

# Databases Illuminated

## Chapter 10

### Transaction Management

# Protecting the Database During Transactions

- **Recovery**-restoring the database to a correct state after a failure
- **Concurrency control**-allows simultaneous use of the database without having users interfere with one another
- Both protect the database

# Steps in a Transaction

- Simple update of one record:
  - Locate the record to be updated
  - Bring the block into the buffer
  - Write the update to buffer
  - Write the modified block out to disk
- More complicated transactions may involve several updates
- Modified buffer block may not be written to disk immediately after transaction terminates-must assume there is a delay before actual write is done

# Basic Ideas About Transactions

- **Transaction**- a logical unit of work that takes the database from one consistent state to another
- Transactions can terminate successfully and **commit** or unsuccessfully and be **aborted**
- Aborted transactions must be undone (**rolled back**) if they changed the database
- Committed transactions cannot be rolled back
- See **Figure 10.2** - transaction state diagram

# ACID Properties of Transactions

- **Atomicity**
  - Single “all or none” unit; entire set of actions carried out or none are
  - DBMS must roll back-UNDO- transactions that will not be able to complete successfully
  - **Log** of transactions’ writes used in the rollback process
- **Consistency**
  - Users responsible for ensuring that each transaction, executed individually, leaves the database in a consistent state
  - Concurrency control subsystem must ensure this for multiple transactions
- **Isolation**
  - When transactions execute simultaneously, DBMS ensures that the final effect is as if the transactions were executed one after another (serially)
- **Durability**
  - Effect of any committed transaction is permanently recorded in the database, even if the system crashes before all its writes are made to the database
  - Recovery subsystem must guarantee durability

# Concurrency Problems

- Concurrency control needed when transactions are permitted to process simultaneously, if at least one is an update
- Potential problems due to lack of concurrency control:
  - **Lost update problem- Figure 10.3**
  - **Uncommitted update problem- Figure 10.4**
  - **Inconsistent analysis problem- Figure 10.5**
  - **Non-repeatable read problem**-first transaction reads an item; second transaction writes a new value for the item; first transaction rereads the item and gets a different value
  - **Phantom data problem**- first transaction reads a set of rows; second transaction inserts a row; first transaction reads the rows again and sees the new row

# Conflict in Transactions

- If two transactions are only reading data items, they do not conflict and order is not important
- If two transactions operate on completely separate data items, they do not conflict and order is not important
- If one transaction writes to a data item and another either reads or writes to the same data item, then the order of execution is important
- Therefore, two operations **conflict** only if **all** of these are true
  - they belong to different transactions
  - they access the same data item
  - at least one of them writes the item

# Serial vs. Interleaved Execution

- **Interleaved execution** – control goes back and forth between operations of two or more transactions
- **Serial execution**- execute one transaction at a time, with no interleaving of operations. Ex. A, then B
  - Can have more than one possible serial execution for two or more transactions –Ex:A,B or B,A
  - For n transactions, there are n! possible serial executions
  - They may not all produce the same results
  - However, DB considers all serial executions to be correct
- See **Figure 10.6**

# Serializable Schedules

- A **schedule** is used to show the timing of the operations of one or more transaction
- Shows the order of operations
- Schedule is **serializable** if it produces the same results as if the transactions were performed serially in some order
- Objective is to find serializable schedules to maximize concurrency while maintaining correctness

# Conflict Serializability

- If schedule orders any conflicting operations in the same way as some serial execution the results of the concurrent execution are the same as the results of that serial schedule
- This type of serializability is called **conflict serializability**

# Precedence Graph

- Used to determine whether a schedule,  $S$ , is conflict serializable
- Draw a node for each transaction,  $T_1, T_2, \dots, T_n$ . For the schedule, draw directed edges as follows
  - If  $T_i$  writes  $X$ , and then  $T_j$  reads  $X$ , draw edge from  $T_i$  to  $T_j$
  - If  $T_i$  reads  $X$ , and then  $T_j$  writes  $X$ , draw edge from  $T_i$  to  $T_j$
  - If  $T_i$  writes  $X$ , and then  $T_j$  writes  $X$ , draw edge from  $T_i$  to  $T_j$
- $S$  is **conflict serializable** if graph has **no cycles**
- If  $S$  is serializable, can use the graph to find an equivalent serial schedule by examining the edges
  - If an edge appears from  $T_i$  to  $T_j$ , put  $T_i$  before  $T_j$
  - If several nodes appear on the graph, you usually get a partial ordering of graph
  - May be several possible serial schedules

