Transactions and Crash Recovery

CPS352: Database Systems

Simon Miner
Gordon College
Last Revised: 11/1/12
Agenda

- Check-in
- Transactions
- Exam 1
- Crash Recovery
Check-in
Ensuring Data Integrity

• Issues related to preserving data integrity
  • Concurrency control
  • Crash control

• *Transactions* are a key concept at the heart of these matters

• Database is in a *consistent* state if there are no contradictions between the data within it
  • Temporary inconsistencies occur by necessity, but must not be allowed to persist
  • Example: transfer of funds between bank accounts
Transactions
Definition

- A transaction is an atomic operation (unit of work) involving a series of processing steps including:
  - One or more reads from the database (one read per item)
  - One or more writes to the database (one write per item)
  - Data computations can happen during a transaction, but the database is mostly concerned with reads and writes

- If the database is in a consistent state at the start of the transaction, it will be in a consistent state at the end of the transaction
ACID

- **Atomicity** – either all of the transaction completes, or none of it completes
  - If any part of the transaction fails, all effects of it must be removed from the database

- **Consistency** – database ends the transaction in a consistent state (provided it started that way)

- **Isolation** – concurrently executing transactions must be unaware of each other (as if they ran serially)
  - It should look to one as if the other has not started or has already completed

- **Durability** – a transaction’s effects must persist in the database after it completes
Transactions in SQL

- **Explicit (within code)**
  - `begin transaction (txn)`
  - `end transaction (txn)`

- **Implicit (more common)**
  - Commit – complete a transaction / write its results to the database
  - Rollback – back out all effects of the transaction
  - Transaction implicitly begins when a program or database session starts
    - Commit or rollback end this transaction and (implicitly start another one)
  - If part of a transaction fails, it must be explicitly rolled back in the code
  - Autocommit – each (DML) SQL statement in the program / session treated as an individual transaction and committed upon completion
Transaction States

- **Active** – from the time a transaction starts until it fails or reach its last statement
- **Partially committed** – last statement executed, but changes to database are not yet permanent (SQL commit)
- **Committed** – changes to database have been made permanent
- **Failed** – logic error or user abort has precluded completion, and transaction’s changes must be undone (SQL rollback)
- **Aborted** – all effects of the transaction have been removed
Schedules

• Transaction consists of a set of read and write operations
  • Other computations as well, but reads and writes are critical, since they are the means that one transaction interacts with another

• For two or more concurrent transactions, the relative sequence of their read and write operations constitutes a *schedule*

• Example: simultaneous $50 deposit to and $100 withdrawal from a checking account
  • In SQL, these two transactions might look like this
    • `update checking_account
      set balance = balance + 50
      where account_no = :acct`
    • `update checking_account
      set balance = balance – 100
      where account_no = :acct`
  • Each update statement actually consists of a read and a write operation
## Possible Schedules (1)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Deposit ($T_1$)</th>
<th>Withdrawal ($T_2$)</th>
<th>Final Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>read(1000) write(1050)</td>
<td>read(1050) write(950)</td>
<td>950</td>
</tr>
<tr>
<td>$S_2$</td>
<td>read(1000) write(1050)</td>
<td>read(1000) write(900)</td>
<td>900</td>
</tr>
<tr>
<td>$S_3$</td>
<td>read(1000) write(1050)</td>
<td>read(1000) write(900)</td>
<td>1050</td>
</tr>
</tbody>
</table>
## Possible Schedules (2)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Deposit ($T_1$)</th>
<th>Withdrawal ($T_2$)</th>
<th>Final Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_4$</td>
<td>read(900) 2rite(950)</td>
<td>read(1000) write(900)</td>
<td>950</td>
</tr>
<tr>
<td>$S_5$</td>
<td>read(1000) write(1050)</td>
<td>read(1000) write(900)</td>
<td>1050</td>
</tr>
<tr>
<td>$S_6$</td>
<td>read(1000) write(1050)</td>
<td>read(1000) write(900)</td>
<td>900</td>
</tr>
</tbody>
</table>
Serial Schedules

- The schedules which yield the correct result are both \textit{serial}
  - One transaction is executed in its entirety before the other starts
  - Serial schedules always lead to consistent results
    - Non-serial schedules can sometimes also yield consistent results, but determining this is not always algorithmically feasible
- To preserve data integrity, ensure that a schedule of concurrent operations is \textit{serializable} – equivalent to some serial schedule
Equivalence of Schedules

- Two schedules are considered *equivalent* if operations in one schedule can be rearranged into another schedule
  - Without altering the resulting computation

- Example:
  - $S_1$ can be converted to $S_2$
  - Swap write(A) and read(B) operations

