Models of Transactions

Chapter 19

Structuring Applications

• Many applications involve long transactions that make many database accesses
• To deal with such complex applications many transaction processing systems provide mechanisms for imposing some structure on transactions

Flat Transaction

• Consists of:
  – Computation on local variables
    • not seen by DBMS; hence will be ignored in most future discussion
  – Access to DBMS using call or statement level interface
    • This is transaction schedule; commit applies to these operations
• No internal structure
• Accesses a single DBMS
• Adequate for simple applications

Some Limitations of Flat Transactions

• Only total rollback (abort) is possible
  – Partial rollback not possible
• All work lost in case of crash
• Limited to accessing a single DBMS
• Entire transaction takes place at a single point in time

Flat Transaction

• Abort causes the execution of a program that restores the variables updated by the transaction to the state they had when the transaction first accessed them.

Providing Structure Within a Single Transaction
**Savepoints**

- **Problem**: Transaction detects condition that requires rollback of recent database changes that it has made
- **Solution 1**: Transaction reverses changes itself
- **Solution 2**: Transaction uses the rollback facility within DBMS to undo the changes

**Example of Savepoints**

- Suppose we are making airplane reservations for a long trip
  - London-NY  NY-Chicago  Chicago-Des Moines
- We might put savepoints after the code that made the London-NY and NY-Chicago reservations
- If we cannot get a reservation from Chicago to Des Moines, we would rollback to the savepoint after the London-NY reservation and then perhaps try to get a reservation through St Louis

**Savepoints**

```
begin transaction
S1:
Call to DBMS
sp1 := create_savepoint();
S2:
sp2 := create_savepoint();
S3:
if (condition) {rollback (sp1); S5};
S4:
commit
```

- Rollback to sp1 causes database updates subsequent to creation of sp1 to be undone
  - S2 and S3 updated the database (else there is no point rolling back over them)
- Program counter and local variables are not rolled back
- Savepoint creation does not make prior database changes durable (abort rolls all changes back)

**Distributed Systems: Integration of Legacy Applications**

- **Problem**: Many enterprises support multiple legacy systems doing separate tasks
  - Increasing automation requires that these systems be integrated

```
withdraw part
return part
stock level
order part
payment
```

```
Inventory
Application
DBMS 1
Site B
Billing
Application
DBMS 2
Site C
```

**Distributed Transactions**

- Incorporate transactions at multiple servers into a single (distributed) transaction
  - Not all distributed applications are legacy systems; some are built from scratch as distributed systems

```
tx_begin;
order_part;
withdraw_part;
payment;
tx_commit;
```

```
Inventory
Application
DBMS 1
Site B
Billing
Application
DBMS 2
Site C
```

**Distributed Transactions**

- **Goal**: distributed transaction should be ACID
  - Each subtransaction is locally ACID (e.g., local constraints maintained, locally serializable)
  - In addition the transaction should be globally ACID
    - A: Either all subtransactions commit or all abort
    - C: Global integrity constraints are maintained
    - I: Concurrently executing distributed transactions are globally serializable
    - D: Each subtransaction is durable
Banking Example

• **Global atomicity** - funds transfer
  – Either both subtransactions commit or neither does
  
  ```
  tx_begin;
  withdraw(acct1);
  deposit(acct2);
  tx_commit;
  ```

Banking Example (con’t)

• **Global consistency** -
  – Sum of all account balances at bank branches =
    total assets recorded at main office

Banking Example (con’t)

• **Global isolation** - local serializability at each site
  does not guarantee global serializability
  – post_interest subtransaction is serialized after audit
    subtransaction in DBMS at branch 1 and before audit
    in DBMS at branch 2 (local isolation), but
  – there is no global order

  ```
  post_interest
  audit
  ```

  ↓

  post_interest time

  ↓

  post interest at branch 1;
  post interest at branch 2;
  sum balances at branch 1;
  sum balances at branch 2;

Exported Interfaces

Local system might export an interface for executing
individual SQL statements.

