Transactions, Constraints, Logic Code in Database

CSC365

Command Syntax

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]
transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}
BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

Example

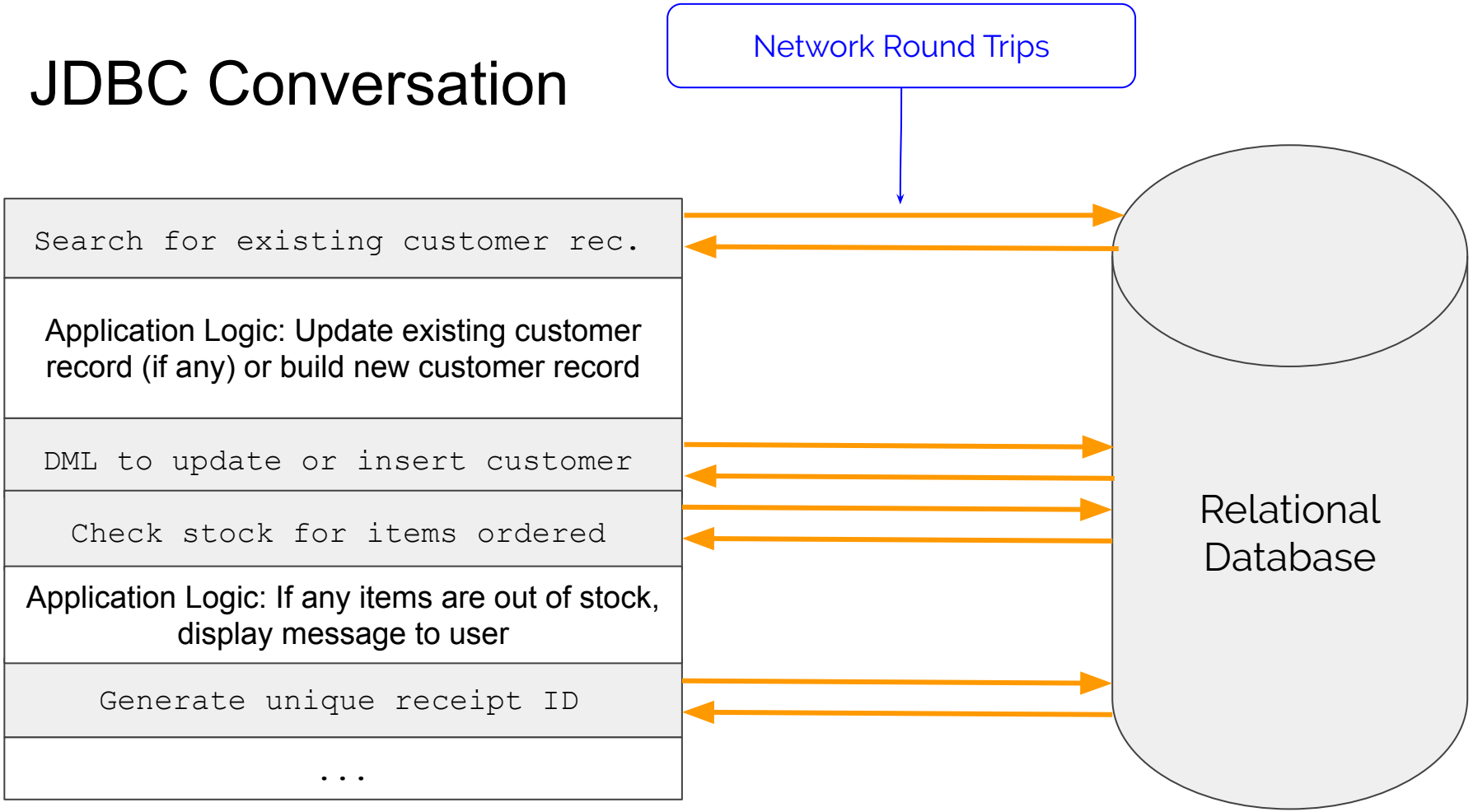
```
START TRANSACTION;

SELECT @A:=SUM(salary) FROM table1 WHERE type=1;

UPDATE table2 SET summary=@A WHERE type=1;

COMMIT;
```

JDBC Applications

A JDBC application typically interacts with a database in a *conversational* way: executing multiple SQL statements, each involving a round trip to the database. Consider a hypothetical customer order:

1. Check whether a customer record exists
2. Update existing customer or create a new customer record
3. Confirm stock level of ordered products
4. Save order detail (receipt & line items)
5. Record payment detail

JDBC Conversation



Multi-Step Conversation: Approaches

- Regular JDBC Statement for each step in process
- Use PreparedStatement to avoid cost of repeated query parse/compile in Java
- Use batch mode to group multiple SQL statements
- Stored Procedures

