Chapter 24: Advanced Transaction Processing

- Transaction-Processing Monitors
- Transactional Workflows
- High-Performance Transaction Systems
  - Main memory databases
  - Real-Time Transaction Systems
- Long-Duration Transactions
- Transaction management in multidatabase systems

Transaction Processing Monitors

- TP monitors initially developed as multithreaded servers to support large numbers of terminals from a single process.
- Provide infrastructure for building and administering complex transaction processing systems with a large number of clients and multiple servers.
- Provide services such as:
  - Presentation facilities to simplify creating user interfaces
  - Persistent queuing of client requests and server responses
  - Routing of client messages to servers
  - Coordination of two-phase commit when transactions access multiple servers.
- Some commercial TP monitors: CICS from IBM, Pathway from Tandem, Top End from NCR, and Encina from Transarc

TP Monitor Architectures

- Process per client model - instead of individual login session per terminal, server process communicates with the terminal, handles authentication, and executes actions.
  - Memory requirements are high
  - Multitasking - high CPU overhead for context switching between processes
- Single process model - all remote terminals connect to a single server process.
  - Used in client-server environments
  - Server process is multi-threaded; low cost for thread switching
  - No protection between applications
  - Not suited for parallel or distributed databases

TP Monitor Architectures (Cont.)

- Many-server single-router model - multiple application server processes access a common database; clients communicate with the application through a single communication process that routes requests.
  - Independent server processes for multiple applications
  - Multithread server process
  - Run on parallel or distributed database
- Many server many-router model - multiple processes communicate with clients.
  - Client communication processes interact with router processes that route their requests to the appropriate server.
  - Controller process starts up and supervises other processes.

Detailed Structure of a TP Monitor

- Queue manager handles incoming messages
- Some queue managers provide persistent or durable message queuing: contents of queue are safe even if systems fail.
- Durable queuing of outgoing messages is important
  - Application server writes message to durable que as part of a transaction
  - Once the transaction commits, the TP monitor guarantees message is eventually delivered, regardless of crashes.
  - ACID properties are thus provided even for messages sent outside the database
- Many TP monitors provide locking, logging and recovery services, to enable application servers to implement ACID properties by themselves.

Application Coordination Using TP Monitors

- A TP monitor treats each subsystem as a resource manager that provides transactional access to some set of resources.
- The interface between the TP monitor and the resource manager is defined by a set of transaction primitives.
- The resource manager interface is defined by the X/Open Distributed Transaction Processing standard.
- TP monitor systems provide a transactional remote procedure call (transactional RPC) interface to their service
  - Transactional RPC provides calls to enclose a series of RPC calls within a transaction.
  - Updates performed by an RPC are carried out within the scope of the transaction, and can be rolled back if there is any failure.
Workflow Systems

Transactional Workflows
- **Workflows** are activities that involve the coordinated execution of multiple tasks performed by different processing entities.
- With the growth of networks, and the existence of multiple autonomous database systems, workflows provide a convenient way of carrying out tasks that involve multiple systems.
- Example of a workflow delivery of an email message, which goes through several mail systems to reach destination.
  - Each mailer performs a task: forwarding of the mail to the next mailer.
  - If a mailer cannot deliver mail, failure must be handled semantically (delivery failure message).
- Workflows usually involve humans: e.g., loan processing, or purchase order processing.

Examples of Workflows

<table>
<thead>
<tr>
<th>Workflow application</th>
<th>Typical task</th>
<th>Typical processing entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>electronic-mail routing</td>
<td>electronic-mail message</td>
<td>mailers</td>
</tr>
<tr>
<td>loan processing</td>
<td>form processing</td>
<td>humans, application software</td>
</tr>
<tr>
<td>purchase-order processing</td>
<td>form processing</td>
<td>humans, application software, DBMS</td>
</tr>
</tbody>
</table>

Loan Processing Workflow

- In the past, workflows were handled by creating and forwarding paper forms
- Computerized workflows aim to automate many of the tasks. But the humans still play role e.g., in approving loans.

Transactional Workflows
- Must address following issues to computerize a workflow.
  - Specification of workflows - detailing the tasks that must be carried out and defining the execution requirements.
  - Execution of workflows - execute transactions specified in the workflow while also providing traditional database safeguards related to the correctness of computations, data integrity, and durability.
    - E.g.: Loan application should not get lost even if system fails.
  - Extend transaction concepts to the context of workflows.
  - State of a workflow - consists of the collection of states of its constituent tasks, and the states (i.e., values) of all variables in the execution plan.