**Exported Interfaces**

- DBMS 1
  - site A
    - subtransaction
      - EXEC SQL SELECT...
      - EXEC SQL INSERT...
      - EXEC SQL SELECT...
    - tx_commit;

- DBMS 2
  - site C
    - subtransaction
      - EXEC SQL SELECT...
      - tx_commit;

Alternatively, the local system might export an interface
for executing subtransactions.

Multidatabase

• Set of databases accessed by a distributed
transaction is referred to as a **multidatabase** (or
federated database)
  – Each database retains its autonomy and might support
    local (non-distributed) transactions
• Multidatabase might have global integrity
  constraints
  – *e.g.*, Sum of balances of individual bank accounts at all
    branch offices = total assets stored at main office

Transaction Hierarchy

• A distributed transaction invokes subtransactions.
• General model: one distributed transaction might
  invoke another as a subtransaction, yielding a
  hierarchical structure
Models of Distributed Transactions

- Can siblings execute concurrently?
- Can parent execute concurrently with children?
- Who initiates commit?

**Hierarchical Model:** No concurrency among subtransactions, root initiates commit

**Peer Model:** Concurrency among siblings and between parent and children, any subtransaction can initiate commit

Distributed Transactions

- Transaction designer has little control over the structure. Decomposition fixed by distribution of data and/or exported interfaces (legacy environment)
- Essentially a *bottom-up* design

Nested Transactions

- **Problem:** Lack of mechanisms that allow:
  - *a top-down*, functional decomposition of a transaction into subtransactions
  - individual subtransactions to abort without aborting the entire transaction
- Although a nested transaction looks similar to a distributed transaction, it is *not* conceived of as a tool for accessing a multidatabase

Characteristics of Nested Transactions

- (1) Parent can create children to perform subtasks; children might execute sequentially or concurrently; parent waits until all children complete (no communication between parent and children).
- (2) Each subtransaction (together with its descendants) is isolated with respect to each sibling (and its descendants). Hence, siblings are serializable, but order is not determined and nested transaction is *non-deterministic*.
- (3) Concurrent nested transactions are serializable.

- (4) A subtransaction is atomic. It can abort or commit independently of other subtransactions. Commit is *conditional* on commit of parent (since child task is a subtask of parent task). Abort causes abort of all subtransaction’s children.
- (5) Nested transaction commits when root commits. At that point updates of committed subtransactions are made durable.

Characteristics of Nested Transactions - Example

- Booking a flight from London to Des Moines
  - L -- DM
  - C
  - L -- NY
  - C
  - NY -- DM
  - C
  - NY -- StL
  - A
  - C
  - NY -- StL
  - C
  - StL -- DM

- C = commit
- A = abort
- Concurrent
- Sequential
- Stop in Chicago
- Stop in St. Louis
**Nested Transactions**

- A nested transaction is a transaction that is enclosed within another transaction.

**Characteristics of Nested Transactions**

- (6) Individual subtransactions are not necessarily consistent, but nested transaction as a whole is consistent.

---

**Structuring to Increase Transaction Performance**

- **Problem:** In the models previously discussed, a transaction generally locks items it accesses and holds locks until commit time to guarantee serializability.

  ```plaintext
  acquire lock on x  release lock on x
  T1: r(x:12) .. compute .. w(x:13) commit
  T2: request read(x) r(x:13) .. compute .. w(x:14) ..
      (wait) acquire lock on x
  - This eliminates bad interleavings, but limits concurrency and hence performance
  ```

---

**Example - Switch Sections**

- **Transaction Moves Student from Section s1 to Section s2, uses TestInc, Dec**

---

**Chained Transactions**

- **Problem 1 (trivial):** Invoking `begin_transaction` at the start of each transaction involves communication overhead.

