Chapter 18 : Concurrency Control

Sistemas de Bases de Dados 2019/20

Capítulo refere-se a: Database System Concepts, 7th Ed

Optimistic vs Pessimistic protocols

T1	T2
Read(A)	
	Write(A)
Read(B) Write(A)	
Write(B)	
	Read(A)

- **What to do now?**
  - It may well be that the complete transactions are serializable
  - But they may also turn out not to be serializable!

- **Optimistic protocols** do not stop at potential conflicts; if something goes wrong, rollback!
- **Pessimistic protocols** stop at potential conflicts, until no possible conflict exists; if in the end no conflict happened, it just lost time!
- Let's start with a pessimistic protocol.

Timestamp Based Concurrency Control

Timestamp-Based Protocols

- Instead of determining the order of each operation in a transaction at execution time, determines the order by the time of beginning of each transaction.
  - Each **transaction** is issued a **timestamp** when it enters the system. If an old transaction T_o has timestamp TS(T_o), a new transaction T_n is assigned time-stamp TS(T_n) such that TS(T_o) <TS(T_n).

- Timestamp-based protocols manage concurrent execution such that
  time-stamp order = serializability order
- Several alternative protocols based on timestamps

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) protocol**

- Maintains for each data item *Q* two timestamp values:
  - **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback
    - It is an optimistic protocol!

Timestamp-Based Protocols (Cont.)

- Suppose a transaction T_r issues a **read**(Q)

  1. If TS(T_r) $\leq$ **W**-timestamp(Q), then T_r needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_r is rolled back.
  2. If TS(T_r) $\geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to

     max(R-timestamp(Q), TS(T_r)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_w issues **write**(Q).
  1. If TS(T_w) < R-timestamp(Q), then the value of Q that T_w is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_w is rolled back.
  2. If TS(T_w) < W-timestamp(Q), then T_w is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_w is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_w).

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free and may not even be recoverable.

Multiversion Concurrency Control

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Key ideas:
  - Each successful **write** results in the creation of a new version of the data item written.
  - Use timestamps to label versions.
  - When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction issuing the read request, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item *Q* has a sequence of versions <Q_1, Q_2,...., Q_m>. Each version Q_k contains three data fields:
  - **Content** – the value of version Q_k.
  - **W-timestamp**(Q_k) – timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) – largest timestamp of a transaction that successfully read version Q_k

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose W-timestamp is the largest write timestamp less than or equal to TS(T_i) – i.e. the “version” of the item right before T_i started
  1. If transaction T_i issues a **read**(Q), then
     - the value returned is the content of version Q_k
     - If R-timestamp(Q_k) < TS(T_i), set R-timestamp(Q_k) := TS(T_i),
  2. If transaction T_i issues a **write**(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. Otherwise, a new version Q_i of Q is created
        - W-timestamp(Q_i) and R-timestamp(Q_i) are initialized to TS(T_i).

Multiversion Timestamp Ordering (Cont)

- Observations
  - Reads always succeed
  - A write by T_w is rejected if some other transaction T_r that (in the serialization order defined by the timestamp values) should read T_r's write, has already read a version created by a transaction older than T_r.
- Protocol guarantees serializability

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- **Update transactions**
  - When an update transaction wants to read a data item:
    - it obtains a shared lock on it and reads the latest version.
  - When it wants to write an item
    - it obtains X-lock; it then creates a new version of the item and sets this version's timestamp to ∞.
      - This is to prevent other concurrent transactions to read its value, and guarantee that other reads on the same transaction get this version.
  - When update transaction *T* completes, commit processing occurs:
    - *T* sets timestamp on the versions it has created to **ts-counter** + 1
    - *T* increments **ts-counter** by 1

Multiversion Two-Phase Locking (Cont.)