Workflow Specification

- **Static specification** of task coordination:
  - Tasks and dependencies among them are defined before the execution of the workflow starts.
  - Can establish preconditions for execution of each task: tasks are executed only when their preconditions are satisfied.
  - Defined preconditions through dependencies:
    - Execution states of other tasks.
    - Output values of other tasks.
    - External variables, that are modified by external events.

Workflow Specification (Cont.)
- **Dynamic task coordination**
  - E.g., Electronic mail routing system in which the text to be schedule for a given mail message depends on the destination address and on which intermediate routers are functioning.

Failure-Automaticity Requirements
- **Usual ACID transactional requirements** are too strong/unimplementable for workflow applications.
- However, workflows must satisfy some limited transactional properties that guarantee a process is not left in an inconsistent state.
- **Acceptable termination states** - every execution of a workflow will terminate in a state that satisfies the failure-automaticity requirements defined by the designer.
  - Committed - objectives of a workflow have been achieved.
  - Aborted - valid termination state in which a workflow has failed to achieve its objectives.
- A workflow must reach an acceptable termination state even in the presence of system failures.
Execution of Workflows

Workflow management systems include:
- **Scheduler** - program that process workflows by submitting various tasks for execution, monitoring various events, and evaluation conditions related to intertask dependencies.
- **Task agents** - control the execution of a task by a processing entity.
- **Mechanism to query to state of the workflow system.**

Workflow Scheduler

- Ideally scheduler should execute a workflow only after ensuring that it will terminate in an acceptable state.
- Consider a workflow consisting of two tasks $S_1$ and $S_2$. Let the failure-atomicity requirement be that either both or neither of the subtransactions should be committed.
  - Suppose systems executing $S_1$ and $S_2$ do not provide prepared-to-commit states and $S_1$ or $S_2$ do not have compensating transactions.
  - It is then possible to reach a state where one subtransaction is committed and the other aborted. Both cannot then be brought to the same state.
  - Workflow specification is unsafe, and should be rejected.
- Determination of safety by the scheduler is not possible in general, and is usually left to the designer of the workflow.

Recovery of a Workflow

- Ensure that a failure occurs in any of the workflow-processing components, the workflow eventually reaches an acceptable termination state.
- Failure-recovery routines need to restore the state information of the scheduler at the time of failure, including the information about the execution states of each task.
- Log status information on stable storage.
- Handoff of tasks between agents should occur exactly once in spite of failure.
- Problem: Repeating handoff on recovery may lead to duplicate execution of task; not repeating handoff may lead to task not being executed.
  - Solution: Persistent messaging systems

High-Performance Transaction Systems

- High-performance hardware and parallelism help improve the rate of transaction processing, but are insufficient to obtain high performance:
  - Disk I/O is a bottleneck — I/O time (10 milliseconds) has no decreased at a rate comparable to the increase in processor speeds.
  - Parallel transactions may attempt to read or write the same data item, resulting in data conflicts that reduce effective parallelism.
- We can reduce the degree to which a database system is disk bound by increasing the size of the database buffer.

Main-Memory Database

- Commercial 64-bit systems can support main memories of tens of gigabytes.
- Memory resident data allows faster processing of transactions.
- Disk-related limitations:
  - Logging is a bottleneck when transaction rate is high.
  - Use group-commit to reduce number of output operations (Will study two slides ahead.)
  - If the update rate for modified buffer blocks is high, the disk data-transfer rate could become a bottleneck.
  - If the system crashes, all of main memory is lost.
Main-Memory Database Optimizations

- To reduce space overheads, main-memory databases can use structures with pointers crossing multiple pages. In disk databases, the I/O cost to traverse multiple pages would be excessively high.
- No need to pin buffer pages in memory before data are accessed, since buffer pages will never be replaced.
- Design query-processing techniques to minimize space overhead - avoid exceeding main memory limits during query evaluation.
- Improve implementation of operations such as locking and latching, so they do not become bottlenecks.
- Optimize recovery algorithms, since pages rarely need to be written out to make space for other pages.