  - With chaining, a new transaction is started automatically for an application program when the program commits or aborts the previous one.
    - This is the approach taken in SQL.
Chained Transactions

- **Problem 2**: If the system crashes during the execution of a long-running transaction, considerable work can be lost.
  - Chaining allows a transaction to be decomposed into sub-transactions with intermediate commit points.
  - Database updates are made durable at intermediate points => less work is lost in a crash.

Chaining Considerations - Atomicity

- **Transaction as a whole is not atomic**. If crash occurs:
  - **DBMS** cannot roll the entire transaction back.
  - Initial subtransactions have committed,
    - Their updates are durable
    - The updates might have been accessed by other transactions (locks have been released)
  - Hence, the *application* must roll itself forward.

Example

- Chaining compared with savepoints:
  - **Savepoint**: explicit rollback to arbitrary savepoint; all updates lost in a crash
  - **Chaining**: abort rolls back to last commit; only the updates of the most recent transaction lost in a crash.

Chaining Considerations - Atomcity

- Roll forward requires that on recovery the application can determine how much work has been committed
  - Each subtransaction must tell successor where it left off.
- Communication between successive subtransactions cannot use local variables (they are lost in a crash)
  - Use the database to communicate between subtransactions
    - **subtrans 1**
      - r(x:15)… w(x:24)… commit
    - **subtrans 2**
      - r(x:30)… w(x:30)… commit

 Chaining Considerations

- **Transaction as a whole is not isolated**.
  - Database state between successive subtransactions might change since locks are released (but performance improves)
  - **Subtransactions might not be consistent**
    - Inconsistent intermediate states visible to concurrent transactions during execution or after a crash.
Alternative Semantics for Chaining

- Chain commits the transaction (makes it durable) and starts a new transaction, but does not release locks
  - Individual transactions do not have to be consistent
  - Recovery is complicated (as before); rollforward required
  - No performance gain

A Problem With Obtaining Atomicity With Chaining

- Suppose we use the first semantics for chaining
  - Subtransactions give up locks when they commit
- Suppose that after a subtransaction of a transaction $T$ makes its changes to some item and commits
  - Another transaction changes the same item and commits
  - $T$ would then like to abort
  - Based on our usual definition of chained transactions, atomicity cannot be achieved because of the committed subtransactions

Partial Atomicity

- Suppose we want to achieve some measure of atomicity by undoing the effects of all the committed subtransactions when the overall transaction wants to abort
- We might think we can undo the updates made by $T$ by just restoring the values each item had when $T$ started (physical logging)
  - This will not work

An Example

$T_1$: Update($x$) commit$_{1,1}$ ... abort$_1$
$T_2$: Update($x$) commit

If, when $T_1$ aborts, we just restore the value of $x$ to the value it had before $T_1$ updated it, $T_2$’s update would be lost

Compensation

- One approach to this problem is compensation
- Instead of restoring a value physically, we restore it logically by executing a compensating transaction
  - In the student registration system, a Derегистration subtransaction compensates for a successful Registration subtransaction
  - Thus Registration increments the Enrollment attribute and Derегистration decrements that same attribute
  - Compensation works even if some other concurrent Registration subtransaction has also incremented Enrollment

Sagas: An Extension To Chained Transactions That Achieves Partial Atomicity

- For each subtransaction, $ST_i$, in a chained transaction, $T$, a compensating transaction, $CT_i$, is designed
- Thus if a transaction $T$ consisting of 5 chained subtransactions aborts after the first 3 subtransactions have committed, then $ST_1, ST_2, ST_3, CT_3, CT_1, CT_2$ will perform the desired compensation
Sagas and Atomicity

- With this type of compensation, when a transaction aborts, the value of every item it changed is eventually restored to the value it had before that transaction started.
- However, complete atomicity is not guaranteed.
  - Some other concurrent transaction might have read the changed value before it was restored to its original value.

