Database Transaction Processing

Tyler Bletsch
Duke University

Slides are adapted from Brian Rogers (Duke)
Transaction Processing Systems

- Systems with large DB’s; many concurrent users
  - As a result, many concurrent database transactions
  - E.g. Reservation systems, banking, credit card processing, stock markets, supermarket checkout
- Need high availability and fast response time
- Concepts
  - Concurrency control and recovery
  - Transactions and transaction processing
  - ACID properties (desirable for transactions)
  - Schedules of transactions and recoverability
  - Serializability
  - Transactions in SQL
Single-User vs. Multi-User

- DBMS can be single-user or multi-user
  - How many users can use the system concurrently?
  - Most DBMSs are multi-user (e.g. airline reservation system)

- Recall our concurrency lectures (similar issues here)
  - Multiprogramming
  - Interleaved execution of multiple processes
  - Parallel processing (if multiple processor cores or HW threads)

Interleaved concurrency is model we will assume
Transactions

- Transaction is logical unit of database processing
  - Contains $\geq 1$ access operation
  - Operations: insertion, deletion, modification, retrieval
    - E.g. things that happen as part of the queries we’ve learned

- Specifying database operations of a transaction:
  - Can be embedded in an application program
  - Can be specified interactively via a query language like SQL
  - May mark transaction boundaries by enclosing operations with:
    - “begin transaction” and “end transaction”

- Read-only transaction:
  - No database update operations; only retrieval operations
Database Model for Transactions

• Database represented as collection of named data items
  – Size of data item is its “granularity”
  – E.g. May be field of a record (row) in a database
  – E.g. May be a whole record (row) or table in a database

• Database access operations can include:
  – read_item(X): read database item named X into a program variable (assume program variable also named X)
  – write_item(X): write value of program variable X into database item named X
Read & Write Commands

• read_item(X)
  1. Find address of disk block containing item X
  2. Copy disk block into a buffer in memory (if not already there)
  3. Copy item X from memory buffer to program variable named X

• write_item(x)
  1. Find address of disk block containing item X
  2. Copy disk block into a buffer in memory (if not already there)
  3. Copy item X from the program variable named X into memory
  4. Store updated block from memory buffer back to disk
    • At some point; does not need to be immediately
    • This is where database is actually updated
• Two example transactions: T1, T2
• Read-set: T1={X,Y}, T2={X}
• Write-set: T1={X,Y}, T2={X}
• **Three problems** can occur with concurrent transactions if executed in an uncontrolled manner:
  1. Lost Update Problem
  2. Temporary Update (Dirty Read) Problem
  3. Incorrect Summary Problem

• We’ll use example of an airline reservation database
  – Record (row) is stored for each airline flight
  – One record field is the number of reserved seats
    • A named data item
Lost Update Problem

T1
-----------------------
read_item(X);
X=X-N;
write_item(X);
read_item(Y);
Y=Y+N;
write_item(Y);

T2
-----------------------
read_item(X);
X=X+M;
write_item(X);

• T1 transfers N reservations from flight X to flight Y
• T2 reserves M new seats on flight X
• Update to flight X from T1 is lost!
  – Similar to our concurrency examples
Temporary Update Problem

- Transaction T1 fails for some reason
- DBMS must **undo** T1; change X back to its original value
- But T2 has already read the temporarily updated value of X
- Value T2 read is **dirty data**
  - Created by transaction not yet completed and committed
Incorrect Summary Problem

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_item(X);</td>
<td>sum = 0;</td>
</tr>
<tr>
<td>X = X - N;</td>
<td>read_item(A);</td>
</tr>
<tr>
<td>write_item(X);</td>
<td>sum = sum + A;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>read_item(Y);</td>
<td>read_item(X);</td>
</tr>
<tr>
<td>Y = Y + N</td>
<td>sum = sum + X;</td>
</tr>
<tr>
<td>write_item(Y);</td>
<td>read_item(Y);</td>
</tr>
<tr>
<td></td>
<td>sum = sum + Y;</td>
</tr>
</tbody>
</table>

T3 reads X after N is subtracted but reads Y before N is added; summary result is off by N.

- One transaction is calculating an aggregate summary function
- Other transactions are updating records
- E.g. calculate total number of reservations on all flights
Recovery

• For each transaction, DBMS is responsible for either:
  – All ops in transaction complete; their effect is recorded in database
    OR
  – Transaction has no effect on database or any other transaction

• DBMS can’t allow some operations to apply and not others
  – This can happen if a transaction fails part of the way through its ops

• How can a failure happen?
  – Logical abort (we tried to reserve enough seats but there weren’t enough)
    ^ This one is common and mundane! Must support!
  – System crash (HW, SW, or network error during transaction exe)
  – Transaction or system error (e.g. integer overflow or divide by 0)
  – Local errors (e.g. data for the transaction is not found)
  – Concurrency control (discussed in a bit may abort transaction)
  – Disk failure (read or write malfunction due to disk crash)
  – Physical problems (power failure, natural disaster, ...)