JDBC - Batch Processing

- The addBatch() method of *Statement, PreparedStatement,* and *CallableStatement* is used to add individual statements to the batch. The executeBatch() is used to start the execution of all the statements grouped together.
- executeBatch() - returns an array of integers, and each element of the array represents the update count for the respective update statement.
- clearBatch() - This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

JDBC Batch Example

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) VALUES (?, ?, ?, ?)";
// Create PrepareStatement object
PreparedStatemen pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);

// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "Pappu" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 33 );
// Add it to the batch
pstmt.addBatch();

//add more batches repeating the Set the Variables and the Add it to the Batch steps
//Create an int[] to hold returned values
int[] count = stmt.executeBatch();
//Explicitly commit statements to apply changes
conn.commit();
```

Stored Procedures

The SQL standard specifies support for *Persistent Stored Modules* (PSM), commonly referred to as Stored Procedures. With this support, procedural code can be stored in the database and executed on the database server. Although the SQL standard defines a standard programming language, vendor implementations are splintered:

- Oracle defines PL/SQL
- MySQL supports a variant of PL/SQL
- Microsoft SQL Server implements Transact-SQL, often abbreviated T-SQL

Stored Procedures - Pros & Cons

Consider "plain" JDBC/SQL versus stored procedures.

Let's list pros & cons of stored procedures.

Stored Procedures - Advantages

- Possible performance gains:
  - Fewer round trips during multi-step conversations
  - Cached / precompiled SQL statements

- Shields applications from details about data structure
  - Security
  - Easier schema evolution

- Common logic which may be shared by multiple applications; no need to re-implement across multiple applications that may share similar needs

Stored Procedures - Disadvantages

- Syntax is often an odd mix of Pascal/Ada and SQL
  - No standardization, every current RDBMS defines its own language
  - Developers must learn specifics
  - Limited extendibility (ie. no "import" capability)
- Debugging and testing are difficult when logic spans both the application and database
- Release / versioning picture is hazy
  - DBAs often reluctant to apply frequent updates due to unknown impact on DB security and stability

Trigger

A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table.

The trigger becomes associated with the table named ***tbl_name***, which must refer to a permanent table.

Triggers are typically written in the procedural language supported by the RDBMS (ie. PL/SQL, T-SQL)

Trigger Events

At a basic level, a trigger may be defined to execute BEFORE or AFTER any of these events:

- INSERT
- UPDATE
- DELETE
- ~~SELECT~~

Each trigger is attached to a single table.

DELETE / UPDATE Triggers

In a DELETE trigger, the OLD keyword allows you to access columns in the row(s) affected by the delete.

UPDATE triggers allow you to reference OLD and NEW:

- OLD refers to the row prior to update (OLD is read-only)
- NEW enables you to access (and possibly change) the updated row values

Example Trigger Use Cases

- Enforce assertions / complex constraints
  - Recall: assertions are constraints that cannot readily be expressed using basic relational model integrity constraints (PK, FK, nullability, CHECK)

- Maintain database integrity
  - Create mandatory "child" records

- Audit logging

Trigger Considerations

Triggers and stored procedures share many of the same pros & cons. Some concerns specific to triggers:

- Performance: trigger code must be extremely efficient, since it runs for *every* record added/updated/removed via DML

- Sequencing of multiple triggers on the same table / same event(s) (if supported)
  - MySQL allows you to specify PRECEDES or FOLLOWS in trigger definition

Using Triggers to Enforce Constraints

It is possible to use BEFORE triggers to prevent an INSERT/UPDATE/DELETE from taking place.

This capability can be used to enforce constraints that go beyond the basic constraints that may be specified in SQL DDL.

Triggers

```
CREATE
    [DEFINER = user]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }
trigger_event: { INSERT | UPDATE | DELETE }
trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

Trigger Examples

```
-- before insert
DELIMITER $
CREATE TRIGGER `creditcard_before_insert` BEFORE INSERT ON
`CreditCard`
FOR EACH ROW
BEGIN
    IF NEW.balance > NEW.limit THEN
        SIGNAL SQLSTATE '99999'
          SET MESSAGE_TEXT = 'check constraint on CreditCard.balance failed';

    END IF;
END$
DELIMITER ;
```

Trigger Examples

```
-- before update
DELIMITER $
CREATE TRIGGER `creditcard_before_update` BEFORE UPDATE ON
`CreditCard`
FOR EACH ROW
BEGIN
    IF NEW.balance > OLD.limit THEN
        SIGNAL SQLSTATE '99999'
          SET MESSAGE_TEXT = 'check constraint on CreditCard.balance failed';

    END IF;
END$
DELIMITER ;
```