Declarative Transaction Demarcation

- We have already talked about two ways in which procedures can execute within a transaction:
  - As a part of the transaction
    - Stored procedure
  - As a child in a nested transaction

Declarative Transaction Demarcation (con’t)

- Two other possible ways:
  - The calling transaction is suspended, and a new transaction is started. When it completes the first transaction continues.
    - Example: The called procedure is at a site that charges for its services and wants to be paid even if the calling transaction aborts.
  - The calling transaction is suspended, and the called procedure executes outside of any transaction. When it completes the first transaction continues.
    - Example: The called procedure accesses a non-transactional file system.

Declarative Transaction Demarcation (con’t)

- One way to implement such alternatives is through declarative transaction demarcation:
  - Declare in some data structure, outside of any transaction, the desired transactional behavior.
  - When the procedure is called, the system intercepts the call and provides the desired behavior.

Implementation of Declarative Transaction Demarcation

- Declarative transaction demarcation is implemented within J2EE and .NET.
  - We discuss J2EE (.NET is similar).
- The desired transactional behavior of each procedure is declared as an attributed in a separate file called the deployment descriptor.

Transaction Attributes

- Possible attributes (in J2EE) are:
  - Required
  - RequiresNew
  - Mandatory
  - NotSupported
  - Supports
  - Never
- The behavior for each attribute depends on whether or not the procedure is called from within a procedure:
  - All possibilities are on the next slide.
### Description (con’t)

- **Required:**
  - The procedure must execute within a transaction
    - If called from outside a transaction, a new transaction is started
    - If called from within a transaction, it executes within that transaction

- **RequiredNew:**
  - Must execute within a new transaction
    - If called from outside a transaction, a new transaction is started
    - If called from within a transaction, that transaction is suspended and a new transaction is started. When that transaction completes, the first transaction resumes
      - Note that this semantics is different from nested transactions. In this case the commit of the new transaction is not conditional.

- **Mandatory:**
  - Must execute within an existing transaction
    - If called from outside a transaction, an exception is thrown
    - If called from within a transaction, it executes within that transaction

- **NotSupported:**
  - Does not support transaction
    - If called from outside a transaction, a transaction is not started
    - If called from inside a transaction, that transaction is suspended until the procedure completes after which the transaction resumes

- **Supported:**
  - Can execute within or not within a transaction, but cannot start a new transaction
    - If called from outside a transaction, a transaction is not started
    - If called from inside a transaction, it executes within that transaction
Description (con’t)

• **Never:**
  – Can never execute within a transaction
    • If called from outside a transaction, a new transaction is not started
    • If called from within a transaction, an exception is thrown

Example

• The **Deposit** and **Withdraw** transactions in a banking application would have attribute **Required**.
  – If called to perform a deposit, a new transaction would be started
  – If called from within a **Transfer** transaction to transfer money between accounts, they would execute within that transaction

Advantages

• Designer of individual procedures does not have to know the transactional context in which the procedure will be used
• The same procedure can be used in different transaction contexts
  – Different attributes are specified for each different context
• We discuss J2EE in more detail and how declarative transaction demarcation is implemented in J2EE in the Architecture chapter.

Multilevel Transactions

• A multilevel transaction is a nested set of subtransactions.
  – The commitment of a subtransaction is **unconditional**, causing it to release its locks, *but*
  – Multilevel transactions are atomic and their concurrent execution is serializable

Multilevel Transactions

• Data is viewed as a sequence of increasing, application oriented, levels of abstraction
• Each level supports a set of abstract objects and abstract operations (methods) for accessing those objects
• Each abstract operation is implemented as a transaction using the abstractions at the next lower level

Example - Switch Sections
Multilevel Transactions

- Parent initiates a single subtransaction at a time and waits for its completion. Hence a multilevel transaction is sequential.
- All leaf subtransactions in the tree are at the same level
- Only leaf transactions access the database.
- Compare with distributed and nested models

---

Multilevel Transactions

- When a subtransaction (at any level) completes, it commits unconditionally and releases locks that it has acquired on items at the next lower level.
  - `TestInc(s2)` locks `t2`; unlocks `t2` when it commits
- The change it has made to the locked item becomes visible to subtransactions of other transactions
  - The incremented value of `t2` is visible to a subsequent execution of `TestInc` or `Dec` by concurrent transactions
- This creates problems maintaining isolation and atomicity.

