Advanced Transaction Management

Literature:

Gray, J., Reuter, A.,
Transaction Processing - Concepts and Techniques
Morgan Kaufmann, 1993, Chapter 4 (Transaction Models)

Garcia-Molina, H., Ullman, J.D., Widom, J.,
Database Systems – The Complete Book
Prentice Hall, 2002, Chapters 18 – 19

Elmasri, R., Navathe, S.B.,
Addison-Wesley, 2000, Chapters 19 - 20

Overview

- Some basics
- Spheres of Control
- Flat transactions with savepoints
- Chained transactions
- Nested transactions
- Multilevel transactions
- Open nested transactions
- Distributed transactions
- Long-lived transactions
  - Sagas
  - Cooperative transactions
  - Workflow management
- Transactions in OO-DBMS
- Summary & comparison
Transaction Models

Aims of DBMS transaction concept:
- consistency checking
- synchronization of multiuser mode
- data integrity (recovery in case of system errors)

Type of Transaction Model (TM) is determined by:
- transaction duration
- size of processed entities
- way of cooperation / concurrency

The classical ACID requirements

- Atomicity
  State transitions should apparently jump from the initial state to the result without any observable intermediate states – or apparently never leave the initial state.

- Consistency
  A transaction produces consistent results only; otherwise it aborts. A result is consistent if the new state of the DB fulfills all the consistency constraints.

- Isolation
  A program running under transaction protection must behave exactly as it would in single-user mode.

- Durability
  Results of transactions having completed successfully must not be forgotten by the system.
Serializability-Based Correctness

Mostly used: Conflict serializability

- A history is a sequence of operations from possibly interleaved transactions
- Two histories are conflict equivalent if the relative order of conflicting operations is the same
- A history is serializable if it is conflict equivalent with a serial history (no interleaving)
- But what is a “conflict”?
  - Commutativity (example: Write always conflicts with other reads and writes)
  - Invalidation (example: P invalidates Q if H1,P,H2 and H1, H2, Q are legal, but H1,P,H2,Q is not)
  - Others …

Non-serializable Correctness Criteria

- The Serializability-based correctness criteria may be too strong to obtain the wanted concurrency
- Weaker correctness-criteria may be established with knowledge about the application semantics
  - Examples:
    - Group transactions into classes that can interleave without problems/ cannot interleave at all
    - User defined correctness criteria
    - Cooperating transactions (see later)
Classification of Action Types

Three types of actions as building blocks for system services

- **Unprotected actions**
  Only C, not AID
  Must either be controlled by the application environment, or must be embedded in a higher-level protected action.
  Example: single disk write.

- **Protected actions**
  ACID properties
  Building blocks for reliable, distributed applications.
  Example: classical DBMS transactions such as SELECT.

- **Real actions**
  Affect the real, physical world in a way that is hard or impossible to reverse: definitely D.
  Example: cutting metal pieces, chemical reactions.

Example

```
Begin_Transaction (work)
SELECT ... (protected action) -> read_block* (unprotected actions)
UPDATE ... (protected action) -> read_block + write_block (unprotected actions)
DRILL_HOLE ... (real action) -> defer execution until end of transaction
INSERT ... (protected action) -> write_block (unprotected action)
SELECT ... (protected action) -> read_block (unprotected action)
...
End_of_Transaction (IF condition COMMIT (work) -> execute real action
ELSE ROLLBACK (work) -> do not execute real action)
```
New Requirements for Transactions

Some examples:

- **CAD (machine design)**
  - Large (complex) entities are processed in a complex way
  - Long duration (hours/days/weeks)
  - Much cooperation between users -> design transactions

- **Trip planning**
  - Large and recursive structures -> selective rollback desirable -> savepoints, nested transactions

- **Bulk updates**
  - Minimize lost work in case of crashes -> chained transactions, sagas

> We need various extensions of the flat transaction model for the problems resulting from the different application domains.