# Methods to Ensure Serializability

- **Locking**
- **Timestamping**
- Concurrency control subsystem is "part of the package" and not directly controllable by either the users or the DBA
- A **scheduler** is used to allow operations to be executed immediately, delayed, or rejected
- If an operation is delayed, it can be done later by the same transaction
- If an operation is rejected, the transaction is aborted but it may be restarted later

# Locks

- Transaction can ask DBMS to place locks on data items in the DB
- Lock prevents another transaction from modifying the object
- Transactions may wait until locks are released before their lock requests can be granted
- Objects of various sizes (DB, table, page, record, data item) can be locked.
- Size determines the fineness, or **granularity**, of the lock
- Lock implemented by inserting a flag in the object or by keeping a list of locked parts of the database
- Locks can be **exclusive** or **shared** by transactions
  - Shared locks are sufficient for read-only access
  - Exclusive locks are necessary for write access
- **Figure 10.8** – lock compatibility matrix

# Deadlock

- Often, transaction cannot specify in advance exactly what records it will need to access in either its **read set** or its **write set**
- **Deadlock**- two or more transactions wait for locks being held by each another
- Deadlock detection uses a **wait-for** graph to identify deadlock
  - Draw a node for each transaction
  - If transaction S is waiting for a lock held by T, draw an edge from S to T
- **Cycle** in the graph shows deadlock
- Deadlock is resolved by choosing a **victim**-newest transaction or one with least resources
- Should avoid always choosing the same transaction as the victim, a situation called **starvation**, because that transaction will never complete

# Two-phase locking protocol

- A protocol that guarantees serializability
- Every transaction acquires all its locks before releasing any, but not necessarily all at once
- Transaction has two phases:
  - In **growing** phase, transaction obtains locks
  - In **shrinking** phase, it releases locks
- Once it enters its shrinking phase, can never obtain a new lock
- For **standard two-phase locking**, the rules are
  - Transaction must acquire a lock on an item before operating on the item. For read-only access, a shared lock is sufficient. For write access, an exclusive lock is required.
  - Once the transaction releases a single lock, it can never acquire any new locks
- Deadlock can still occur

# Cascading Rollbacks

- **Cascading rollback.**
  - Locks can be released before COMMIT in standard two-phase locking protocol
  - An uncommitted transaction may be rolled back after releasing its locks
  - If a second transaction has read a value written by the rolled back transaction, it must also roll back, since it read **dirty data**
- **Avoiding cascading rollback**
  - **Strict** two phase locking: transactions hold their exclusive locks until COMMIT, preventing cascading rollbacks
  - **Rigorous** two-phase locking: transactions hold all locks, both shared and exclusive, until COMMIT

# Lock Upgrading and Downgrading

- Transaction may at first request shared locks-allow other transactions concurrent read access to the items
- When transaction ready to do an update, requests that the shared lock be **upgraded**, converted into an exclusive lock
- Upgrading can take place only during the growing phase, and may require that the transaction wait until another transaction releases a shared lock on the item
- Once an item has been updated, its lock can be **downgraded**, converted from exclusive to shared mode
- Downgrading can take place only during the shrinking phase.

# Intention Locking

- Can represent database objects as a hierarchy by size
- Root node is the entire DB, level 1 nodes tables, level 2 nodes pages, level 3 nodes records, level 4 data items
- If a node is locked, all its descendants are also locked
- If a second transaction requests an incompatible lock on the **same** node, system knows that the lock cannot be granted
- If second transaction requests a lock on any descendant, system checks to see if any of its **ancestors** are locked before deciding whether to grant the lock
- If a transaction requests a lock on a node when a **descendant** is already locked, we don't want to search too much to determine this
- Need an **intention** lock, which may be shared or exclusive-shows some descendant is probably locked
- Two-phase protocol is used
  - No lock can be granted once any node has been unlocked
  - No node may be locked until its parent is locked by an intention lock
  - No node may be unlocked until all its descendants are unlocked
- Apply locking from the root down, using intention locks until the node is reached, and release locks from leaves up
- Deadlock is still possible

# Timestamping

- Each transaction has a timestamp; gives the relative order of the transaction
- Timestamp could be clock reading or logical counter
- Each data item has
  - a **Read-Timestamp**-timestamp of last transaction that read the item
  - **Write-Timestamp**-timestamp of last transaction that wrote the item
- Problems
  - Transaction tries to read an item already updated by a younger transaction (late read)
  - Transaction tries to write an item already updated by a later transaction (late write)
- Protocol takes care of these problems by rolling back transactions that cannot execute correctly