---

Maintaining Isolation

- `p2` is unlocked when `Sel` commits
- `TestInc1: Sel(t2) Upd(t2)`
- `TestInc2: Sel(t2) Upd(t2)`
  - `Sel2 can lock p2`
- **Problem:** Interleaved execution of two `TestInc`’s results in error (we will return to this later)

---

Maintaining Atomicity

- `Move1: TestInc(s2) Dec(s1)` **abort**
- `Move2: TestInc(s3) Dec(s1) commit`
- When `T1` aborts, the value of `s1` that existed prior to its access cannot simply be restored (physical restoration)
- Logical restoration must be done using **compensating transactions**
  - `Inc` compensates for `Dec`; `Dec` compensates for a successful `TestInc`; no compensation needed for unsuccessful `TestInc`

---

Compensating Transactions

- Multilevel model uses compensating transaction

  - `T1: TestInc(s2) Dec(s1)`
  - `Inc(s1) Dec(s2)`
  - `T2: TestInc(s3) Dec(s1)`
  - `Abort`

---

Correctness of Multilevel Transactions

- As we shall see later,
  - Multilevel transactions are atomic
    - In contrast with Sagas, which also use compensation, but do not guarantee atomicity
  - Concurrent execution of multilevel transactions is serializable
Recoverable Queues

- **Problem**: Distributed model assumes that the subtransactions of a transaction follow one another immediately (or are concurrent).
- In some applications the requirement is that a subtransaction be *eventually* executed, but not necessarily immediately.
- A recoverable queue is a transactional data structure in which information about transactions to be executed later can be durably stored.

---

Transactional Features

T1: begin transaction;  
compute:  
item := service  
enqueue(item);  
commit;  

- Item is enqueued if T1 commits (deleted if it aborts); item is deleted if T2 commits (restored if it aborts)
- An item enqueued by T1 cannot be dequeued by T2 until T1 commits
- Queue is durable

---

Pipeline Queue for Billing Application

- Order entry transaction
- Shipping transaction
- Billing transaction

---

Concurrent Implementation of the Same Application

- Order entry transaction
- Shipping transaction
- Billing transaction

---

Recoverable Queue

- Queue could be implemented within database, but performance suffers
  - A transaction should not hold long duration locks on a heavily used data structure

---

Recoverable Queue

- Separate implementation takes advantage of semantics to improve performance
  - *enqueue* and *dequeue* are atomic and isolated, but some queue locks are released immediately
**Recoverable Queue**

- Queue and DBMS are two separate systems
  - Transaction must be committed at both but
    - isolation is implemented at the DBMS and applies to the schedule of requests made to the DBMS only

**Scheduling**

- As a result, any scheduling policy for accessing the queue might be enforced
  - but a FIFO queue might not behave in a FIFO manner

```
T1: enq(I1) …commit  restore I1
T2: enq(I2) …commit    
T3: deq(I1) … abort
T4: deq(I2) …commit
```

**Performing Real-World Actions**

- **Problem:** A real-world action performed from within a transaction, T, cannot be rolled back if crash occurs before commit.
  
  ```
  T: begin_transaction;
  compute;
  update_database;
  activate_device;
  commit;
  ```

- On recovery after a crash, how can we tell if the action has occurred?
  - ATM example: We do not want to dispense cash twice.