Sphere of Control (SOC)

– jf. Gray & Reuter section 4.3

- **Sphere of Control**
  - Containing the effects of arbitrary operations as long as there might be a necessity to revoke them
  - Monitoring the dependencies of operations on each other in order to be able to trace the execution history in case faulty data are found.

- System must be structured into a hierarchy of abstract data types.
- SOCs are general – can contain different things
- SOCs are dynamic – can be created, modified and removed at run-time
- SOCs gave rise to transactions, which may be seen as a special case
- The DBMS has the Database as its SOC – the DBSOC
- When the DBMS allows a transaction to run, the transaction creates its own SOC (by locking)
Transaction Processing Context

A general view of a program:
\[ f(\text{input\_message}, \text{context}) \rightarrow (\text{output\_message}, \text{context}) \]

- Programs with no context information (context is the empty set) are called context-free.
- Programs with context information are called context-sensitive.

Examples of context:
- counters (how often was the program called)
- user-related data (last screen sent)
- durable storage information (last record read/modified)

Notation for Transaction Models

- State-transition diagram for a flat transaction as seen by the application (see Figure).
- What is required to describe TMs?
  - Structural dependencies (hierarchy of abstract data types)
  - Dynamic dependencies (use of shared data)
Diagramming technique (Gray & Reuter)

signal entries for the atomic action to perform a state transition

unique persistent transaction identifier

state indicators of the action’s outcome

<table>
<thead>
<tr>
<th>A(bort)</th>
<th>B(egin)</th>
<th>C(ommit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(borted)</th>
<th>C(ommitted)</th>
</tr>
</thead>
</table>

- Active part: 3 events can cause an atomic action to change its state - BEGIN, ROLLBACK, and COMMIT.
- TMs can define conditions that trigger these events.
- Passive part: Atomic actions that depend on other transactions might not be able to make certain state transitions all by themselves. There must be rules specifying the conditions for performing a (signalled) state transition.

Example: Describing Flat Transactions

deactivated

- (a) T is active; abort of system TA will cause T to roll back, too
- (b) Termination 1: T has committed
- (c) Termination 2: T has aborted
Flat transaction with Savepoints

- Relaxing the “all-or-nothing” characteristics of flat transactions.
- Savepoint: causes the system to record the current state of processing and returns a handle to the application program.
- Execution between two savepoints = atomic action (SoC)
- Application program can do a partial ROLLBACK to a specific Savepoint.
- A crash (murder) rolls back the whole transaction

See also Gray & Reuter Figure 4.8 (page 188)

Savepoint execution

Not shown graphically:
- Rollback to a previous savepoint
- Abort of whole transaction

See also Gray & Reuter Figure 4.8 (page 188)
Chained Transactions

- Commit of $T_i$ + begin of $T_{i+1} = $ one atomic action
- No other $T$ can see or alter the context data passed from $T_i$ to $T_{i+1}$
- By commit of $T_i$, the program waives its rights to do a rollback
- By commit of $T_i$, the program can release all the objects that are no longer needed, and pass on the processing context to $T_{i+1}$
- At the same time, the whole chain can be seen as one $T$, that keeps all its locks and cursors

Variation of savepoints!

Chained Transactions – normal execution

- First TA in the chain has been started; start of the second TA will be triggered by commit of the first TA.

- First TA in the chain has committed; second TA got started as part of commit of $C_1$. Now the second TA is structurally dependent of “system”. 
Chained Transactions – restart processing

Restart processing in a chained transaction

- **Trigger**
- **System**
  - A
  - B
  - C

- **A**
- **B**
- **C**

- **Restart**
  - A
  - B
  - C

- **C1**
  - A
  - B
  - C

- **C1’**
  - A
  - B
  - C

- **C2**
  - A
  - B
  - C

Chained Transactions vs. Savepoints

- **Workflow structure:** substructure similar to savepoints
- **Commit versus savepoint:** only rollback to last savepoint possible (savepoints in chained transactions - other approach)
- **Lock handling:** COMMIT allows to free locks, savepoints have no implications on locks
- **Work lost:** savepoints allow for a more flexible state restoration
- **Restart handling:** reestablishes the state of the most recent COMMIT
Nested transactions

