

**COSC 304**  
***Introduction to Database Systems***

***Advanced SQL***

**Dr. Ramon Lawrence**  
**University of British Columbia Okanagan**  
**[ramon.lawrence@ubc.ca](mailto:ramon.lawrence@ubc.ca)**

# Transaction Management Overview

---

The database system must ensure that the data stored in the database is always **consistent**. There are two challenges in preserving database consistency:

- ◆ 1) The database system must handle **failures** of various kinds such as hardware failures and system crashes.
- ◆ 2) The database system must support **concurrent execution** of multiple transactions and guarantee that this concurrency does not lead to inconsistency.

A **transaction** is an **atomic** program that executes on the database and preserves the consistency of the database.

- ◆ The input to a transaction is a consistent database AND the output of the transaction must also be a consistent database.
- ◆ A transaction must execute completely or not at all.

# *Transaction Management*

## *Motivating Example*

---

Consider a person who wants to transfer \$50 from a savings account with balance \$1000 to a checking account with current balance = \$250.

- ◆ 1) At the ATM, the person starts the process by telling the bank to remove \$50 from the savings account.
- ◆ 2) The \$50 is removed from the savings account by the bank.
- ◆ 3) Before the customer can tell the ATM to deposit the \$50 in the checking account, the ATM “crashes.”

Where has the \$50 gone?

It is lost if the ATM did not support transactions!  
The customer wanted the withdraw and deposit to both happen in one step, or neither action to happen.



# ACID Properties

---

To preserve integrity, transactions have the following properties:

- ◆ **Atomicity** - Either all operations of the transaction are properly reflected in the database or none are.
- ◆ **Consistency** - Execution of a transaction in isolation preserves the consistency of the database.
- ◆ **Isolation** - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
- ◆ **Durability** - After a transaction successfully completes, the changes it has made to the database persist, even if there are system failures.

# ACID Properties

---

**Question:** Two transactions running at the same time can see each other's updates. What ACID property is violated?

- A) atomicity
- B) consistency
- C) isolation
- D) durability
- E) none of them

# *Transaction Definition in SQL*

---

In SQL, a transaction begins implicitly.

A transaction in SQL ends by:

- ◆ **Commit** accepts updates of current transaction.
- ◆ **Rollback** aborts current transaction and discards its updates.  
Failures may also cause a transaction to be aborted.

An **isolation level** reflects how a transaction perceives the results of other transactions. It applies only to your perspective of the database, not other transactions/users. Lowering isolation level improves performance but may potentially sacrifice consistency.

# Example Transactions

---

Transaction to deposit \$50 into a bank account:

-- **TRANSACTION T1:**

```
UPDATE Account WHERE num = 'S1' SET balance=balance+50;  
COMMIT;
```

Transaction to calculate totals for all accounts (twice):

-- **TRANSACTION T2:**

```
SELECT SUM(balance) as total1 FROM Account;  
SELECT SUM(balance) as total2 FROM Account;  
COMMIT;
```

Transaction to add a new account:

-- **TRANSACTION T3:**

```
INSERT INTO ACCOUNT (num, balance) VALUES ('S5', 100);  
COMMIT;
```



# Levels of Consistency in SQL

---

The isolation level can be specified by:

SET TRANSACTION ISOLATION LEVEL =  $X$  where  $X$  is

- ◆ **Serializable** - transactions behave like executed one at a time.
- ◆ **Repeatable read** - repeated reads must return same data. Does not necessarily read newly inserted records.
- ◆ **Read committed** - only committed values can be read, but successive reads may return different values.
- ◆ **Read uncommitted** - even uncommitted records may be read. Reading an uncommitted value is called a *dirty read*.



# *Scheduling of Transactions*

---

Each transaction in a database is a separate executing program.

- ◆ A transaction may be its own program or a thread of execution.

The operating system schedules the execution of programs outside of the control of the DBMS.

- ◆ Thus, transactions may be executed in any order (as long as the order of operations within a transaction are the same). This interleaving is what produces different schedules.

The DBMS uses its concurrency control protocol to restrict the schedules to those that respect the consistency specified by the user for the transaction isolation level.

- ◆ All transactions must write lock any data item updated and the relation lock if inserting.
- ◆ Isolation level only affects read locks.

# Transaction Schedule Example

---

Transaction **T2** that does two queries has its statements interleaved with transaction **T1** that does an update:

```
SELECT SUM(balance) as total1 FROM Account;  --T2
UPDATE Account WHERE num='S1' SET balance=balance+50; --T1
COMMIT;  -- T1
SELECT SUM(balance) as total2 FROM Account;  -- T2
COMMIT;  -- T2
```

With isolation level **read committed**, total1 will not be the same as total2.

With isolation level **repeatable read**, this schedule is not possible as **T2** will lock all accounts stopping **T1** from updating.

## Transaction Schedule Example (2)

---

Transaction **T2** that does two queries has its statements interleaved with transaction **T3** that does an insert:

```
SELECT SUM(balance) as total1 FROM Account;  --T2
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT;  -- T3
SELECT SUM(balance) as total2 FROM Account;  -- T2
COMMIT;  -- T2
```

With isolation level **repeatable read**, total1 will not be the same as total2.