**Performing Real-World Actions**

- **Solution:** (part 1) T enqueues entry. If T aborts, item is dequeued; if T commits action executed later

```
T: begin_transaction;  
queue
T: begin_transaction;  
dequeue_entry;  
activate_device;  
commit;
```

**Performing Real-World Actions**

- Server executes T_D in a loop
  - but problem still exists within T_D

**Performing Real-World Actions**

- **Solution:** (part 2)
  - Device maintains read-only counter (hardware) that is automatically incremented with each action
    - Action and increment are assumed to occur atomically
  - Server performs:  
    ```
    T_D: begin_transaction;
    dequeue;
    activate_device;
    record_counter_in_db;
    commit;
    ```

**Performing Real-World Actions**

- On recovery:
  
  ```
  Restore queue and database (value read from counter) to last commit;
  if (device value > recorded value)  
  then discard head entry;  
  Restart server;
  ```
Example of Real World Action

• Suppose the hardware counter and the database counter were both at 100 before the transaction started
  – When the hardware performs its action, it increments its counter to 101
  – TD would then increment the database counter to 101
• If the system crashed after the hardware performed its action the database increment (if it had occurred) would be rolled back to 100
• Thus when the system recovered
  – If the hardware counter was 101 and the database counter was 100, we would know that the action had been performed.
  – If both counters were the same (100), we would know that the action had not taken place.

Workflow

• **Problem:** None of the previous models are sufficiently flexible to describe complex, long-running enterprise processes involving computational and non-computational tasks in distributed, heterogeneous systems over extended periods of time

Workflow Task

• Self-contained job performed by an agent
  – Inventory transaction (agent = database server)
  – Packing task (agent = human)
• Has an associated role that defines type of job
  – An agent can perform specified roles
• Accepts input from other tasks, produces output
• Has physical status: committed, aborted, ...
  – Committed task has logical status: success, failure

Workflow

• Task execution precedence specified separately from task itself
  – using control flow language:
    ```
    initiate T2, T3 when T1 committed
    ```
  – or using graphical tool:

• Conditional alternatives can be specified:
  ```
  if (condition) execute T1 else execute T2
  ```

• Conditions:
  – Logical/physical status of a task
  – Time of day
  – Value of a variable output by a task
• Alternative paths can be specified in case of task failure

Forwarding Agent

• Implementing deferred service.
Workflow
• Specifies flow of data between tasks

Execution Precedence in a Catalog Ordering System

Flow of Data in a Catalog Ordering System

Workflow Agent
• Capable of performing tasks
• Has a set of associated roles describing tasks it can do
• Has a worklist listing tasks that have been assigned to it
• Possible implementation:
  – Worklist stored in a recoverable queue
  – Agent is an infinitely looping process that processes one queue element on each iteration

Workflow and ACID Properties
• Individual tasks might be ACID, but workflow as a whole is not
  – Some task might not be essential: its failure is ignored even though workflow completes
  – Concurrent workflows might see each other’s intermediate state
  – Might not choose to compensate for a task even though workflow fails

Workflow and ACID Properties
• Each task is either
  – Retriable: Can ultimately be made to commit if retried a sufficient number of times (e.g., deposit)
  – Compensatable: Compensating task exists (e.g., withdraw)
  – Pivot: Neither retrievable nor compensatable (e.g., buy a non-refundable ticket)
Workflow and ACID Properties

- The atomicity of a workflow is guaranteed if each execution path is characterized by \{compensatable\}*, \{pivot\}, \{retriable\}*
- This does not guarantee isolation since intermediate states are visible

Workflow Management System

- Provides mechanism for specifying workflow (control flow language, GUI)
- Provides mechanism for controlling execution of concurrent workflows:
  - Roles and agents
  - Worklists and load balancing
  - Filters (data reformatting) and controls flow of data
  - Task activation
  - Maintain workflow state durably (data, task status)
  - Use of recoverable queues
  - Failure recovery of WFMS itself (resume workflows)

Importance of Workflows

- Allows management of an enterprise to guarantee that certain activities are carried out in accordance with established business rules, even though those activities involve a collection of agents, perhaps in different locations and perhaps with minimal training