is *read from* and *written to* the database.

ACID Properties of Transactions

- *Atomic*: the end effect of a transaction should be *all or nothing*. Either it is executed to completion, or it is as if it never happened. (DBMS provides this)
- *Consistency*: Every transaction must preserve all integrity constraints of the database. (User and DBMS)
- *Isolation*: The result of a transaction should give predictable results regardless of any other concurrent transactions. (DBMS)
- *Durability*: Transactions must tolerate and recover from crashes and allow for being aborted before completion. The result after crash recovery or aborting a transaction should leave the database in a consistent state. (DBMS)

Concurrency in a DBMS

- Users submit a transaction, and can consider it as executing *by itself* on the database.
  - Concurrency is provided by the DBMS, which interleaves the actions (reads/writes) of many ongoing transactions.
  - Each transaction must leave the database in a consistent state if the DB was consistent when the transaction began.
  - DBMSs only enforce Integrity Constraints
  - Beyond this, the DBMS does not understand the data. (e.g., it does not understand how interest on a bank account is computed).
- *Issues:* Effect of *interleaving transactions* and *crashes.*

Interleaving's Impact

- Interleaving improves database performance
  - While one transaction waits for pages to be read from disk, the CPU processes other transactions. I/Os proceed in parallel with CPU activity
    (greater system utilization)
  - Increased system *throughput* (transactions/sec)
  - More "fair" than true sequential access; allows all pending transactions to make progress (heavy transactions, don't starve out light ones)
  - Predictable *latency* (delay from request to completion)
- However, ad hoc interleaving can lead to anomalies
  - Sequential inconsistency

Example

- Consider two transactions (*Xacts*):

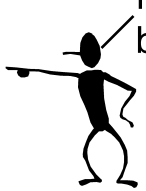
T1: BEGIN C=C+100, S=S-100 END
T2: BEGIN C=1.02*C, S=1.04*S END

- Intuitively, the first transaction is transferring $100 from a savings to a checking account. The second is crediting both accounts' interest payments.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to some execution of these two transactions run sequentially.

All Schedules are not Equal

- Consider a possible interleaving (schedule):

T1:	C=C+100,		S=S-100	
T2:		C=1.02*C,		S=1.04*S



Same as T1 followed by T2

- This is OK. But what about:

T1:	C=C+100,		S=S-100
T2:		C=1.02*C, S=1.04*S	

Inconsistent with any order of T1 and T2

- The DBMS's view of the second schedule:

T1:	$R_1(C)$, $W_1(C)$,		$R_1(S)$, $W_1(S)$
T2:		$R_2(C)$, $W_2(C)$, $R_2(S)$, $W_2(S)$,	

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions. *Too rigid, creates bottlenecks, reduces performance*
- *Equivalent schedules:* For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule:* A schedule that is equivalent to ***some*** serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule also preserves consistency.)

Atomicity of Transactions

- An important property guaranteed by the DBMS is that transactions are *atomic.* That is, a user can think of a Xact as either always executing all its actions in uninterrupted in order, or not executing any actions at all.
- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- DBMS *logs* all actions so that it can *undo* aborted transactions.

The 3 Classes of Anomalies

- Reading Uncommitted Data--
  Write-Read (WR) Conflict, "dirty reads":

T1: R(A), W(A), R(B), W(B), Abort
T2: R(A), W(A), R(B), W(B), C,

- Unrepeatable Reads--
  Read-Write (RW) Conflict:

T1: R(A), W(A), R(B), W(B), C
T2: R(A), W(A), R(B), W(B), C,

T2's write of A is lost

Anomalies (Continued)

- Overwriting Uncommitted Data
  Write-Write (WW) Conflict, "blind write":

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

T1's write of A is lost

- All 3 anomalies involve at least one write
- How do we avoid these?

Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a *shared* (S) lock on an object before reading, and an *exclusive* (X) lock on an object before writing. (of course, you can both read and write an object with an X lock)
  - All locks held by a transaction are released when the transaction completes (at Commit or Abort)
  - If an Xact holds an X lock on an object, no other Xact can get either an S or X lock on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies aborts (more soon)

Examples

- **Common case**: Xacts affect different parts of db.
  T1: B = f(B, A), T2: C = g(C, A)

T1: S(A), R(A),		X(B), R(B), W(B),C
T2:	S(A), R(A), X(C), R(C), W(C), C	

- **Hot spots**: Xacts reference a common record.
  T1: A = f(A), T2: B = f(B,A)

T1: X(A), R(A),		W(A), C	
T2:	S(A), … Waiting for lock X(A) to be released		R(A), X(B), R(B), W(B), C

T1:	X(A), … Waiting for lock S(A) to be released		R(A), W(A), C
T2: S(A), R(A), X(B),		R(B), W(B), C	

Deadlocks

- Transactions request exclusive access to a common locked record. T1: B = f(B, A), T2: A = g(A, B)

```
T1: S(A),R(A),X(B),R(B),      W(B),C
T2:                      S(B),...      R(B),X(A),R(A),W(A),C
```

- A rare unfortunate ordering, where both transactions wait, and make no progress

```
T1: S(A),R(A),            X(B),...      Abort,
T2:           S(B),R(B),          X(A), ...      R(A),W(A),C
```

- Soln: DBMS monitors how long a transaction has been waiting and aborts it, thus freeing its locks

Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Releasing transaction locks only on commit/abort avoids *cascading aborts* (abort handling is simplified)
  - If *Ti* writes an object, *Tj* can read it only after *Ti* frees lock.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

Transactions in SQL

- Transactions begin on any statement that references a table (CREATE, UPDATE, SELECT, INSERT, etc.)
- Transactions end when either a "COMMIT" or "ROLLBACK" (Abort) command is reached
- SQL provides a "SAVEPOINT *name*" to break up transactions into intermediate pieces, which can be gotten back to using
  "ROLLBACK TO SAVEPOINT *name*"
- Operations between 2 savepoints are handled as separate Xactions, in terms of concurrency control

The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: the *old value* and the *new value*.
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- All log related activities (and in fact, all concurrency-control related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.
- Complication: committed writes might be held in the buffer pool

Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were in progress, and all dirty pages in the buffer pool at crash time
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - *Undo*: The writes of all Xacts that were in progress at crash time are undone (by restoring the *old value* of the data, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - *Consistent state*: Only the effects of commited Xacts seen.