- Note that operations in the same schedule cannot be interchanged
  - Could lead to changes in transaction’s computation

<table>
<thead>
<tr>
<th>Schedule</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>read A</td>
<td>read B</td>
</tr>
<tr>
<td></td>
<td>write A</td>
<td>write B</td>
</tr>
<tr>
<td>$S_2$</td>
<td>read A</td>
<td>read B</td>
</tr>
<tr>
<td></td>
<td>write A</td>
<td>write B</td>
</tr>
</tbody>
</table>
Conflicting Operations between Transactions

- Two operations in two different transactions *conflict* if
  - They access the same data item (same column value in a single record)
    - Not same column in different records
    - Not different columns in same record
  - At least one of the operations is a write
  - Changing the relative order of two conflicting operations can result in different final outcomes

- Examples:
  - Schedules 1, 2, and 3 have conflicting operations – switching operations would lead to different outcomes
  - Schedules 4 and 5 do not have operations in conflict – no writes

<table>
<thead>
<tr>
<th>Schedule</th>
<th>T_1</th>
<th>T_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td>write A</td>
<td>read A</td>
</tr>
<tr>
<td>S_2</td>
<td>read A</td>
<td>write A</td>
</tr>
<tr>
<td>S_3</td>
<td>write A</td>
<td>write A</td>
</tr>
<tr>
<td>S_4</td>
<td>read A</td>
<td>read A</td>
</tr>
<tr>
<td>S_5</td>
<td>read A</td>
<td>read A</td>
</tr>
</tbody>
</table>
Conflict Equivalence

- Two schedules $S_1$ and $S_2$ on the same set of transactions are conflict equivalent if one can be transformed into the other by a series of interchanges of non-conflicting operations.

- Examples
  - $S_1$ and $S_2$ are conflict equivalent
    - Access different data items
  - $S_3$ and $S_4$ are not conflict equivalent

- A schedule is conflict serializable if there is a serial schedule to which it is equivalent.
Two schedules $S_1$ and $S_2$ on the same set of transactions are view equivalent if
- Some transaction in both schedules reads the initial value of the same data item
- If in $S_1$ some transaction reads a data item that was written by another transaction, the same holds for the two transactions in $S_2$
- If a transaction does the last write to some data item in $S_1$, it also does the last write to the same data item in $S_2$

This is less strict than conflict equivalence
- Requires that two schedules have the same outcome, but are not necessarily conflict equivalent
- Conflict equivalence implies view equivalence, but not vice versa

A schedule is view serializable if it is view equivalent to some serial schedule
Equivalence ≠ Producing the Same Result

- Two equivalent schedules (by either standard) will always produce the same final results
  - But not vice versa

- Example: from account deposit and withdrawal schedules
  - $S_1$ and $S_2$ produce same result, but are not equivalent

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Deposit ($T_1$)</th>
<th>Withdrawal ($T_2$)</th>
<th>Final Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>read(1000) write(1050)</td>
<td>read(1050) write(950)</td>
<td>950</td>
</tr>
<tr>
<td>$S_4$</td>
<td>read(900) write(950)</td>
<td>read(1000) write(900)</td>
<td>950</td>
</tr>
</tbody>
</table>
Testing for Serializability

- To ensure correctness of concurrent operations, ensure that the schedule followed is serializable.

- Want to test a schedule for serializability:
  - Can be very expensive to test for view serializability,
  - More feasible to test for conflict serializability.
Construct a *precedence graph* of a schedule to test it for conflict serializability
- Each transaction is a node on the precedence graph
- There is a directed edge between two transactions if there are conflicting operations between them – that is, at least one of the following occurs
  - $T_1$ reads an item before $T_2$ writes it
  - $T_1$ writes an item before $T_2$ reads it
  - $T_1$ writes an item before $T_2$ writes it
- If the resulting graph contains a cycle, the schedule is not conflict serializable
- If there are no cycles, then any topological sorting of the precedence graph will give an equivalent serial schedule
Consider $S_2$ from the deposit/withdrawal schedules.

- $T_1$ must do its read before $T_2$ does its write.
- $T_2$ must do its read before $T_1$ does its write.

Yields a cyclical precedence graph.
- $S_2$ is not serializable.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Deposit ($T_1$)</th>
<th>Withdrawal ($T_2$)</th>
<th>Final Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_2$</td>
<td>read(1000)</td>
<td>read(1000)</td>
<td>900</td>
</tr>
<tr>
<td></td>
<td>write(1050)</td>
<td>write(900)</td>
<td></td>
</tr>
</tbody>
</table>
Precedence Graph Example 2

Consider a transfer of $50 from a savings (with a $2000 starting balance) to a checking (instead of a $50 deposit) that occurs at the same time as a $100 checking withdrawal via the following schedule.

