INFSCI 1022
Database Management Systems

Today’s Evil Plan

- Control flow functions
- Transactions, rollbacks and commits
- Triggers

Control Flow Functions

Commit

Transaction

Rollback

Triggers

Control Flow Functions

- CASE – Case operator
- IF() – If/else construct
- IFNULL() – Null if/else construct
- NULLIF() – Return NULL if expr1 = expr2

CASE / WHEN

- CASE value WHEN [compare_value] THEN result [WHEN [compare_value] THEN result ...] [ELSE result] END

- CASE WHEN [condition] THEN result [WHEN [condition] THEN result ...] [ELSE result] END

CASE / WHEN

```
CASE value WHEN [compare_value] THEN result
    [WHEN [compare_value] THEN result ...]
    [ELSE result]
END
```

- Returns the result where value = compare_value.

CASE / WHEN

```
CASE WHEN [condition] THEN result
    [WHEN [condition] THEN result ...]
    [ELSE result]
END
```

- Returns the result for the first condition that is true. If there was no matching result value, the result after ELSE is returned, or NULL if there is no ELSE part.

CASE / WHEN

```
SELECT CASE 1
        WHEN 1 THEN 'one’
        WHEN 2 THEN 'two'
        ELSE 'more’
END;
```

Will return 'one’

CASE / WHEN

```
SELECT CASE WHEN 1>0 THEN 'true'
            ELSE 'false'
END;
```

Will return 'true’

CASE / WHEN

```
SELECT patientID,
CASE
    WHEN l3 = '1' THEN 3
    WHEN l2 = '1' THEN 2
    WHEN l1 = '1' THEN 1
    ELSE 0
END AS DISEASE_LOCATION
FROM patient_info;
```

IF()

IF(expr1,expr2,expr3)

- If expr1 is TRUE (expr1 <> 0 and expr1 <> NULL) then IF() returns expr2; otherwise it returns expr3.
- IF() returns a numeric or string value, depending on the context in which it is used.

IF()

- SELECT IF(1>2,2,3); $\longrightarrow$ Returns **3**
- SELECT IF(1<2,'yes','no'); $\longrightarrow$ Returns **'yes’**
- SELECT IF(STRCMP('test','test1'),'no','yes'); $\longrightarrow$ Returns **'no'**

IF()

SELECT orderNumber, orderDate, IF(YEAR(orderDate) <= 2003, 'old', 'new') FROM orders;

- Will return create a column that will display ‘new’ for every order that was placed after 2003 and ‘old’ for every order placed before or during 2003.

IFNULL()

IFNULL(expr1,expr2)

- If expr1 is not NULL, IFNULL() returns expr1; otherwise it returns expr2.
- IFNULL() returns a numeric or string value, depending on the context in which it is used.

IFNULL()

- SELECT IFNULL(1,0); ⟶ Returns **1**
- SELECT IFNULL(NULL,10); ⟶ Returns **10**
- SELECT IFNULL(1/0,10); ⟶ Returns **10**
- SELECT IFNULL(1/0,'yes'); ⟶ Returns **‘yes’**

IFNULL()

fk_patientID	fk_labID	result	rangeLow	rangeHigh	flag
1	1	27	20	30	N
2	1	34	20	30	H
3	1	19	20	30	L
2	1	22	20	30	*NULL*

IFNULL()

fk_patientID	fk_labID	result	rangeLow	rangeHigh	flag
1	1	27	20	30	N
2	1	34	20	30	H
3	1	19	20	30	L
2	1	22	20	30	*NULL*

```
SELECT fk_patientID, fk_labID, result, rangeLow, rangeHigh,
IFNULL(flag, 'U')
FROM patient_labs
```

IFNULL()

```
SELECT fk_patientID, fk_labID, result, rangeLow, rangeHigh,
IFNULL(flag, 'U')
FROM patient_labs
```

fk_patientID	fk_labID	result	rangeLow	rangeHigh	flag
1	1	27	20	30	N
2	1	34	20	30	H
3	1	19	20	30	L
2	1	22	20	30	***U***

NULLIF()

NULLIF(expr1,expr2)

- Returns NULL if expr1 = expr2 is true, otherwise returns expr1.
- This is the same as

```
CASE WHEN expr1 = expr2 THEN NULL
    ELSE expr1
END
```

NULLIF()

- SELECT NULLIF(1,1); ⟶ Returns **NULL**
- SELECT NULLIF(1,2); ⟶ Returns **1**

Control Flow Functions

Commit

Transaction

Rollback

Triggers

COMMIT

- By default, MySQL commits (saves) changes generated by INSERT/UPDATE/DELETE queries automatically
- To disable automatic commits of your query, you need to set autocommit to OFF

```
SET autocommit = 0;
```