- **Read-only transactions**
  - are assigned a timestamp = **ts-counter** when they start execution
  - follow the multiversion timestamp-ordering protocol for performing reads
    - Do not obtain any locks
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g., if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, then Q5 will never be required again
- Issues with
  - primary key and foreign key constraint checking
  - Indexing of records with multiple versions

  See textbook for details

Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Use multiversion 2-phase locking
  - Give logical “snapshot” of database state to read only transaction
    - Reads performed on snapshot
  - Update (read-write) transactions use normal locking
  - Works well, but how does the system know a transaction is read only?
- Solution 2 (partial): Give snapshot of database state to every transaction
  - Reads performed on snapshot
  - Use 2-phase locking on updated data items
  - Problem: variety of anomalies such as lost update can result
  - Better solution: snapshot isolation level (next slide)

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent transactions are not visible to T1
  - Writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

T_1 deposits 50 in X	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(X_1, 150)$	
$commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

Benefits and problems of SI

- Reads are *never* blocked,
  - and don't block other transactions activities
- Performance like Read Committed
- Avoids several anomalies
  - No dirty read, i.e. no read of uncommitted data
  - tNo lost update
    - I.e., update made by a transaction is overwritten by another transaction that did not see the update)
  - No non-repeatable read
    - I.e., if read is executed again, it will see he same value
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent transactions, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- Example of problem with SI
  - Initially A = 3 and B = 17
    - In the end succeeds with A = 17 and B = 3
    - Serializing T_i before T_j results in A = B = 17
    - Serializing T_i after T_j results in A = B = 3
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
    - Two transaction can both create order with same number
      - Is an example of phantom phenomenon

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI)**: extension of snapshot isolation that ensures serializability
- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
  - Where T_i writes a data a data item Q, T_j reads an earlier version of Q, but T_j is serialized after T_i
- Idea: track read-write dependencies separately, and roll-back transactions where cycles can occur
  - Ensures serializability
  - Details in book
- Implemented in PostgreSQL from version 9.1 onwards
  - PostgreSQL implementation of SSI also uses index locking to detect phantom conflicts, thus ensuring true serializability

SI Implementations

- Snapshot isolation supported by many databases
  - Including Oracle, PostgreSQL, SQL Server, IBM DB2, etc
  - Isolation level can be set to snapshot isolation
- Oracle implements "first updater wins" rule (variant of "first committer wins")
  - Concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier
- **Warning**: *even if isolation level is set to serializable, Oracle actually uses snapshot isolation*
  - Old versions of PostgreSQL prior to 9.1 did this too
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly, after previous transaction.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC,   connection.setAutoCommit(false);
- Four levels of (weak) consistency, cf. before.

Transaction management in Oracle

- Transaction beginning and ending as in SQL
  - Explicit **commit work** and **rollback work**
  - Implicit commit on session end, and implicit rollback on failure
  - Implicit commit before and after DDL commands
- Log-based deferred recovery using rollback segment
- Checkpoints (inside transactions) can be handled explicitly
  - **savepoint** <name>
  - **rollback to** <name>
- Concurrency control is made by snapshot isolation
- Deadlock are detected using a *wait-graph*
  - Upon deadlock detection, the operation locked for longer fails (but the transaction is not rolled back)

Consistency verification in Oracle

- By default, consistency is verified after each command, rather than at the end of the transaction, as is prescribed by ACID properties
- However, it is possible to defer the verification of constraints to the end of transactions
- This requires both:
  - A prior declaration of all constraints that can possibly be deferred
    - Done by adding **deferrable** to the end of the declarations of the constraint
  - an instruction in the beginning of each of the transactions where constraints are deferred
    - Done with:
      - **set constraints all deferred** or
      - **set constraints** ***<nome$_1$>, ..., <nome$_n$>*** **deferred**

Levels of Consistency in Oracle

- Oracle implements 2 of the 4 of levels of SQL
  - *Read committed*, by default in Oracle and with
    - **set transaction isolation level read committed**
  - *Serializable* (which indeed implements *Snapshot Isolation*) with
    - **set transaction isolation level serializable**
    - Appropriate for large databases with only few updates, and usually with not many conflicts. Otherwise it is too costly.
- Further, it supports a level similar to *repeatable read*:
  - Read only mode, only allow reads on committed data, and further doesn't allow INSERT, UPDATE or DELETE on that data (without unrepeatable reads!)
    - **set transaction read only**

