Chapter 21
Transactions and Concurrency

© Pearson Education Limited 2005

- Concurrency in system transactions
  - ACID, isolation, locks
  - Transactional models and concurrency schemes
- Concurrency in business transactions
  - Business transactions
  - Transaction services across tiers
  - Offline concurrency patterns
    - Unit of work
    - Optimistic offline lock
    - Pessimistic offline lock
Main concepts

- **Transaction = business transaction**
  - a logical unit of work that accomplishes a particular business task and guarantees the integrity of the database after the task completes

- **System transaction**
  - defined by the support mechanisms provided to the transaction notion by a DBMS or other transaction management system
  - a technical concept that does not have understanding of the business reasons for the transaction as a unit of work
  - typically, a business transaction consists of a number of system transactions

- **Short transactions**
  - in (highly concurrent) business applications

- **Long transactions**
  - in (highly cooperative) workgroup computing systems

- **Relationship to database recovery issues**

Concurrency in system transactions

- **Demarcation or bracketing** of a transaction
  - the act of establishing transaction boundary
  - demarcation of a system transaction → server-side demarcation
  - demarcation of a business transaction → client-side demarcation

- **Transaction start** – implicit or `begin tran`

- **Transaction end** – `commit tran` or `rollback tran`, but can be implicit
ACID properties

- **Atomicity**
  - the smallest possible unit
  - all-or-nothing requirement
  - voting schemes (two-phase commit and three-phase commit)

- **Durability**
  - guarantees that committed changes made to the database are permanent (persistent)
    - atomicity = undo
    - durability = redo
  - (persistent) transaction log

- **Consistency**
  - transaction will fail if it attempts to violate the specified integrity rules

- **Isolation**
  - transaction produces the results on the database that do not depend in any way on other transactions
    - Serializable execution
    - locks

Isolation levels

- The context for tradeoffs between execution correctness, concurrency and performance is provided by four isolation levels defined in the SQL, Java and other standards

- **Serializable**
  - protected from reading uncommitted changes from other transactions
  - read and query consistency

- **Repeatable read**
  - guarantees read consistency, but does not guarantee query consistency \(\rightarrow\) permits phantoms

- **Read committed**
  - can lead to nonrepeatable reads
  - places shared locks on data being read by the transaction and honors exclusive locks held on data by other transactions

- **Read uncommitted**
  - dirty reads
  - no shared locks are issued and no exclusive locks are honored
Isolation levels behavior

- A programmer can set different isolation levels for different transactions executing concurrently
- The default isolation level in most systems is read committed

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>dirty read</th>
<th>nonrepeatable read</th>
<th>phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>serializable</td>
<td>not possible</td>
<td>not possible</td>
<td>not possible</td>
</tr>
<tr>
<td>repeatable read</td>
<td>not possible</td>
<td>not possible</td>
<td>possible</td>
</tr>
<tr>
<td>read committed</td>
<td>not possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>read uncommitted</td>
<td>possible</td>
<td>possible</td>
<td>possible</td>
</tr>
</tbody>
</table>

Lock modes

- exclusive (write) locks
  - deadlock possible
- update (write intent) locks
  - prevent many deadlocks
  - guarantees that the transaction will be able to upgrade the lock to the exclusive mode as soon as it requests such upgrade
    - only one transaction can hold an update lock on a resource at a time
- shared (read) locks
  - does not restrict other transactions from reading the same resource or from obtaining an update lock on that resource
    - For the repeatable read or serializable isolation, shared locks are held until the end of transaction
    - For the read committed isolation level, shared locks are only held for the duration of the server reading the data
Lock levels

- **row**
  - lowest possible lock in relational databases

- **page**
  - locks all rows that reside on a single page (typically between 8KB and 128KB)

- **table**
  - engages all rows of a database table

Database servers have a capability of lock escalation

---

Flat transaction

1. findFlight()
2. beginTransaction()
3. reserveSeat(Flight)
4. payAccount(CreditCard, Reservation)
5. commit() or rollback()
Chained transaction

1. findFlight()
2. beginTran()
3. reserveSeat(Flight)
4. savePoint("Parent Tran").
5. payAccount(CreditCard, Reservation)
6. commit() or rollback()

Nested transaction

1. beginTran()
2. reservePlaneTicket()
3. beginSubTran("Car")
4. reserveCarHire()
5. commit("Car") or rollback("Car")
6. beginSubTran("Hotel")
7. reserveHotelRoom()
8. commit("Hotel") or rollback("Hotel")
9. commit() or rollback()
Concurrent control schemes

- **Pessimistic concurrency control**
  - as discussed so far
  - based on an assumption that conflicts between transactions are likely