Commit

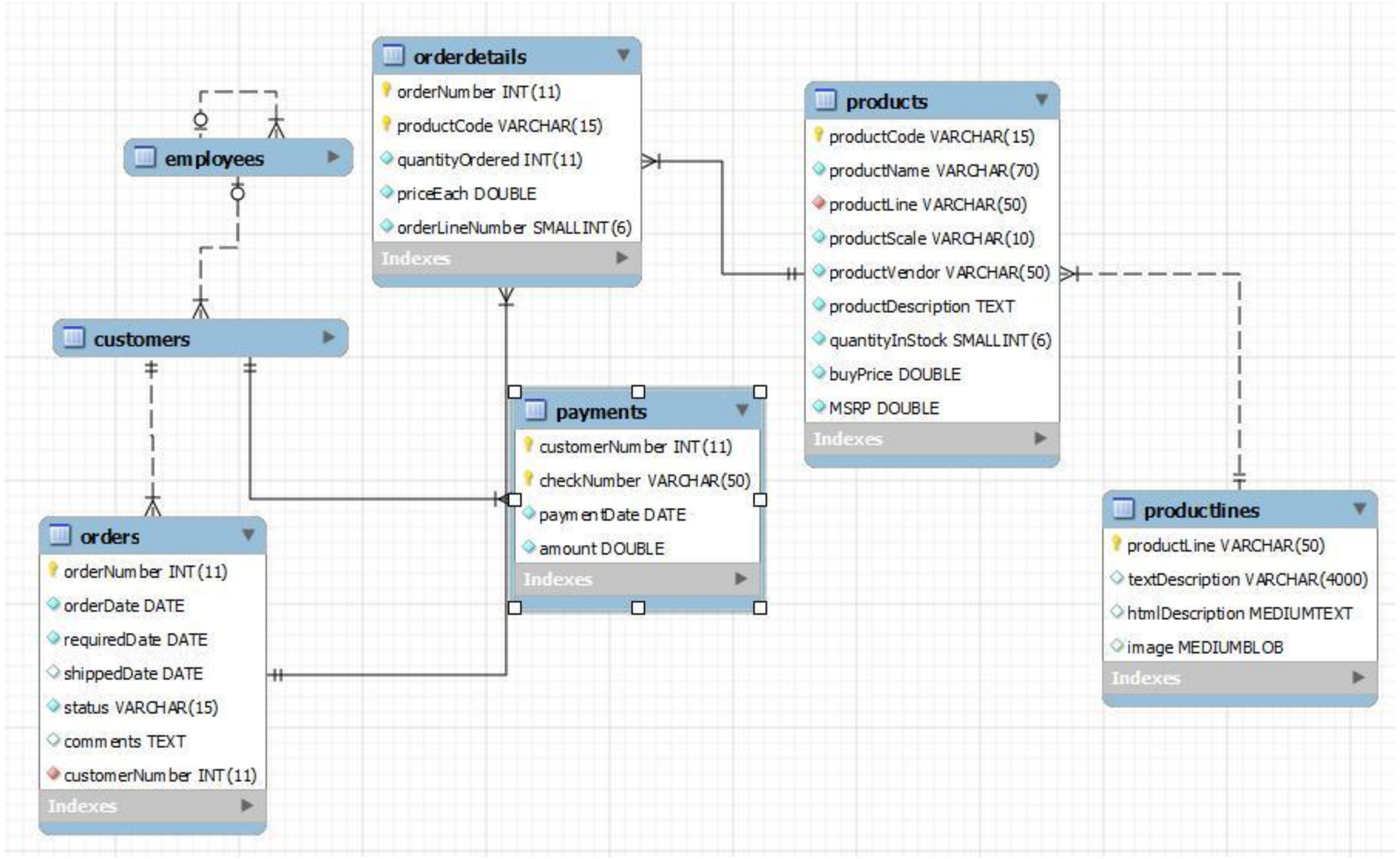
Control Flow Functions

Transaction

Rollback

Triggers

Order Tracking System



To Place an Order

1. Get latest sale order number from ***orders*** table, and use the next sale order number as the new sale order number.
2. Insert a new sale order into ***orders*** table for a given customer
3. Insert new sale order items into ***orderdetails*** table
4. Get data from both table ***orders*** and ***orderdetails*** tables to confirm the changes