Granularity in Oracle

- By default Oracle performs **row level locking**.
- Command
- **select … for update**
- locks the selected rows so that other users cannot lock or update the rows until you end your transaction. Restriction:
  - Only at top-level select (not in sub-queries)
  - Not possible with **DISTINCT** operator, **CURSOR** expression, set operators, **group by** clause, or aggregate functions.
- Explicit locking of tables is possible in several modes, with
  - **lock table** <name> **in**
    - **row share mode**
    - **row exclusive mode**
    - **share mode**
    - **share row exclusive mode**
    - **exclusive mode**

Lock modes in Oracle

- Row share mode
  - The least restrictive mode (with highest degree of concurrency)
  - Allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table, except for exclusive mode
- Row exclusive mode
  - As before, but doesn't allow setting other modes except for row share.
  - Acquired automatically after a **insert**, **update** or **delete** command on a table
- Exclusive mode
  - Only allows queries to records of the locked table
  - No modifications are allowed
  - No other transaction can lock the table in any other mode
- See manual for details of other (intermediate) modes

Chapter 19: Recovery System

Sistemas de Bases de Dados 2019/20

Capítulo refere-se a: Database System Concepts, 7th Ed

Failure Classification

- **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
- **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

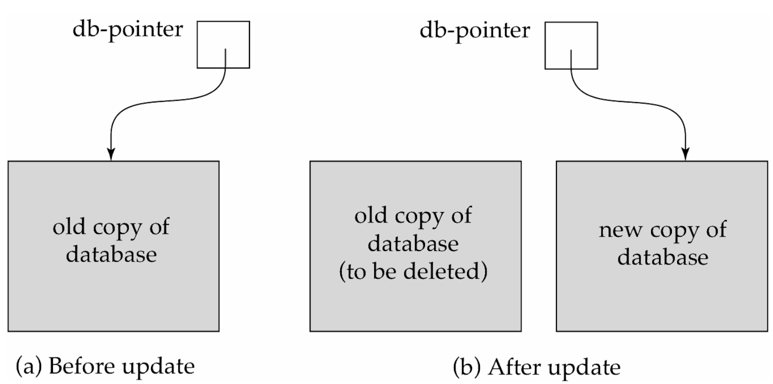
Recovery Algorithms

- Suppose transaction T_i transfers €50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms**
- Less used alternative: **shadow-copy** and **shadow-paging**

shadow-copy



Log-Based Recovery

- A **log** is a sequence of **log records**. The records keep information about update activities on the database.
  - The **log** is kept on stable storage
- When transaction T_i starts, it registers itself by writing a

  <T_i **start**> log record

- *Before* T_i executes **write**(X), a log record

  $<T_i, X, V_1, V_2>$

  is written, where V_1 is the value of X before the write (the **old value**)**,** and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record <T_i **commi**t> is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification.

Deferred Database Modifiction

- The **deferred database modification** scheme records all modifications to the log, and defers actual **write**s to after partial commit.

- Transaction starts by writing <*T* ***start***> record to log.
- A **write**(*X*) operation results in a log record <*T, X, V*> being written, where *V* is the new value for *X* (the old value is not needed).
  - The write is not performed on *X* at this time, but is deferred.
- When *T* partially commits, <*T* **commit**> is written to the log
- After that, the log records are read and used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone iff both <*T* **start**> and <*T* **commit**> are (still) in the log.
- Redoing a transaction *T* (**redo** *T*) sets the value of all data items updated by the transaction to the new values.

Immediate Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification (cont)

- Recovery procedure has two operations instead of one:
  - **undo**(*T*) restores the value of all data items updated by *T* to their old values, going backwards from the last log record for *T*
  - **redo**(*T*) sets the value of all data items updated by *T* to the new values, going forward from the first log record for *T*
- Both operations must be **idempotent**
  - I.e. even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction *T* needs to be undone if the log contains the record <*T* **start**>, but does not contain the record <*T* **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <*T* **start**> and the record <*T* **commit**>.
- Undo operations are performed before redo operations.

Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database.
- 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.
  4. All updates are stopped while doing checkpointing

Checkpoints (Cont.)

- 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.
  - 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.
- Some earlier part of the log may be needed for undo operations
  - 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.