Concurrency Control & Recovery

Mrs Sangeeta Bhandari
Assistant Professor

Hans Raj Mahila Maha Vidyalaya,
Jalandhar

Time Stamp Method

- Assigns a global unique time stamp to each transaction
  - All database operations within the same transaction must have the same time stamp
- Produces an explicit order in which transactions are submitted to the DBMS
- Uniqueness
  - Ensures that no equal time stamp values can exist

Time Stamp Method

- Monotonicity
  - Ensures that time stamp values always increase
- Disadvantage
  - Each value stored in the database requires two additional time stamp fields – last read, last update

Time Stamp Method

- Wait/die
  - Older transaction waits and the younger is rolled back and rescheduled
- Wound/wait
  - Older transaction rolls back the younger transaction and reschedules it
- In the situation where a transaction is requests multiple locks, each lock request has an associated time-out value. If the lock is not granted before the time-out expires, the transaction is rolled back

Time Stamp Method

TABLE 9.12 WAIT/DIE AND WOUND/WAIT CONCURRENCY CONTROL SCHEMES

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
T1 (11548789)	T2 (19562545)	• T1 waits until T2 is completed and T2 releases its locks.	• T1 preempts (rolls back) T2. T2 is rescheduled using the same time stamp.
T2 (19562545)	T1 (11548789)	• T2 Dies (rolls back). • T2 is rescheduled using the same time stamp.	• T2 waits until T1 is completed and T1 releases its locks.

Concurrency Control

- Optimistic approach
  - Based on the assumption that the majority of database operations do not conflict
  - Does not require locking or time stamping techniques
  - Transaction is executed without restrictions until it is committed
  - Acceptable for mostly read or query database systems that require very few update transactions
  - Phases are read, validation, and write

Concurrency Control

- Phases are read, validation, and write
  - ***Read phase*** – transaction reads the database, executes the needed computations and makes the updates to a private copy of the database values.
    - All update operations of the transaction are recorded in a temporary update file which is not accessed by the remaining transactions
  - ***Validation phase*** – transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database
    - If the validation test is positive, the transaction goes to the writing phase. If negative, the transaction is restarted and the changes discarded
  - ***Writing phase*** – the changes are permanently applied to the database

Deadlocks

- Condition that occurs when two transactions wait for each other to unlock data
  - T1 needs data items X and Y; T needs Y and X
  - Each gets the its first piece of data but then waits to get the second (which the other transaction is already holding) – ***deadly embrace***
- Possible only if one of the transactions wants to obtain an exclusive lock on a data item
  - No deadlock condition can exist among *shared* locks

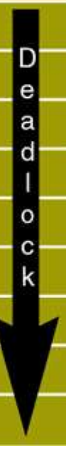
Deadlocks

- Control through
  - Prevention
  - Detection
  - Avoidance

Deadlock Condition

TABLE 9.11 How a Deadlock Condition Is Created

TIME	TRANSACTION	REPLY	LOCK STATUS		
0			Data X		Data Y
1	T1:LOCK(X)	OK	Unlocked		Unlocked
2	T2: LOCK(Y)	OK	Locked		Unlocked
3	T1:LOCK(Y)	WAIT	Locked		Locked
4	T2:LOCK(X)	WAIT	Locked	**Deadlock**	Locked
5	T1:LOCK(Y)	WAIT	Locked		Locked
6	T2:LOCK(X)	WAIT	Locked		Locked
7	T1:LOCK(Y)	WAIT	Locked		Locked
8	T2:LOCK(X)	WAIT	Locked		Locked
9	T1:LOCK(Y)	WAIT	Locked		Locked
...	.....	.....	.....		.....
...	.....	.....	.....		.....
...	.....	.....	.....		.....
...	.....	.....	.....		.....



Types of Failures

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data

Types of Failures

- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Types of Storage

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - but may still fail, losing data

Types of Storage

- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

Implementing Stable Storage

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated

Implementing Stable Storage

- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. we might unnecessarily redo transactions which have already output their updates to the database.

Checkpoints

- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
     - All updates are stopped while doing checkpointing

Recovery

- Database recovery
  - Restores database from a given state, usually inconsistent, to a previously consistent state
  - Based on the atomic transaction property
    - All portions of the transaction must be treated as a single logical unit of work, in which all operations must be applied and completed to produce a consistent database
  - If transaction operation cannot be completed, transaction must be aborted, and any changes to the database must be rolled back (undone)

Transaction Recovery

- The database recovery process involves bringing the database to a consistent state after failure.
- Transaction recovery procedures generally make use of deferred-write and write-through techniques

Transaction Recovery

- Deferred write
  - Transaction operations do not immediately update the physical database
  - Only the transaction log is updated
  - Database is physically updated only after the transaction reaches its commit point using the transaction log information
  - If the transaction aborts before it reaches its commit point, no ROLLBACK is needed because the DB was never updated
  - A transaction that performed a COMMIT after the last checkpoint is redone using the “after” values of the transaction log

Transaction Recovery

- Write-through
  - Database is immediately updated by transaction operations during the transaction's execution, even before the transaction reaches its commit point
  - If the transaction aborts before it reaches its commit point, a ROLLBACK is done to restore the database to a consistent state
    - A transaction that committed after the last checkpoint is redone using the "after" values of the log
    - A transaction with a ROLLBACK after the last checkpoint is rolled back using the "before" values in the log

Recovery using Logs

- Keeps track of all transactions that update the database. It contains:
  - A record for the beginning of transaction
  - For each transaction component (SQL statement)
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - "Before" and "after" values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction
- Increases processing overhead but the ability to restore a corrupted database is worth the price

Recovery using Logs

- Increases processing overhead but the ability to restore a corrupted database is worth the price
- If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to a previous state
- The log it itself a database and to maintain its integrity many DBMSs will implement it on several different disks to reduce the risk of system failure

The Transaction Log

TABLE 9.1 A TRANSACTION LOG

TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	PRODUCT	1558-QW1	PROD_QOH	25	23
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	525.75	615.73
365	101	363	Null	COMMIT	**** End of Transaction				

TRL_ID = Transaction log record ID

PTR = Pointer to a transaction log record ID

TRX_NUM = Transaction number

(Note: The transaction number is automatically assigned by the DBMS.)

Recovery

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
     - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
     - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.

Recovery

- Some earlier part of the log may be needed for undo operations
  1. Continue scanning backwards till a record $<T_i$ **start**> is found for every transaction T_i in L.
     - Parts of log prior to earliest $<T_i$ **start**> record above are not needed for recovery, and can be erased whenever desired.

Thanks
Happy Learning