<table>
<thead>
<tr>
<th>Transfer ($T_1$)</th>
<th>Withdrawal ($T_2$)</th>
<th>Final Balances</th>
</tr>
</thead>
<tbody>
<tr>
<td>read savings (2000)</td>
<td>read checking (1000)</td>
<td>1950 (savings)</td>
</tr>
<tr>
<td>write savings (1950)</td>
<td>write checking (900)</td>
<td>950 (checking)</td>
</tr>
<tr>
<td>read checking (900)</td>
<td>write checking (950)</td>
<td></td>
</tr>
<tr>
<td>write checking (950)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note the following conflicting operations in this schedule:
- $T_2$ must do its (checking) read before $T_1$ does its (checking) write.
- $T_1$ reads the (checking) value written by $T_2$. 
Precedence Graph Example 2 (Continued)

- Yields this precedence graph
  - Acyclic – indicates a serializable schedule
  - T₂ can be done before T₁
  - Leads to the following conflict equivalent serial schedule

<table>
<thead>
<tr>
<th>Transfer (T₁)</th>
<th>Withdrawal (T₂)</th>
<th>Final Balances</th>
</tr>
</thead>
<tbody>
<tr>
<td>read savings (2000)</td>
<td>read checking (1000)</td>
<td>1950 (savings)</td>
</tr>
<tr>
<td>write savings (1950)</td>
<td>write checking (900)</td>
<td>950 (checking)</td>
</tr>
<tr>
<td>read checking (900)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write checking (950)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transaction Recoverability

• Schedules must not only serializable, but *recoverable*
  • Unrecoverable schedules can lead to inconsistencies
  • A transaction $T_2$ must not commit until any transaction $T_1$ which produces data used by $T_2$ commits
    • If $T_1$ fails, then $T_2$ must also fail

• Avoid *cascading rollback* – possibility of chain of failed transactions
  • $T_2$ reads data from $T_1$, $T_3$ reads data from $T_2$ $T_4$ reads data from $T_3$
  • If $T_1$ fails – $T_2$, $T_3$, and $T_4$ must also fail

• Producing only *cascadeless schedules* is desirable
  • No transaction $T_2$ is allowed to read a value written by another transaction $T_1$ until $T_1$ has fully committed
    • $T_2$ must wait until $T_1$ commits or fails (in which the previous value of the uncommitted item is used)
Exam 1
Crash Recovery
Causes of Data Corruption

• Logical errors related to incoming data
  • Aborted operations (both programmatic and interactive)

• Transaction failures (i.e. from rollback, deadlock, etc.)

• System crashes
  • Power failure
  • Hardware failure (i.e. failed CPU)
  • Software failure (i.e. operating system crash)
  • Network communication failure
  • Human error
  • Security breach or cyber-attack

• Disk failures that destroy the medium storing the data

• External catastrophes (i.e. fire, flood, etc.)
Storage Types and Data Loss

- Volatile storage – main memory
  - Subject to data loss at any time from many factors (i.e. power, hardware, software failure, etc.)

- Non-volatile storage – disk
  - Not as prone to data corruption
  - Still susceptible to power failures during writes, disk failures, and external catastrophes

- “Stable” storage – approaches immunity to data loss
  - Write-once media (i.e. CDs, DVDs, etc.)
  - Duplication of data (i.e. RAID, remote backup)
Approaches to Data Protection

• Regular system backups
  • Protect data against non-volatile storage failure and some inadvertent data erasure (i.e. human error)
  • Fairly rare occurrences
  • System backups are essential but not enough
  • Need fast restoration of changes since the last backup

• Crash Recovery Measures
  • Restore the system to a consistent state after an aborted operation or crash that does not involve non-volatile media failure
  • Ensure the durability property of transactions – that commits “stick”
  • Each transaction assigned a unique identifier (i.e. serial number)
  • Keep some record of incoming transactions
  • Deal with in-process transactions when the system failed
Transaction Processing Log

- Keeps track of what each transaction is doing
  - Transaction start
  - Details of changes the transaction makes to the database
  - Transaction end messages
    - Commit entry indicates successful completion of a transaction – all of its changes to the database should persist
    - Abort entry indicates the transaction failed – none of its changes should be allowed to remain
    - Neither a commit nor an abort entry will be present in the log if the system crashes while a transaction is in process
      - No changes that the transaction has made to the database should persist when the crash recovery is complete
      - If possible, the transaction can be restarted once the database is restored to a consistent state
- Can also be used for database replication
Protecting the Log

- The transaction processing log needs to be protected against corruption
  - Writing it to stable storage
  - Keep multiple copies of the log in different locations

- Ensure the log data is written before the actual changes are written to the database
  - System typically buffers log entries until a block of them can be written
    - Actual database updates written after the log buffer is flushed
    - Sometimes it might be necessary to write out data block before the logging block is full
      - This leads to a forced write of a partial log buffer

- Ensure that a crash that occurs while the log block is being written does not corrupt previous log entries
Crash Control Schemes

- **Incremental Log with Deferred Updates**
  - No changes are made to the database until after the transaction commits and the commit entry is written to the log.