# Basic Timestamping Protocol

Compare  $TS(T)$  with  $WriteTimestamp(P)$  and/or  $ReadTimestamp(P)$  which identify the transaction(s) that last wrote or read the data item,  $P$

1. If  $T$  asks to **read**  $P$ , compare  $TS(T)$  with  $WriteTimestamp(P)$ 
  - (a) If  $WriteTimestamp(P) \leq TS(T)$  then proceed using the current data value and replace  $ReadTimestamp(P)$  with  $TS(T)$ . However, if  $ReadTimestamp(P)$  is already larger than  $TS(T)$ , just do the read and do not change  $ReadTimestamp(P)$
  - (b) If  $WriteTimestamp(P) > TS(T)$ , then  $T$  is late doing its read, and the value of  $P$  that it needs is already overwritten, so roll back  $T$
2. If  $T$  asks to **write**  $P$ , compare  $TS(T)$  with both  $WriteTimestamp(P)$  and the  $ReadTimestamp(P)$ 
  - (a) If  $WriteTimestamp(P) \leq TS(T)$  and  $ReadTimestamp(P) \leq TS(T)$ , do the write and replace  $WriteTimestamp(P)$  with  $TS(T)$
  - (b) else roll back  $T$ , assign a new timestamp, and restart  $T$

# Thomas' Write Rule

- Variation of the basic timestamping protocol that allows greater concurrency
- Applies when T is trying to write P, but new value has already been written for P by a younger transaction
- If a younger transaction has already **read** P, then it needed the value that T is trying to write, so roll back T and restart it
- Otherwise ignore T's write of P, and let T proceed

# Multi-versioning

- Concurrency can be increased if we allow multiple versions of data items to be stored
- Transactions can access the version that is consistent for them
- Data item  $P$  has a sequence of versions  $\langle P_1, P_2, \dots, P_n \rangle$ , each of which has
  - The content field, a value for  $P_i$ ,
  - Write-Timestamp( $P_i$ ), timestamp of transaction that wrote the value
  - Read-Timestamp( $P_i$ ), timestamp of youngest transaction that has read version  $P_i$
- When write( $P$ ) is done, a new version of  $P$  is created, with appropriate write-timestamp
- When read( $P$ ) is done, the system selects the appropriate version of  $P$ .

# Multi-version Timestamp Protocol

- When T does a read(P)
  - Value used is the value of the content field associated with the latest Write-Timestamp that is less than or equal to  $TS(T)$
  - Read-Timestamp is set to later of  $TS(T)$  or current value
- When T does a write(P)
  - Version used is the one whose write timestamp is the largest one that is less than or equal to  $TS(T)$
  - For that version
    - If  $Read-Timestamp(P) > TS(T)$ , P has already been read by a younger transaction, so roll back T, since it would be a late write
    - Else create a new version of P, with read and write timestamps  $TS(T)$

# Validation Techniques

- Also called **optimistic** techniques
- Assume that conflict will be rare
- Transactions proceed as if there were no concurrency problems
- Before a transaction commits perform check to determine whether a conflict has occurred
- If there is a conflict, the transaction must be rolled back
- Assume rollback will be rare
- Rollback is the price to be paid for eliminating locks
- No cascading rollbacks, since writes are to local copy only
- Allow more concurrency, since no locking is done

# Phases in Validation Techniques

- Transaction goes through two phases for read-only, three for updating:
  - **Read** phase, from transaction's start until just before it commits
    - reads all the variables it needs, stores them in local variables
    - Does any writes to a local copy of the data, not to the database
  - **Validation** phase, follows the read phase
    - Tests to determine whether there is any interference
    - For read-only transaction, checks to see that there was no error due to another transaction active when the data values were read. If no error, the transaction is committed. If interference occurred, the transaction is aborted and restarted
    - For a transaction that does updates, checks whether the current transaction will leave the database in a consistent state, with serializability. If not, the transaction is aborted..
  - **Write** phase, follows successful validation phase for update transaction
    - The updates made to the local copy are applied to the database