Transaction Concepts

• Transaction is an atomic unit of work
  – All operations completed in entirety or none of them are done

• DBMS tracks when transaction starts, terminate, commit or abort
  – BEGIN_TRANSACTION: beginning of transaction execution
  – READ or WRITE: read or write ops on database items
  – END_TRANSACTION: specifies that READ and WRITE operations have completed in the transaction
    • DBMS may need to check whether the changes can be *committed*  
      – i.e. permanently applied to the database
    • Or whether transaction must be aborted
  – COMMIT_TRANSACTION: successful end of transaction
    • Changes (updates) can be safely committed to database
  – ABORT: unsuccessful end of transaction
    • Changes that may have been applied to database must be undone
- Transaction moves to active state right when it begins
- Transaction can issue read & write operations until it ends
- Transaction moves to partial committed state
  - Recovery protocols need to ensure absence of a failure
- Transaction has reached commit point; changes can be recorded in DB
- Transaction can be aborted & go to failed state
- Terminated state corresponds to transaction leaving system
- Transaction info maintained in DBMS tables; failed trans may be restarted
System Log

• Used to recover from failures that affect transactions
  – Track transaction operations that affect DB values
  – Keep log on disk so it is not affected except by catastrophic fails

• Log records (T is a unique transaction ID)
  – [start_transaction,T]
    • transaction T has started
  – [write_item,T,X,old_val,new_val]
    • transaction T has changed database item X from old_val to new_val
  – [read_item,T,X] (not strictly needed)
    • transaction T has read the value of item X
  – [commit,T]
    • transaction T has completed successfully, effects can be
  – [abort,T]
    • transaction T has been aborted
Transaction Commit Point

• “Commit point”
  – A point in time in which all operations that access the DB have executed successfully
  – Effect of all operations on the DB have been recorded in the log
  – Sometimes also called a “consistency point” or “checkpoint”

• Transaction said to be “committed”
  – Its effect assumed to be permanently recorded in the DB
  – Transaction writes a commit record \([\text{commit},T]\) to the log

• On a failure:
  – Search log for started but not committed transactions
    • Roll back their effects to undo their effects of updating the DB
  – Search for transactions that have written their commit record
    • Apply their write operations from the log to the DB
ACID Properties

• Transactions should possess ACID properties
  – These should be enforced by concurrency control & recovery methods of the DBMS
    • Atomicity
    • Consistent
    • Isolation
    • Durability
Atomicity

“Atomicity”:  
- Transaction is atomic unit of processing  
- It is performed entirely or not at all

Managed by the DBMS  
- As part of the transaction recovery subsystem  
- Requires executing every transaction (eventually) to completion  
- Partial effects of an aborted transaction must be undone
Consistency

- **“Consistency”:**
  - Complete execution of a transaction takes the database from one consistent state to another

- **Responsibility:**
  - Programmers of database programs
  - And/Or DBMS module that enforces integrity constraints

- **Database State**
  - Collection of all stored data items in the DB at a given point in time
  - Consistent state satisfies all constraints of the schemas
  - DB program should be written to guarantee this
Isolation

• “Isolation”:  
  – Transaction appears as if executed in isolation from other transactions (no interference)

• Enforced by the “concurrency control” subsystem of DBMS  
  – E.g. a transaction only makes its updates visible to other transactions after it commits  
  – There are many options for these types of protocols
Durability

• “Durability”:  
  – Changes applied to database by a committed transaction must be persistent (e.g. not lost due to any failure)

• Responsibility of recover subsystem of DBMS  
  – Also many options for recovery protocols
Schedule of Transactions

• Schedule of n transactions: T1, T2, ..., Tn
  – Ordering of operations of the transactions
  – Each operation is in-order within a given transaction, Ti
  – But operations may be interleaved between Ti and Tj

• Notation
  – read_item, write_item, commit, abort abbreviated as r, w, c, a
  – Transaction ID is subscript following the operation
  – E.g. Sₐ: r₁(X); r₂(X); w₁(X); r₁(Y); w₂(X); w₁(Y)
Complete Schedule

• Conflicting operations:
  1. Belong to different transactions
  2. Access the same named item X
  3. At least one of the operations is a write_item(X)

• Schedule $S$ of $n$ transactions is complete schedule if:
  1. Operations in $S$ are exactly the operations in $T_1, T_2, ..., T_n$, with a commit or abort operation as the last op for each transaction
  2. Any pair of ops from same transaction $T_i$ appear in order
  3. For any 2 conflicting ops, one must occur before the other (i.e., order is explicit)