Real-Time Transaction Systems

- In systems with real-time constraints, correctness of execution involves both database consistency and the satisfaction of deadlines.
  - Hard deadline - Serious problems may occur if task is not completed within deadline
  - Firm deadline - The task has zero value if it completed after the deadline.
  - Soft deadline - The task has diminishing value if it is completed after the deadline.
- The wide variance of execution times for read and write operations on disks complicates the transaction management problem for time-constrained systems.
  - Main-memory databases are thus often used
  - Waits for locks, transaction aborts, contention for resources remain as problems even if data is in main memory.
- Design of a real-time system involves ensuring that enough processing power exists to meet deadline without requiring excessive hardware resources.

Long-Duration Transactions

- Represent as a nested transaction
  - Atomic database operations (read/write) at a lowest level.
- If transaction fails, only active short-duration transactions abort.
- Active long-duration transactions resume once any short duration transactions have recovered.
- The efficient management of long-duration waits, and the possibility of aborts.
- Need alternatives to waits and aborts; alternative techniques must ensure correctness without requiring serializability.

Concurrency Control

- Correctness without serializability:
  - Correctness depends on the specific consistency constraints for the databases.
  - Correctness depends on the properties of operations performed by each transaction.
- Use database consistency constraints as to split the database into subdatabases on which concurrency can be managed separately.
- Treat some operations besides read and write as fundamental low-level operations and extend concurrency control to deal with them.

Nested and Multilevel Transactions

- A nested or multilevel transaction $T$ is represented by a set $T = \{ t_1, t_2, ..., t_n \}$ of subtransactions and a partial order $P$ on $T$.
- A subtransaction $t_i$ in $T$ may abort without forcing $T$ to abort.
- Instead, $T$ may either restart $t_i$ or simply choose not to run $t_i$.
- If $t_i$ commits, this action does not make $t_i$ permanent (unlike the situation in Chapter 15). Instead, $t_i$ commits to $T$, and may still abort (or require compensation) if $T$ aborts.
- An execution of $T$ must not violate the partial order $P$, i.e., if an edge $t_i \rightarrow t_j$ appears in the precedence graph, then $t_i \rightarrow t_j$ must not be in the transitive closure of $P$. 

Concurrency Control (Cont.)

A non-conflict-serializable schedule that preserves the sum of $A + B$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>read($A$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td></td>
<td>$A := A - 50$</td>
<td>$B := B + 10$</td>
</tr>
<tr>
<td></td>
<td>write($A$)</td>
<td>write($B$)</td>
</tr>
<tr>
<td></td>
<td>read($B$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td></td>
<td>$B := B + 50$</td>
<td>$A := A + 10$</td>
</tr>
<tr>
<td></td>
<td>write($B$)</td>
<td>write($A$)</td>
</tr>
</tbody>
</table>
Nested and Multilevel Transactions (Cont.)

- Subtransactions can themselves be nested/multilevel transactions.
- Lowest level of nesting: standard read and write operations.
- Nesting can create higher-level operations that may enhance concurrency.
- Types of nested/multilevel transactions:
  - Multilevel transaction: subtransaction of T is permitted to release locks on completion.
  - Saga: multilevel long-duration transaction.
  - Nested transaction: locks held by a subtransaction ti of T are automatically assign to T on completion of ti.

Compensating Transactions

- Alternative to undo operation; compensating transactions deal with the problem of cascading rollbacks.
- Instead of undoing all changes made by the failed transaction, action is taken to "compensate" for the failure.
- Consider a long-duration transaction T representing a travel reservation, with subtransactions T1,v which makes airline reservations, T1,r which reserves rental cars, and T1,h which reserves a hotel room.
- Hotel cancels the reservation.
- Instead of undoing all of T, the failure of T1,h is compensated for by deleting the old hotel reservation and making a new one.
- Requires use of semantics of the failed transaction.

Implementation Issues

- For long-duration transactions to survive system crashes, we must log not only changes to the database, but also changes to internal system data pertaining to these transactions.
- Logging of updates is made more complex by physically large data items (CAD design, document text); undesirable to store both old and new values.
- Two approaches to reducing the overhead of ensuring the recoverability of large data items:
  - Operation logging. Only the operation performed on the data item and the data-item name are stored in the log.
  - Logging and shadow paging. Use logging from small data items; use shadow paging for large data items. Only modified pages need to be stored in duplicate.

Example of Nesting

- Rewrite transaction T, using subtransactions T, and T, that perform increment or decrement operations:
  - T, consists of
    - T, which subtracts 50 from A
    - T, which adds 50 to B
- Rewrite transaction T, using subtransactions T, and T, that perform increment or decrement operations:
  - T, consists of
    - T, which subtracts 10 from B
    - T, which adds 10 to A
- No ordering is specified on subtransactions; any execution generates a correct result.