Trigger Examples

```
-- before insert
DELIMITER $
CREATE TRIGGER `customer_before_insert` BEFORE INSERT ON `Customer`
FOR EACH ROW
BEGIN
    IF NEW.ssn < 100000000 THEN
        SIGNAL SQLSTATE '12345'
          SET MESSAGE_TEXT = 'check constraint on Customer.ssn failed';

    END IF;
END$
DELIMITER ;
```

Trigger Examples

```
-- before update
DELIMITER $
CREATE TRIGGER `customer_before_update` BEFORE UPDATE ON `Customer`
FOR EACH ROW
BEGIN
    IF NEW.ssn < 100000000 THEN
        SIGNAL SQLSTATE '12345'
          SET MESSAGE_TEXT = 'check constraint on Customer.ssn failed';

    END IF;
END$
DELIMITER ;
```

Stored Procedure Syntax

```
CREATE
    [DEFINER = user]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body
CREATE
    [DEFINER = user]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body
proc_parameter:
    [ IN | OUT | INOUT ] param_name type
func_parameter:
    param_name type
type:
    Any valid MySQL data type
characteristic:
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA |
MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
```

```
routine_body:
    Valid SQL routine statement
```

Stored Procedure Example

```
DELIMITER $
CREATE PROCEDURE `check_balance`(IN balance DECIMAL(10,2), IN lim
DECIMAL(10,2))
BEGIN
    IF balance > lim THEN
        SIGNAL SQLSTATE '99999'
          SET MESSAGE_TEXT = 'check constraint on CreditCard.balance failed';
    END IF;
END$
DELIMITER ;
```

Stored Procedure Example

```
-- before update
DELIMITER $
CREATE TRIGGER `creditcard_before_update` BEFORE UPDATE ON
`CreditCard`
FOR EACH ROW
BEGIN
    CALL check_balance(NEW.balance, OLD.limit);
END$
DELIMITER ;
```

Functions

In addition to stored procedures, many databases support user-defined **functions.** Functions are stored in the database catalog, again using an RDBMS-specific language such as PL/SQL.

Unlike stored procedures, functions must return a value and may be used within SELECT statements.

User-Defined Function Example

```
-- DROP FUNCTION miles_to_km;
-- No ; chars in the function body, no need to change delimiter
CREATE FUNCTION miles_to_km (miles DECIMAL(15,5))
RETURNS DECIMAL(15,5) DETERMINISTIC
  RETURN miles * 1.60934;



SELECT miles_to_km(100.0);
```

`DETERMINISTIC` tells MySQL that the function always produces the same result for the same input

Dropping Triggers, Stored Procedures

```
DROP TRIGGER `trigger_name`;

DROP PROCEDURE `procedure_name`;

DROP FUNCTION `function_name`;
```

Constraints

Expressing Constraints in Relational Algebra

- $R = \varnothing$
  - “The value of R must be empty.”
- $R \subseteq S$
  - “Every tuple in the result of R must also be in the result of S.”

Algebraic Primary Key Constraint

The primary key (PK) must be *unique* across all tuples. Following from this, if two tuples agree on the PK, they must also agree on *all* other attributes.

Expressed as an algebraic constraint:

$$\sigma_{\textit{R1.<PK> = R2.<PK> AND R1.<non key attribute> != R2.<non key attribute>}} (\varrho_{R1}(R) \times \varrho_{R2}(R)) = \varnothing$$

Referential Integrity Constraints

A foreign key (FK) value must also be a key value of a relation in the same database. Expressing this as an algebraic constraint using set containment: $\pi_{FK}(R) \subseteq \pi_{K}(S)$

or, equivalently:

$$\pi_{FK}(R) - \pi_{K}(S) = \varnothing$$

$\pi_{owner\_id}(\text{Ownership}) \subseteq \pi_{id}(\text{Customer})$ or
$\pi_{owner\_id}(\text{Ownership}) - \pi_{id}(\text{Customer}) = \varnothing$

Expressing Other Constraints

The balance on a Credit Card must be less than its credit limit

$$\sigma_{R.limit < R.balance}(\varrho_R(\text{CreditCard})) = \varnothing$$

Setting Constraints in DDL (SQL)

```
CREATE TABLE CreditCard (
    number INT NOT NULL AUTO_INCREMENT,
    type ENUM('Visa', 'Master Card', 'American Express', 'Discover') NOT NULL,
    `limit` INT,
    balance INT,
    active TINYINT(1),
    PRIMARY KEY (number),
    CHECK (`limit` >= balance)
 );
```

CHECK is now supported by MySQL as of v8.0.
Use triggers in older versions.