- A transaction tree in which a transaction \( T \) may contain subtransactions
- The top-level \( T \) appears to the outside as a normal atomic transaction
- Subtransactions can fail and be restarted or replaced by another subtransaction without causing the whole \( T \) to fail or restart
- In case the recovery of a subtransaction fails, the whole \( T \) will fail

Nested Transactions

- Multiple variants (open and closed)
- Can increase intra-transaction parallelism
- Closed variant does not increase inter-transaction parallelism
- Can be used as basis for more complex transaction models
- Especially suitable for distributed environments/applications
- Internally, subtransactions are run concurrently and their actions are synchronized by an internal concurrency control mechanism.
Closed Nested Transactions

- Subtransactions maintain isolation (I) against each other
- Locks are kept until the end of the subtransaction (which enables the isolation of parallel subtransactions)
- By end of subtransaction, locks are inherited to the supertransaction and are released only when the top-level transaction terminates
- Does not increase the intertransaction parallelism
  This is a crucial point especially in long and interactive transactions

Open Nested Transactions

- Changes made by subtransactions are visible before the top-level transaction terminates
- Avoid conflicts, parallel subtransactions are separated
- Locks are released at the end of the subtransaction
- Two important problems can occur in this approach:
  - Terminated subtransactions may be rolled back because a supertransaction aborts (problem of cascading rollback)
    => One solution for this problem is to introduce compensating transactions (More later!)
  - Loss of serializability
    => One solution of this problem is semantic concurrency control
Nested Transactions - Graphical Representation

Using Nested Transactions

- Powerful mechanism for fine-tuning the scope of rollback in applications with complex structure
  Example: trip planning
- Modularization in software engineering and nested transactions complement each other:
  - modular design takes care of the application structure and encapsulates local (dynamic) data structures
  - the transaction makes sure that the global data used by these modules are isolated and that recovery with the same granularity is provided

*Nested transactions are a generalization of savepoints: Whereas savepoints allow organizing a transaction into a sequence of actions that can be rolled back individually, nested transactions form a hierarchy of pieces of work (similar to nesting SoCs)*
Multi-Level Transactions

- Multi-Level Transactions: Transactions that are abstracted into subtransactions implemented by a set of lower-level operations
- The abstraction can be extended to multiple levels
- Multi-Level Ts are a generalized version of nested transactions
- Multi-level Ts let the subtransactions commit immediately (pre-commit), thereby waiving the right to uniliteral backout of updates
- Instead, a compensating subtransaction for the subT that is about to commit is installed, so it can be executed in case the pre-committed results turn out to be invalid
- The compensating subTs can be thrown away unused when the root transaction commits
- Since compensating subTs exist at all levels, it is guaranteed that all updates can be revoked, even if the root transaction fails and a whole number of subTs have committed before
- For long-duration Ts, concurrency can be increased

Multi-Level vs. Nested Transactions

<table>
<thead>
<tr>
<th></th>
<th>Multi-Level</th>
<th>Nested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layers</td>
<td>A predefined number of levels, two adjacent pairs define a layer</td>
<td>No predefined notion of layers</td>
</tr>
<tr>
<td>Abstractions</td>
<td>The higher the level, the more abstract the operations</td>
<td>Not necessarily a notion of abstraction</td>
</tr>
<tr>
<td>Concurrency control</td>
<td>Opens for a modular approach to concurrency control</td>
<td>A single global mechanism must be used</td>
</tr>
</tbody>
</table>

The multi-level Ts notion of layers is very important: It prevents serialization problems between precommitted subtransactions, because they are protected by the transactions on the higher levels
Multi-Level Transactions - Graphical Representation

Top-level T is running

SubT N has been started

SubT N has committed and has installed its compensating T CN, that will get started if T aborts; CN must commit