Transaction Management

- Local transactions are executed by each local DBMS, outside of the MDBS system control.
- Global transactions are executed under multidatabase control.
- Local autonomy - local DBMSs cannot communicate directly to synchronize global transaction execution and the multidatabase has no control over local transaction execution.
  - local concurrency control scheme needed to ensure that DBMS's schedule is serializable
  - in case of locking, DBMS must be able to guard against local deadlocks.
  - need additional mechanisms to ensure global serializability

Transaction Management in Multidatabase Systems

- Transaction management is complicated in multidatabase systems because of the assumption of autonomy
  - Global 2PL - each local site uses a strict 2PL (locks are released at the end); locks set as a result of a global transaction are released only when that transaction reaches the end.
    - Guarantees global serializability
    - Due to autonomy requirements, sites cannot cooperate and execute a common concurrency control scheme.
    - E.g. no way to ensure that all databases follow strict 2PL
- Solutions:
  - provide very low level of concurrent execution, or
  - use weaker levels of consistency

Two-Level Serializability

- DBMS ensures local serializability among its local transactions, including those that are part of a global transaction.
- The multidatabase ensures serializability among global transactions alone - ignoring the orderings induced by local transactions.
- 2LSR does not ensure global serializability, however, it can fulfill requirements for strong correctness.
  1. Preserve consistency as specified by a given set of constraints
  2. Guarantee that the set of data items read by each transaction is consistent
- Global-read protocol: Global transactions can read, but not update, local data items; local transactions do not have access to global data. There are no consistency constraints between local and global data items.

Two-Level Serializability (Cont.)

- Local-read protocol: Local transactions have read access to global data; disallows all access to local data by global transactions.
  - A transaction has a value dependency if the value that it writes to a data item at one site depends on a value that it read for a data item on another site.
  - For strong correctness: No transaction may have a value dependency.
- Global-read/write/local-read protocol: Local transactions have read access to global data; global transactions may read and write all data;
  - No consistency constraints between local and global data items.
  - No transaction may have value dependency.
Global Serializability

- Even if no information is available concerning the structure of the various concurrency control schemes, a very restrictive protocol that ensures serializability is available.
- Transaction-graph: a graph with vertices being global transaction names and site names.
- An undirected edge \((T_i, S_j)\) exists if \(T_i\) is active at site \(S_j\).
- Global serializability is assured if transaction-graph contains no undirected cycles.

Ensuring Global Serializability

- Each site \(S_i\) has a special data item, called ticket.
- Every transaction \(T_j\) that runs at site \(S_i\) writes to the ticket at site \(S_i\).
- Ensures global transactions are serialized at each site, regardless of local concurrency control method, so long as the method guarantees local serializability.
- Global transaction manager decides serial ordering of global transactions by controlling order in which tickets are accessed.
- However, above protocol results in low concurrency between global transactions.

Weak Levels Consistency

- Use alternative notions of consistency that do not ensure serializability, to improve performance.
- Degree-two consistency avoids cascading aborts without necessarily ensuring serializability.
  - Unlike two-phase locking, S-locks may be released at any time, and X-locks may be acquired at any time.
  - X-locks be released until the transaction either commits or aborts.

Example Schedule with Degree-Two Consistency

Nonserializable schedule with degree-two consistency (Figure 20.5) where \(T_3\) reads the value if \(Q\) before and after that value is written by \(T_4\).

<table>
<thead>
<tr>
<th>(T_3)</th>
<th>(T_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-S (Q)</td>
<td>lock-X (Q)</td>
</tr>
<tr>
<td>read (Q)</td>
<td>read (Q)</td>
</tr>
<tr>
<td>unlock (Q)</td>
<td>write (Q)</td>
</tr>
</tbody>
</table>

Cursor Stability

- Form of degree-two consistency designed for programs written in general-purpose, record-oriented languages (e.g., Pascal, C, Cobol, PL/I, Fortran).
- Rather than locking the entire relation, cursor stability ensures that:
  - The tuple that is currently being processed by the iteration is locked in shared mode.
  - Any modified tuples are locked in exclusive mode until the transaction commits.
- Used on heavily accessed relations as a means of increasing concurrency and improving system performance.
- Use is limited to specialized situations with simple consistency constraints.