- **Optimistic concurrency control**

- **Multiversion concurrency control**

- **Collaborative concurrency control**

**Optimistic concurrency control**

- assumes that conflicts between transactions are infrequent
- does not use locks
- three stages:
  - the transaction reads data from the database without any restrictions,
  - when the transaction decides to commit changes in the database, the system checks if the changes would not overwrite changes made by another transaction in the meantime (i.e. since the transaction started),
  - if no conflict is detected, the transaction will be allowed to commit; otherwise, the transaction will be rolled back and the user may need to restart it
Multiversion concurrency control
- sometimes treated as a variant of the optimistic control
- a transaction never waits to read a data object because it is given the object with a new version number \( \Rightarrow \) check-out operation
- check-in always succeeds because the object has its own unique version number
  - however, the system checks for versions in the database that are in conflict and instigates a conflict resolution process
- a preferred modus operandi in object databases
  - long transactions

Collaborative concurrency control
- combined use of different schemes
- concurrency during a long transaction is managed by a persistent (long) lock, which can span short transactions and database sessions
  - long transaction runs in a private workspace of the user – objectives are to:
    - minimize rollbacks and deadlocks,
    - exchange information between cooperating users,
    - allow concurrent updates on the same objects (in private databases),
    - detect data inconsistencies and mediate their resolutions
  - long transaction activates short locks for the duration check-in/check-out
  - long transaction activates also persistent locks placed on objects in the group workspace
- having a long transaction executing in a private workspace and short transactions running in both group and private workspaces, allows the implementation of nested transactions
  - updates to a workgroup database can be committed or aborted without committing or aborting the surrounding transaction in a private workspace
In a **system transaction**, the demarcation is between the application and the database
- A system transaction fulfils a single user request, such as “modify customer address”, “make a payment”, etc.

In a **business transaction**, the demarcation is between the user and the application
- A business transaction
  - groups multiple user requests into a demarcated sequence of work
  - spans a number of system transactions \( \rightarrow \) the steps needed to move from one system transaction to another do not get server-side support similar to concurrency control mechanisms discussed before \( \rightarrow \) the responsibility for the ACID properties of a business transaction is moved up the tiers from a database server towards the application client and the user \( \rightarrow \) **offline concurrency**
Concurrency in business transactions

- In a **system transaction**, the demarcation is between the application and the database.
  - In typical scenarios, system transactions are short and do not span sessions or requests.
- In a **business transaction**, the demarcation is between the user and the application.
  - A business transaction groups multiple user requests into a demarcated sequence of work → it spans a number of system transactions.
  - The responsibility for the ACID properties of a business transaction is moved up the tiers from a database server towards the application client and the user → **offline concurrency**.

Business transaction execution contexts

- A **session** is defined by user/computer interactions between the time the user logs in to the system and logs out from the system.
- A **request** corresponds to a single call from the outside world which the software works on and for which it optionally sends back a response.” (Fowler, 2003, p.65)
  - Transactions spanning multiple requests involve **offline concurrency**.
  - There may be many sessions for a single request (for example, a request from a web-based client may engage a HTTP session to the web server, an application session to the application server, and a database session to the database server).
- A transaction executes within a **process** or within a **thread**
  - A **process** represents a time slice during which the program executes.
    - A program may be executing in more than one process.
  - Execution of such program involves **interprocess communication**
  - A **thread** is a lightweight portion of a process that takes advantage of resources allocated to the process.
The J2EE platform defines components for transaction services within the Java Transaction API (JTA) and the Java Transaction Service (JTS).

- JTA specifies standard Java interfaces between various application tiers and resources and a transaction manager.
- JTS specifies the implementation of a transaction manager that supports JTA.

A JTA transaction can be a distributed transaction spanning multiple databases.
- supports only flat transactions; it does not support chained and nested transactions

The J2EE platform offers two kinds of transaction demarcation:
- **Declarative demarcation** uses the deployment configuration information to select the way of automatically starting and completing transactions.
  - Enterprise beans in an application server support the declarative demarcation.
- **Programmatic demarcation** requires direct coding of demarcation using JTA.

In most cases, business transactions are initiated in applets and application clients but the real transactional work is done in system’s lower tiers.
- The task of demarcating the transactional request received from the user is normally delegated to a web server or application server (possibly via web server).

**Web tier**

**Business transaction demarcated in a servlet**

```
1: Context ic = new InitialContext();
2: UserTransaction ut =
    (UserTransaction) ic.lookup("java:comp/env/UserTransaction");
3: ut.begin();
4: try{
5:    insertMovieToDB(movie);
6:    Iterator it = actors.iterator();
7:    while(it.hasNext())
8:       insertActorToDB((Actor)it.next());
9:    it = listedas.iterator();
10:   while(it.hasNext())
11:      insertListedAsToDB((ListedAs)it.next());
12:   ut.commit();
13: }catch( Exception exc){
14:   ut.rollback();
15: }
```
Business transactions should be managed by the application server tier, whenever possible.

- Transactional enterprise beans are available for this task.

Enterprise beans offer two kinds of transaction demarcation:

- programmatic (bean-managed) and
- declarative (container-managed)

---

Programmatic demarcation

```
1: MovieHome = (MovieHome) PortableRemoteObject.narrow(
    ctx.lookup("MovieHome"), MovieHome.class);
2: 3: UserTransaction ut = ejbContext.getUserTransaction();
4: ut.begin();
5: try{
6:    home.create("Neil Jordan", "Interview with the Vampire",
        "The vampire bla bla bla");
7:    ut.commit();
8: }catch(Exception exc){
9:    ut.rollback();
10: }
```
Declarative demarcation

The J2EE platform encourages the use of container-managed transactional services.

