

Database Transaction Processing

ECE 650
Systems Programming & Engineering
Duke University, Spring 2016

Transaction Processing Systems

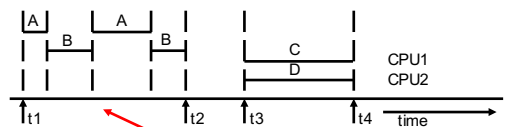
- Systems with large DB's; many concurrent users
 - As a result, many concurrent database transactions
 - E.g. Reservation systems, banking, credit card processing, stock markets, supermarket checkout
- Need high availability and fast response time
- Concepts
 - Concurrency control and recovery
 - Transactions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

ECE 650 – Spring 2016

2

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)



ECE 650 – Spring 2016

Interleaved concurrency is model we will assume

Transactions

- Transaction is logical unit of database processing
 - Contains ≥ 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

ECE 650 – Spring 2016

4

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

ECE 650 – Spring 2016

5

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

ECE 650 – Spring 2016

6

Example

T1	T2
read_item(X);	read_item(X);
X=X-N;	X=X+M;
write_item(X);	write_item(X);
read_item(Y);	
Y=Y+N;	
write_item(Y);	

- Two example transactions: T1, T2
- Read-set: T1={X,Y}, T2={X}
- Write-set: T1={X,Y}, T2={X}

Concurrency Control Motivation

- Three problems can occur with concurrent transactions
 - If executed in an uncontrolled manner
- 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
- Problems
 - Lost Update Problem
 - Temporary Update (Dirty Read) Problem
 - Incorrect Summary Problem

Lost Update Problem

T1	T2
read_item(X);	
X=X-N;	
	read_item(X);
	X=X+M;
write_item(X);	
read_item(Y);	write_item(X);
Y=Y+N;	
write_item(Y);	

- 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

T1	T2
read_item(X);	
X=X-N;	
write_item(X);	
	read_item(X);
	X=X+M;
	write_item(X);
read_item(Y);	
...	

- 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

T1	T3
	sum = 0;
	read_item(A);
	sum=sum+A;
read_item(X);	
X=X-N;	
write_item(X);	
	read_item(X);
	sum=sum+X;
	read_item(Y)
	sum=sum+Y;
read_item(Y);	
Y=Y+N	
write_item(Y);	

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

- One trans is calculating an aggregate summary function
- Other trans 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?
 - 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, ...)

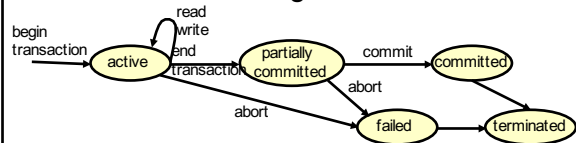
Transaction Concepts

- Transaction is an atomic unit of work
 - All operations completed in entirety or none of them are done
- DBMS tracks when trans 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

ECE 650 – Spring 2016

13

State Transition Diagram



- 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

ECE 650 – Spring 2016

14

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 recorded[abort, T]
 - transaction T has been aborted

ECE 650 – Spring 2016

15

Transaction Commit Point

- Definition of commit point
 - All operations that access the DB have executed successfully
 - Effect of all operations on the DB have been recorded in the log
- Transaction said to be "committed"
 - Its effect assumed to be permanently recorded in the DB
 - Transaction writes a commit record [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

ECE 650 – Spring 2016

16

ACID Properties

- Transactions should possess ACID properties
 - These should be enforced by concurrency control & recovery methods of the DBMS
 - Atomicity
 - Consistency preservation
 - Isolation
 - Durability

ECE 650 – Spring 2016

17

Atomicity

- Definition
 - 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

ECE 650 – Spring 2016

18

Consistency Preservation

- Definition:
 - 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

- Definition
 - 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

- Definition
 - 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: T_1, T_2, \dots, T_n
 - Ordering of operations of the transactions
 - Each operation is in-order within a given transaction, T_i
 - But operations may be interleaved between T_i and T_j
- Notation
 - read_item, write_item, commit, abort abbreviated as r, w, c, a
 - Transaction ID is subscript following the operation
 - E.g. $S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y)$

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 of n transactions is complete schedule if:
 1. Operations in S are exactly the operations in T_1, T_2, \dots, 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

Schedule Recoverability

- Need to recover from transaction failures
 - Easy for some schedules; very difficult for others
 - 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 and 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:
 - $S_0: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1$
 - 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:
 - $S_0: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$
 - $S_0: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2$

Cascading Rollback

- Description:
 - When an "uncommitted" transaction needs to rollback because it read an item from a transaction that failed
 - E.g. S_0 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 there were written by already committed transactions
- 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)

Serializability of Schedules

- Characterize types of schedules considered correct...
 - 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

- 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?

Example

T1	T2
read_item(X);	
X=X-N;	
	read_item(X);
	X=X+M;
write_item(X);	
read_item(Y);	write_item(X);
Y=Y+N;	
write_item(Y);	

- 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 $Y=93$

Serializable Schedule

- 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 – Serializable Interleaving

T1	T2
read_item(X);	
X=X-N;	
write_item(X);	
read_item(Y);	
	read_item(X);
	X=X+M;
Y=Y+N;	
write_item(Y);	
	write_item(X);

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