# Validation Phase

- Examines reads and writes of other transactions, T, that may cause interference
- Each other transaction, T, has three timestamps
  - Start(T), the relative starting time of the transaction
  - Validation(T), given at the end of its read phase as it enters its validation phase
  - Finish(T), its finishing time, the time it finished (including its write phase, if any)
- To pass the validation test, **one** of the following must be true:
  1. All transactions with earlier timestamps must have finished (including their writes) before the current transaction started **OR**
  2. If the current transaction starts before earlier one finishes, then **both** of these are true
    - a) the items written by the earlier transaction are not the ones read by the current transaction, **and**
    - b) the earlier transaction completes its write phase before the current transaction enters its validation phase
- Rule (a) guarantees that the writes of the earlier transaction are not read by the current transaction; rule (b) guarantees that the writes are done serially

# Need for Recovery

- Many different types of failures that can affect database processing
- Some causes of failure
  - Natural physical disasters
  - Sabotage
  - Carelessness
  - Disk malfunctions- result in loss of stored data
  - System crashes due to hardware malfunction-result in loss of main and cache memory
  - System software errors-result in abnormal termination or damage to the DBMS
  - Applications software errors

# Possible Effects of Failure

- Loss of main memory, including database buffers
- Loss of the disk copy of the database
- DBMS recovery subsystem uses techniques that minimize these effects

# Recovery Manager

- DBMS subsystem responsible for ensuring **atomicity** and **durability** for transactions in the event of failure
  - Atomicity-all of a transaction is performed or none
    - Recovery manager ensures that all the effects of committed transactions reach the database, and that the effects of any uncommitted transactions are undone
  - Durability-effects of a committed transaction are permanent
    - Effects must survive both loss of main memory and loss of disk storage

# Loss of Disk Data

- Handled by doing frequent backups-  
making copies of the database
- In case of disk failure, the backup can be brought up to date using a log of transactions

# System Failure

- If system failure occurs
  - Database buffers are lost
  - Disk copy of the database survives, but it may be incorrect
- A transaction can commit once its writes are made to the database buffers
- Updates made to buffer are not automatically written to disk, even for committed transactions
- May be a delay between commit and actual disk writing
- If system fails during this delay, we must ensure that these updates reach the disk copy of the database

# Recovery Log

- Contains records of each transaction showing
  - The start of transaction
  - Write operations of transaction
  - End of transaction
- If system fails, the log is examined to see what transactions to **redo** and/or what transactions to **undo**
- Several different protocols are used

# Deferred Update Protocol

- DBMS does all database writes in the log, and does not write to the database until the transaction is ready to commit
- Uses the log to protect against system failures :
  - When transaction starts, write a record of the form <T starts> to the log
  - When a write is performed, do not write the update to the database buffers or the database itself. Instead, write a log record of the form <T,X, n>
  - When a transaction is about to commit, write a log record of the form <T commits>, write all the log records for the transaction to disk, and then commit the transaction. Use the log records to perform the updates to the database buffers. Later, these updates pages will be written to disk
  - If the transaction aborts, simply ignore the log records for the transaction and do not perform the writes.
  - Called a **redo/no undo** method since we redo committed transactions and don't undo anything

# Checkpoints

- After a failure, we may not know how far back in the log to search for redo of transactions
- Can limit log searching using **checkpoints**
- Scheduled at predetermined intervals
- Checkpoint operations
  - Write modified blocks in the database buffers to disk
  - Write a checkpoint record to the log-contains the names of all transactions that are active at the time of the checkpoint
  - Write all log records now in main memory out to disk

# Using Checkpoint Records

- When a failure occurs, check the log
- If transactions are performed serially
  - Find the last transaction that started before the last checkpoint
  - Any earlier transaction would have committed previously and would have been written to the database at the checkpoint
  - Need only redo the one that was active at the checkpoint (provided it committed) and any subsequent transactions for which both start and commit records appear in the log
- If transactions are performed concurrently
  - Checkpoint record contains the names of **all** transactions that were active at checkpoint time
  - Redo all those transactions (if they committed) and all subsequent ones that committed

# Immediate Update Protocol

- Updates are applied to the database buffers as they occur and written to the database itself when convenient
- A log record is written first, since this is a **write-ahead** log protocol
- Protocol
  - When a transaction starts, write a record of the form <T starts> to the log
  - When a write operation is performed, write a log record with the name of the transaction, the field name, the old value, and the new value of the field. This has the form <T,X,o,n>
  - After writing log record, write the update to the database buffers
  - When convenient, write the log records to disk and then write updates to the database itself
  - When the transaction commits, write a record of the form <T commits> to the log