- **Incremental Log with Immediate Updates**
  - Changes are made to the database during the transaction, but only after a log entry is written that includes the initial values of the things changed (so they can be recovered if necessary).

- **Shadow Paging**
  - Two copies of the relevant database data are kept during the transaction – both original and modified values. Once the transaction commits, the modified values permanently replace the original ones. (No log required.)
Incremental Log with Deferred Updates

- Example: A transaction to transfer $50 from checking to savings (with initial balances of $1000 and $2000, respectively).

<table>
<thead>
<tr>
<th>SQL</th>
<th>Log Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>update checking_accounts</td>
<td>T1234 starts</td>
</tr>
<tr>
<td>set balance = balance – 50</td>
<td>T1234 writes 950 to balance of</td>
</tr>
<tr>
<td>where account_no = :caccount;</td>
<td>checking_accounts record 127</td>
</tr>
<tr>
<td>update savings_accounts</td>
<td>T1234 writes 2050 to balance of</td>
</tr>
<tr>
<td>set balance = balance + 50</td>
<td>savings_accounts record 253</td>
</tr>
<tr>
<td>where account_no = :saaccount;</td>
<td>T1234 commits</td>
</tr>
</tbody>
</table>

- Once transaction partially commits (e.g. commit log entry is written, actual updates to the database occur
  - If the transaction fails or aborts, no changes have been made to the database
Deferred Updates and Crash Recovery

- If the system crashes during a transaction,
  - If the crash occurs before the commit log entry is written, it can be restarted or ignored when the system is restored
  - If the crash occurs after the commit log entry is written, each value specified to the log will be (re)written to the database
    - No harm in writing the same values to the database a second time

- This *redo log* approach has the following recovery algorithm
  - for each transaction with a start record in the log
    - If its commit record is also in the log
      - Write each new value for the transaction in the log to the database

- Checkpoint – periodic automated flush of buffers to disk
  - Causes committed transactions to be reflected in non-volatile storage
  - DBMS writes a checkpoint to the log
  - Only transactions after the checkpoint need to be applied after a crash
Immediate Log with Incremental Update

- Since database updates happen during the course of a transaction, log entries (written before the updates) must contain both old and new values.

<table>
<thead>
<tr>
<th>SQL</th>
<th>Log Entries</th>
</tr>
</thead>
</table>
| update checking_accounts  
  set balance = balance – 50  
  where account_no = :caccount; | T1234 starts  
  T1234 writes 950 to balance of  
  checking_accounts record 127  
  (old value was 1000) |
| update savings_accounts  
  set balance = balance + 50  
  where account_no = :saaccount; | T1234 writes 2050 to balance of  
  savings_accounts record 253  
  (old value was 2000)  
  T1234 commits |

- If the transaction fails or aborts, all database updates must be undone by writing the original values back to the database.
Immediate Update and Crash Recovery

- **Redo** and **undo** log approach to crash recovery
  - for each transaction with a start record in the log
  - if its commit record is also in the log
    - write each new value for the transaction in the log to the database (redo)
  - else
    - rewrite each old value for the transaction in the log to the database (undo)

- **Order is critical here**
  - Undo operations must happen first (from newest to oldest)
  - Redo operations can happen afterward (from oldest to newest)

- Checkpoints can be used to minimize undo/redo work
Incremental Update Tradeoffs

- Incremental update has more overhead than deferred update
  - Longer log entries – both old and new values stored
  - Failed transactions have to be “cleaned up”
  - Crash recovery requires processing every transaction, not just the ones that committed
  - Every database write requires the corresponding log entry to be written to disk/stable storage (not just on commit)

- Allows changes made by transactions to the database to become visible more quickly
  - Useful in bulk writes – can see updates as they occur
    - i.e. adding monthly interest to all savings accounts
Shadow Paging

- Maintain two copies of the active portion of the database
  - Current version – reflects all changes since start of current transaction
  - Shadow version – state of database before current transaction began
- If transaction fails or aborts, current version is discarded
- If transaction commits, current version replaces shadow version
- Crash recovery is automatic – since changes are only made to the current version, simply revert to the shadow version