Methods of a business component (enterprise bean) can be assigned one of six transaction attributes, which determine transactional behavior of the component:

- **Required**
  - A bean’s method performs always within a JTA transaction context

- **RequiresNew**
  - A bean’s method starts a new JTA transaction

- **NotSupported**
  - A bean’s method suspends any incoming transactional context and the method executes without transactional demarcation

- **Supports**
  - If the method is called with transaction context, it behaves as for the Required case. If it is called without transaction context, it behaves as for the NotSupported case.

- **Mandatory**
  - If the method is called with transaction context, it behaves as for the Required case. However, when it is called without transaction context, the container throws an exception.

- **Never**
  - A bean’s method with Never attribute forbids transactional context on the method.

Database tier

System transactions are demarcated within the **database tier**.

In J2EE specifications, such transactions are called **local transactions** of the resource manager (e.g. a database manager).

A recommended practice of the J2EE platform (from the perspective of business transactions) is to demarcate transactions above the database tier:

- Databases are accessed within the demarcation of JTA transactions, originating in enterprise beans of the application tier or in the servlets/JSP pages of the web tier.
- These longer transactions can span more than one database connection.

The problem with client-demarcated transactions is that each server transaction ends with explicit commit or rollback → changes committed to the database cannot be simply uncommitted (“rolled back”) by a JTA transaction → **compensating transaction**.
The **Unit of Work** pattern "maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems." (Fowler, 2003).

The Unit of Work pattern prescribes what actions should be taken on transient in-memory objects prior to opening and demarcating a business transaction aimed at committing changes to the database.

```java
public class UnitOfWork {
    private final List newObjects = new ArrayList();
    private final List dirtyObjects = new ArrayList();
    private final List removedObjects = new ArrayList();

    public void registerNew(IEEntity entity) {
        newObjects.add(entity);
    }

    public void registerDirty(IEEntity entity) {
        dirtyObjects.add(entity);
    }

    public void registerClean(IEEntity entity) {
        newObjects.remove(entity);
    }

    public void registerRemoved(IEEntity entity) {
        removedObjects.add(entity);
    }

    public void commit() {
        for (IEEntity entity : newObjects) {
            registerDirty(entity);
            registerClean(entity);
        }
        for (IEEntity entity : dirtyObjects) {
            registerRemoved(entity);
        }
        for (IEEntity entity : removedObjects) {
            registerClean(entity);
        }
    }

    public IEEntity getCurrent() {
        return null;
    }

    public void setCurrent(IEEntity entity) {
    }

    public void insertNew(IEEntity entity) {
    }

    public void updateDirty(IEEntity entity) {
    }

    public void deleteRemoved(IEEntity entity) {
    }
}
```
Optimistic Offline Lock pattern

- Unit of Work is only a single request, single session, single process, single thread, and single business transaction solution.

- The Optimistic Offline Lock pattern “prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction” (Fowler, 2003).
  - The assumption of the Optimistic Offline Lock is that conflicts are rare.
  - The pattern checks at commit time if the data objects to be written to persistent store have not been changed by another business transaction since they have been read from the store. If this has been the case, the business transaction is rolled back.

- In practice, this pattern amounts to application-implemented methods of non-pessimistic concurrency control schemes.

- To resolve conflicts “offline”, the pattern recommends using:
  - version numbers, and
  - information about when and who last modified the record.

Pessimistic Offline Lock pattern

- The Pessimistic Offline Lock pattern “prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data” (Fowler, 2003).

- Pessimistic Offline Lock recommends three types of locks on session data used by a business transaction:
  - exclusive write lock
  - exclusive read lock
  - read/write lock
    - allows multiple concurrent read locks under the condition that no write lock is held

- A need for a shareable persistent table in a database that maps locks to business transactions ➔ semaphore table ➔ it stores what is currently locked, which business transaction locks it, and what kind of lock is applied ➔ business transactions are obliged to query the semaphore table before they can proceed.
**Summary**

- A **business transaction** consists of one or more **system transactions**.
- System transactions must satisfy the **ACID properties** (Atomicity, Consistency, Isolation, and Durability).
- SQL standard defines four **isolation levels** for transactions: serializable, repeatable read, read committed, and read uncommitted.
- There are various **transactional models**:
  - flat transaction
  - chained transaction
  - nested transaction.
- There are various **concurrency schemes**:
  - pessimistic
  - optimistic
  - multiversion
  - collaborative
- In a **system transaction**, the demarcation is between the application and the database. In a **business transaction**, the demarcation is between the user and the application.
- The developer's task of putting together a number of system transactions into a business transaction and satisfy the ACID properties in the process, is called **offline concurrency**.
- The J2EE platform offers two kinds of **transaction demarcation** – declarative and programmatic.
- Patterns for programming offline concurrency include Unit of Work, Optimistic Offline Lock and Pessimistic Offline Lock.