# Using the Immediate Update Log

- If a transaction aborts, use log to **undo** it, since it contains all the old values for the updated fields
  - Writes are undone in reverse order
  - Writing the old values means the database will be restored to its state prior to the start of the transaction
- If the system fails
  - In recovery, use the log to **undo** or **redo** transactions, making this a **redo/undo** protocol
  - For any transaction, T, for which both <T starts> and <T commits> records appear in the log, **redo** by using the log records to write the new values of updated fields-any write that did not actually reach the database will now be performed
  - For any transaction, S, for which the log contains an <S starts> record, but not an <S commits> record, need to **undo** - log records are used to write the old values of the affected fields, in reverse order

# Shadow Paging-Page Tables

- Alternative to logging
- DBMS has a **page table** with pointers to all current database pages
- Keeps both a **current page table** and a **shadow page table**, which are initially identical
- All modifications are made to the current page table- shadow table is left unchanged
- To modify a database page, system finds an unused page on disk, copies the old database page to the new one, and makes changes to the new page
- Updates the current page table to point to the new page

# Shadow Paging-Transaction End

- If the transaction completes successfully, current page table becomes the shadow page table
  - Write all modified pages from the database buffers to disk
  - Copy the current page table to disk
  - In the location on disk where the address of the shadow page table is recorded, write the address of the current page table, making it the new shadow page table
- If the transaction fails, new pages are ignored; shadow page table becomes the current page table

# ARIES Recovery Technique

- Flexible and conceptually simple method for recovery
- Each log record is given a unique **log sequence number** (LSN), assigned in increasing order
- Each log record records the LSN of the previous log record for the same transaction, forming a linked list
- Each database page has a **pageLSN**, the LSN of the last log record that updated it
- **Transaction table** has an entry for each active transaction, with the transaction identifier, the status (active, committed or aborted), and the **lastLSN**, the LSN of the latest log record for the transaction
- **Dirty page table** has an entry for each page in the buffer that has been updated but not yet written to disk, and the **recLSN**, the LSN of the oldest log record for any update to the buffer page
- Uses write-ahead logging-log record is written to disk before any database disk update
- Does check-pointing to limit log searching in recovery

# ARIES Recovery Protocol

- Tries to repeat history during recovery-repeats all database actions done before the crash, even those of incomplete transactions
- Does redo and undo as needed
- Three phases
  - **Analysis:** begins with most recent checkpoint record, reads forward in the log to identify which transactions were active at the time of failure; uses transaction table and dirty page table to determine which buffer pages contain updates not yet written to disk; determines how far back in the log it needs to go to recover, using the linked lists of LSNs
  - **Redo:** from starting point in the log identified during analysis, goes forward in the log, applies all the unapplied updates from the log records
  - **Undo:** going backwards from the end of the log, undoes updates done by uncommitted transactions, ending at the oldest log record of any transaction that was active at the time of the crash

# Oracle Transaction Management

- Multi-version concurrency control mechanism, with no read locks
- For read-only transactions, uses a consistent view of the database at the point in time when it began, including only those updates that were committed at that time
- Creates **rollback segments** that contain the older versions of data items-used for both read consistency and undo operations that may be needed
- Uses type of timestamp called a **system change number** (SCN) given to each transaction at its start

# Oracle Concurrency Control

- Several types of locks available, including both DML and DDL locks
- DDL locks are applied at the table level
- DML locks are at the row-level
- Uses a deadlock detection scheme, and rolls back one of the transactions if needed
- Provides two **isolation levels**, degrees of protection from other transactions
  - **Read committed**: default level-guarantees that each statement in a transaction reads only data committed before the statement started. Since data may be changed during the transaction, there may be non-repeatable reads and phantom data.
  - **Serializable**: gives transaction-level consistency-ensures that a transaction sees only data committed before the transaction started

# Oracle Recovery

- **Recovery manager** (RMAN) is a GUI tool that the DBA can use to control backup and recovery operations
- RMAN can make backups of the database or parts of it, backups of recovery logs, can restore data from backups, can perform recovery operations of redo, undo
- Maintains control files, rollback segments, redo logs, and archived redo logs
- When a redo log is filled, it can be archived automatically
- Can also provide a managed standby database
  - Copy of the operational database kept at another location
  - Takes over if the regular database fails
  - Kept nearly up to date by shipping the archived redo logs and applying the updates to the standby database