Distributed Transactions

- A distributed transaction is typically a flat transaction that runs in a distributed environment and therefore has to visit several nodes in the network, depending on where the data is.
- The conceptual difference between a distributed transaction and a nested transaction is:
  - The structure of nested transactions is determined by the functional decomposition of the application, that is, by what the application views as SoC.
  - The structure of a distributed transaction depends on the distribution of data in a network. Even for a flat transaction, from the application’s point of view a distributed transaction may have to be executed if the data involved are scattered across a number of nodes.
**Distributed Transactions - Graphical Representation**

- **System**
  - Node A: T is running at node A
  - Node A: remote subTs T1 and T2 have been started

**Short vs. long transactions**

- Traditional transactions are of short duration (seconds and minutes), traditional schemes rely on blocking transactions terminating quickly, thus enabling other transactions to continue execution.
- Traditional DBMS enforce serializable schedules of transactions, with the major differences being in the size of the information grain whose access is individually controlled.
- When transactions execute for long durations, this scheme does not work; a long-running transaction may block out other transactions for days or months.
- Transactions in DBMS for nonstandard application domains tend to be very long, consequently the usual serializability criterion is not adequate. The long duration of these tasks (transactions) means that the concurrency control strategies used in conventional DBMS are not appropriate.
Long-Lived Transactions

- Examples:
  - Computer aided design
  - CASE
  - Bulk updates
- Long transaction mechanisms should be able to save intermediate states, this can be accomplished by Savepoints or Nested transactions
  - however, these techniques do not increase concurrency
- Some solutions
  - Minibatch
  - Sagas
  - Cooperative transactions

Mini-Batch

A primitive solution for bulk updates (example: Adding interest to 100 000 accounts)

- The huge sequence of actions is split into shorter sequences which are run as separate transactions (mini-batches) under the control of the application program
- The application program keeps track of progress, saving the identity of the last successfully executed minibatch in a safe place – i.e. the database

Simple solution, minimize lost work, but:

- The application program is really doing transaction-monitor work
- Transactions of other application programs may intermingle
Sagas

- Split a long T (Saga) into arbitrary many (shorter) subT $T_1 \ldots T_n$ with corresponding compensating transactions $CT_1 \ldots CT_n$
- A commit of a subT make its changes visible to the rest of the world
- An abort rolls back the subT, then recovery with help of compensating CTs: If $T_k$ fails, the transactions $CT_{k-1} \ldots CT_1$ are executed
- Transactions may be rolled back to a chosen commit

Sagas support long-running transactions/activities, **but:** Sagas are not suitable for cooperating and unpredictable (dynamic) T-activities – this is the case when the user is controlling the T (interactive mode)

Sagas

- Two extensions of chained transaction concept:
  - It defines a chain of transactions as a unit of control – a saga
  - It uses the compensation idea from multi-level transactions to make the entire chain atomic
- The backward chain of compensating transactions is established as the original chain proceeds in a forward direction
- The compensating transaction can undo the transaction’s effect should the saga abort, even when another unrelated transaction has already executed
- Sagas relax the restriction that the subtransactions of the saga execute without external interference
- Additional concurrency can be gained by type-specific concurrency control

Sagas are an alternative to mini-batches
Cooperative Transactions

A solution for design work
– in a design team, designers often look at each others’ incomplete or inconsistent results to guide their own work (examples: Computer Aided Design, CASE applications)
- The concept of cooperative transactions allow transactions to view each others’ partial results under certain conditions
- Possible solution: Check-in/check-out model

Transaction A
do some design work
show me object X
use object X
give back X

create object X
Transaction B

granted
Check-in/check-out

One possible scheme to support cooperative work

- Users wishing to cooperate "check out" objects from the global DB into a private workspace
- In the private workspace, users operate on the objects outside the DB concurrency control
- When work is finished, the objects are "checked in" to the global DB, which enforces normal concurrency control. The entire collection of operations in the private workspace appear as a single transaction
- The check-in and check-out operations are normal (short) transactions against the global DB
- The check-in/check-out scheme supports flexible concurrency control while not requiring changes to the DB concurrency control; but it delegates much of the responsibility for data integrity and concurrency control to the users