With isolation level **serializable**, this schedule is not possible as **T2** will lock the account table, stopping **T3** from inserting.

# Summary of Isolation Levels

Isolation Level	Problems	Lock Usage	Speed	Comments
Serializable	None	Read locks held to commit ; read lock on relation	Slowest	Only level that guarantees correctness.
Repeatable read	Phantom tuples	Read locks held to commit	Medium	Useful for modify transactions. May not see tuples inserted by others.
Read committed	Phantom tuples, values may change	Read locks released after each statement	Fast	Useful for transactions where operations are separable but updates are all or none.  Re-reading same value may produce different results.
Read uncommitted	Phantoms, values may change, dirty reads	No read locks	Fastest	Useful for read-only transactions that tolerate inaccurate results. May see updates that will never be committed.

# Read Committed Question

---

**Question:** Will this transaction always see the same results for both queries if executed using **READ COMMITTED**?

```
SELECT SUM(balance) as total1 FROM Account;  
SELECT SUM(balance) as total2 FROM Account;  
COMMIT;
```

**A)** yes

**B)** no

# Repeatable Read Question

---

**Question:** Will this transaction always see the same results for both queries if executed using **REPEATABLE READ**?

```
SELECT SUM(balance) as total1 FROM Account;  
SELECT SUM(balance) as total2 FROM Account;  
COMMIT;
```

**A)** yes

**B)** no

# JDBC Transaction Example

```

try (Connection con = DriverManager.getConnection(url, uid, pw);
     Statement stmt = con.createStatement();)
{
    con.setAutoCommit(false); ← Force explicit commit/rollback
    ResultSet rst = stmt.executeQuery("SELECT ename,salary
                                      FROM Emp WHERE eno='E1'");

    if (rst.next() && rst.getDouble(2) < 50000)
    {
        stmt.executeUpdate("UPDATE Emp SET salary=100000
                           WHERE eno='E1'");
        con.commit(); ← Commit work to DB
    }
    con.rollback(); ← Rollback work done
}
catch (SQLException ex)
{
    System.err.println(ex); con.rollback();
}

```

# Advanced SQL Overview

---

The SQL standard has evolved to define a language that is computationally complete and supports features required by a modern object-relational DBMS.

An **object-relational DBMS** is a relational DBMS extended to support object-oriented features such as inheritance, user defined types, and polymorphism.

- ◆ Supports active code such as triggers and stored procedures.
- ◆ Handle impedance mismatch between program code and SQL.
  - ⇒ SQL is declarative and set based -- Java is procedural and object based.
- ◆ Extend database to support new types and operations.



# *User-Defined Types in SQL*

---

General form (simple version):

```
CREATE TYPE typeName AS builtInType [FINAL];
```

⇒ `Final` means that cannot create subtypes of the new type.

Example: Create SSN type:

```
CREATE TYPE SSN AS VARCHAR(10) FINAL;
```

User-defined types may either extend base types as shown here or define a structure similar to a class with attributes, routines, and operators.

# User-Defined Types Example

```
CREATE TYPE PersonType AS (
    dateOfBirth DATE,
    fName        VARCHAR(15),
    lName        VARCHAR(15),
    sex          CHAR)
```

**INSTANTIABLE**

← Can create objects

**NOT FINAL**

← Can create subtypes

**REF IS SYSTEM GENERATED**

← DB creates references

**INSTANCE METHOD age() RETURNS INTEGER;**

← Define method

```
CREATE INSTANCE METHOD age() RETURNS INTEGER
FOR PersonType
BEGIN
    RETURN /* Code to find age from Self.dateOfBirth */
END;
```

# Subtypes in SQL

---

General form:

```
CREATE TYPE typeName UNDER inheritType AS ...
```

Example:

```
CREATE TYPE StaffType UNDER PersonType AS (  
    staffNo      VARCHAR(5),  
    salary       DECIMAL(7,2))  
INSTANTIABLE  
NOT FINAL;
```

A subtype inherits all methods and attributes from its supertype. It may override any inherited methods/attributes.

# *Recursive Queries in SQL*

---

General form:

```
WITH RECURSIVE tableName(attr1,attr2,..attrN) AS  
    <SELECT query that defines recursive relation>  
    <SELECT query that uses recursive relation>
```

Example: Return all employees supervised by 'J. Jones'.

```
WITH RECURSIVE supervises(supId,empId) AS  
    (SELECT supereno, eno FROM emp)  
UNION  
    (SELECT S1.supId, S2.empId  
    FROM supervises S1, supervises S2  
    WHERE S1.empId = S2.supId)  
SELECT E1.ename FROM supervises, emp AS E, emp AS E2  
WHERE supervises.supId = E2.eno and E2.ename = 'J. Jones'  
    and supervises.empId = E1.ename;
```

# Conclusion

---

In SQL, different isolation levels can be specified:

- ◆ serializable, repeatable read, read committed, read uncommitted
- ◆ Weaker forms of isolation do not guarantee the ACID properties, but may be useful for read transactions that do not need exact data and require faster execution.

Object-relational DBMSs support user defined data types, active consistency checks (triggers), inheritance, and recursive queries.

# Objectives

---

- ◆ List and explain the isolation levels in SQL.
- ◆ List some features of object-relational DBMSs.