• Trivially correct schedule: serial schedule
  - When all transactions are done strictly in order with no interleaving
  - Definitely correct, but this kills performance
  - We want this property but also to allow some interleaving...
• One strategy: **Recovery.**
  - For schedules where transactions commit only after all transactions whose changes they read have committed.
  - If first transaction aborts, then we can abort second transaction.
  - Ability to do this depends on the schedule, some are not recoverable
    - We can characterize schedules for which recovery is possible
  - For recoverable schedules there may be a variety of algorithms

• Recoverable schedules:
  - Once a trans T is committed, should never be necessary to undo it
  - If no trans T in S commits until all transactions T’ that have written an item that T reads have committed
    - If this is not true, then the schedule is nonrecoverable

• Example of recoverable schedule
  - \( S_a: r_1(X); w_1(X); r_2(X); r_1(Y); w_1(Y); c_1; w_2(X); c_2; \)
Non-Recoverable Schedule

• Example:
  – $S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_2; a_1$
  – Non-recoverable because $T2$ reads $X$ from $T1$; $T2$ commits before $T1$ commits; what if $T1$ aborts?
    • Value $T2$ read for $X$ is no longer valid; it needs to abort as well

• Examples of making previous schedule recoverable:
  – $S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$
  – $S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2$

• Summary of above:
  – $r_1(X); w_1(X); r_2(X); r_1(Y); w_1(Y); c_1; w_2(X); c_2; w_2(X); c_2$ Recoverable (from last slide)
  – $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_2; a_1$ Unrecoverable (top of slide)
  – $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$ Fix #1
  – $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2$ Fix #2
Cascading Rollback

- **“Cascading Rollback”:**
  - When an *uncommitted* transaction needs to rollback because it read an item from a transaction that failed
  - E.g. $S_e$ from previous slide

- This can be costly; thus important to characterize schedules where this is guaranteed not to occur
  - Called a *cascadeless schedule*
  - If every transaction only reads items that were written by already committed transactions
  - E.g., $r_1(A); w_1(A); r_2(A); w_2(A); c_1; c_2$ becomes $r_2(A); r_1(A); w_1(A); w_2(A); a_1; c_2$

- Final type of schedule: **strict schedule**
  - Transactions cannot read or write $X$ until last transaction that write $X$ has committed (or aborted)
  - Even more restrictive than cascadeless (eases recovery)
  - E.g., $r_1(A); w_1(A); r_2(A); w_2(A); c_1; c_2$ becomes $r_1(A); w_1(A); c_1; r_2(A); w_2(A); c_2$
• Another strategy: **serializability**

• Characterize types of schedules considered correct...  
  – Even when concurrent transactions are executing!

• Consider two transactions T1 and T2  
  – E.g. the airline reservation transactions we looked at earlier  
  – If no operation interleaving is possible then two outcomes:  
    • Execute all of T1 then all of T2  
    • Or execute all of T2 then all of T1  
  – If operation interleaving is possible then many possible orderings

• **Serializability of schedules:**  
  – Used to identify which schedules are correct when transaction executions have interleaving operations
Serial Schedule

“Serial Schedule”:

• All operation of each transaction executed consecutively
• No interleaving
• Formally:
  – If for every transaction T in the schedule, all operations of T are executed consecutively in the schedule
  – Commit or abort of a transaction signals start of next transaction
  – Otherwise the schedule is nonserial
• Easy to reason about correctness, but...
  – Problem with serial schedules is performance
  – Limited concurrency
    • What if one operation requires a slow I/O operation?
A schedule $S$ of $n$ transactions is **Serializable** if:

- Results are equivalent to *some* serial schedule of the same $n$ transactions

Saying that a nonserial schedule $S$ is serializable is equivalent to saying that it is correct.
Example: NOT a serializable schedule

Recall our Lost Update problem
- Assume $X=90$ and $Y=90$ at start; $N=3$ and $M=2$
- We’d expect $X=89$ and $Y=93$ in database at end
- In this interleaving we end up with $X=92$ and $Y=93$
  BROKEN!
Example: Serializable schedule

T1
read_item(X);
X=X-N;
write_item(X);
read_item(Y);
Y=Y+N;
write_item(Y);

T2
read_item(X);
X=X+M;

write_item(X);

• This is a serializable schedule
• Would be allowed by the DBMS
• Non-serializable schedules can be aborted before commit
Conclusion

• We want parallelism in our database and we want ACID properties
  ▪ But we’re accessing shared data, so conflicts arise

• Simple mutex too expensive

• Unlike simple RAM (like the malloc assignment), our data is structured, so we can reason about how we interleave operations

• Database schedules operations to ensure correctness
• Tradeoffs exist between performance and cost/correctness of recovery in exceptional circumstances