Xymphonic Systems (APOTRAM)

- Developed by Telenor Research and Development
  Dr. Ole Jørgen Anfindsen
- Based on two ideas:
- Relaxing the rules of conflict serializability
  Not ACID, but ACCID (CC – Conditional Concurrency)
- Modified nested transaction model: Subdatabases
- see http://www.xymphonic.com/
Conditional Conflict serializability (CCSR)

- Classical Conflict serializability: two operations are conflicting if at least one of them is a write operation.
- Conditional Conflict serializability (CCSR): Applications can use reads/writes with “quality”-parameters.
- Let A and B be sets of “quality”-parameter values.
  Let W(A) and R(B) denote write and read operations.
  W(A) and R(B) will conflict with each other unless A is a subset of B.
- Example:
  - The set of available parameter values is {good, medium, bad}.
  - W(bad) and R(good) would conflict.
  - W(good) and R(medium, good) would not conflict.
- The application programmers need to understand the semantics of the parameter values, the transaction manager does not.

Formalism

- The only change is in the definition of conflict.
- The Generalized Serializability theorem: A transaction history is CCSR if and only if its conditional conflict graph is acyclic.
- CCSR can be enforced by 2PL.
- Transaction histories that are rigorous, strict etc modulo CCSR can be defined (the formal foundation for non-compensation based recovery).
  - i.e.: CCSR also offers automatic recovery.
  - (No compensation transactions necessary)
Nested conflict serializability (NCSR)

- The write Sphere of Control (WSOC) can be seen as a single user database
- The idea:
  Replace a (single-user) WSOC with a (multi-user) DBSOC (subdatabase)
- Other users can then execute transactions within this dynamically created
  DBSOC on behalf of the transaction that created it
- Can be extended to arbitrary number of levels -> nested databases
- If we enforce conflict serializability as the correctness criterion at all levels -> nested conflict serializability (NCSR)
- NCSR generalizes CSR by allowing the sequence of operations within a
  WSOC to be serializable rather than serial; thus applying CSR recursively
- This idea allows arbitrarily complex patterns of collaboration involving two
  or more users

ODMG: Transactions i OO-databases

- Programs using persistent objects are organized into transactions
- Assume a linear sequence of transactions within a thread
- A transaction run against a single logical ODMS
- Transient objects are not subject to transaction semantics
- Transactions are run under the control of Transaction objects
**ODMG: The Transaction Object**

```java
db = DatabaseFactory.new( );
db.open("Databasename");
t = TransactionFactory.new( );
t.join( ); // connect t to thread
t.begin( );
... // objects explicitly or implicitly locked
myobject.lock(read) // write, upgrade
t.commit( ); or t.abort( );
t.leave( );
db.close( );
```

<table>
<thead>
<tr>
<th>db:Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>void open(in string odms_name)</td>
</tr>
<tr>
<td>void close( )</td>
</tr>
<tr>
<td>void bind(in Object an_object, in string name)</td>
</tr>
<tr>
<td>Object unbind(in string name)</td>
</tr>
<tr>
<td>Object lookup(in string object_name)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t:Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>void begin( )</td>
</tr>
<tr>
<td>void commit( )</td>
</tr>
<tr>
<td>void abort( )</td>
</tr>
<tr>
<td>void checkpoint( )</td>
</tr>
<tr>
<td>void join( )</td>
</tr>
<tr>
<td>void leave( )</td>
</tr>
<tr>
<td>boolean isOpen( )</td>
</tr>
</tbody>
</table>

*Exceptions are left out*

---

**Transactions i OO-databases – beyond ODMG**

- Nested transactions
- Long-lived transactions
  - Make the transaction object persistent!
- Multiprocess transactions
- Multi-database transactions
- Transaction consistency over transient objects
- Operations for ODMS administration in the Database type
  - (create, delete, move, copy, reorganize, verify, backup, restore)
Workflow Management (1)

- Workflows:
  Coordinating execution of multiple tasks performed by different processing entities
- A task defines some work to be done and can be specified in a number of ways, e.g., textual description in a file or email, form, message, or computer program
- A processing entity that performs the tasks may be a person or a software system, e.g., mailer, application program, or DBMS
- Specification of a workflow involves describing those aspects of its tasks and processing entities that are relevant to controlling and coordinating the execution. It also requires specification of the relationships among tasks and their execution requirements. These can be specified using a variety of software paradigms, e.g., rules, constraints, or programs
- Execution of multiple tasks by different processing entities may be controlled by a human coordinator or by a software system called a workflow management system

Workflow Management (2)

Workflow models:

- Based on ideas borrowed from operating systems (job control language), long-running activities, and other transaction models

Specification of workflows:

- Task specification: execution structure of each task is defined by providing a set of externally observable execution states and a set of transitions between these states. Characteristics of processing entities relevant for task-execution requirements may be defined
- Task coordination requirements: are expressed as intertask-execution and data-flow dependencies, and termination conditions of the workflow. -> statically or dynamically
- Execution (correctness) requirements: are defined to restrict the execution of the workflow(s) to meet application-specific correctness criteria, including failure-atomicity and execution-atomicity requirements, and workflow concurrency control and recovery requirements
Workflow Management (3)

Three implementation approaches:

- Embedded approach: requires and exploits significant support from the processing entities for specification and enforcement of dependencies.

- Layering approach: implements workflow-control facilities on top of uniform application-level interfaces to entities by developing modules to support intertask dependencies and other workflow specifications.

- Approach providing limited workflow-management facilities in an environment consisting of transaction monitors and event monitoring/synchronizing facilities (similar to approach proposed in CORBA).

Summary (1)

- Flat (short) transactions are transactions like they are known from traditional DBMS, they do not have any subtransactions. Traditional DBMS must support the notion of atomic, recoverable, and serializable transactions (ACID properties). The "classical" transaction management is not flexible enough for nonstandard application domains. Depending on the application domain some of the ACID properties are too restrictive for Nonstandard-DBMS. Therefore, the four properties atomicity (A), consistency (C), isolation (I), and durability (D) are only partially desirable.

- Atomicity: Owing to long transaction durations (e.g., transactions in CAD applications can last days or even weeks) it is not acceptable to undo a complete transaction if a hardware failure or deadlock occurs. Therefore, it must be possible to define additional recovery units, to get specific save points.
Summary (2)

- **Consistency**: Consistency constraints should not be checked at the end of a transaction. It is necessary to introduce smaller consistency units. Non-standard-DBMS must check very complex integrity constraints which requires new mechanisms to define and control such integrity constraints.

- **Isolation**: In many nonstandard application domains it is necessary to share data (e.g., in a group of engineers) by exchanging uncommitted data (dirty read). Corresponding protocols should control such exchanges.

- **Durability**: It should be possible to define savepoints within a long transaction or to divide transactions into subtransactions to provide a partial rollback/undo to avoid the complete rollback/undo if any failure occurs during the long transaction. If the supertransaction aborts it may be necessary to rollback/undo an already committed subtransaction or to execute a compensating transaction for the specific subtransactions.

Summary (3)

- **Synchronization mechanisms** determine crucially the efficiency of DBMS. There has to be a balance between the necessary effort of a locking technique and its performance.

- **Optimistic synchronization mechanisms** are not suitable for long transactions, because access conflicts are first recognized at the end of a transaction which might cause a rollback (that might destroy the work of weeks).

- **Nonstandard-DBMS** should support cooperative work more directly by providing and enforcing a wider variety of lock types (e.g., object locks) than the customary read/write locks.

- **The Nonstandard-DBMS** should provide the same level of recovery as conventional DBMS. Nonstandard-DBMS are often used in client-server environments, consequently recovery and logging mechanisms have often to solve the same problems as they occur in distributed DBMS. Some solutions provide extended logging approaches for snapshot refreshing or the distributed two-phase locking protocol.