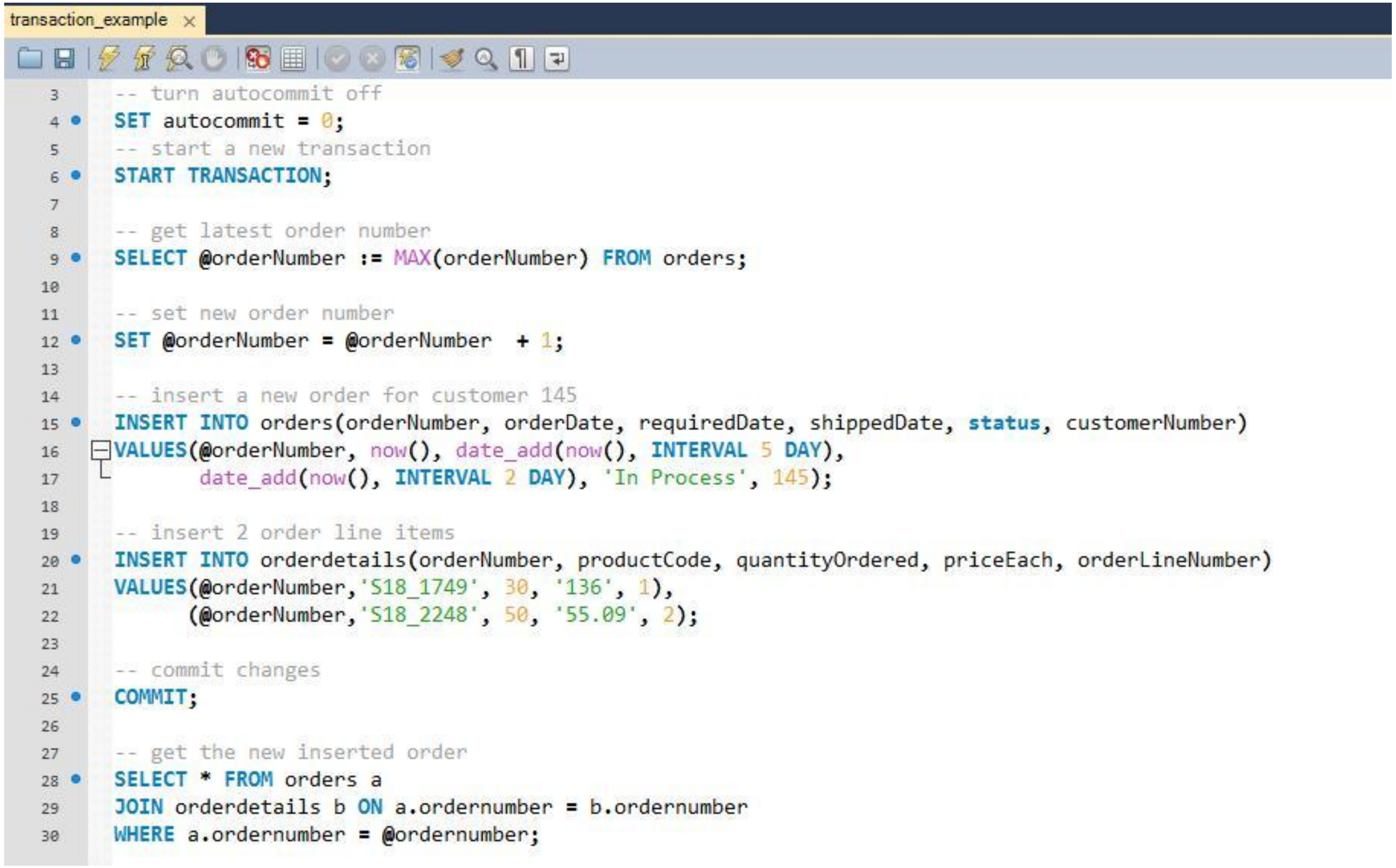
Transactions

- MySQL **transaction** enables you to execute a set of MySQL operations to ensure that the database never contains the result of partial operations.
- In a set of operations, if one of them fails, the rollback occurs to restore the database.
- If no error occurred, the entire set of statements is committed to the database.

Using MySQL Transaction

1. Start by setting autocommit to off: ***SET autocommit = 0;***
2. Start a transaction using ***START TRANSACTION*** statement.
3. Get latest sale order number from orders table, and use the next sale order number as the new sale order number.
4. Insert a new sale order into orders table for a given customer.
5. Insert new sale order items into orderdetails table.
6. Commit changes using COMMIT statement.
7. Get data from both table orders and orderdetails tables to confirm the changes.

Transaction Example



```sql
-- turn autocommit off
SET autocommit = 0;
-- start a new transaction
START TRANSACTION;

-- get latest order number
SELECT @orderNumber := MAX(orderNumber) FROM orders;

-- set new order number
SET @orderNumber = @orderNumber  + 1;

-- insert a new order for customer 145
INSERT INTO orders(orderNumber, orderDate, requiredDate, shippedDate, status, customerNumber)
VALUES(@orderNumber, now(), date_add(now(), INTERVAL 5 DAY),
       date_add(now(), INTERVAL 2 DAY), 'In Process', 145);

-- insert 2 order line items
INSERT INTO orderdetails(orderNumber, productCode, quantityOrdered, priceEach, orderLineNumber)
VALUES(@orderNumber,'S18_1749', 30, '136', 1),
      (@orderNumber,'S18_2248', 50, '55.09', 2);

-- commit changes
COMMIT;

-- get the new inserted order
SELECT * FROM orders a
JOIN orderdetails b ON a.ordernumber = b.ordernumber
WHERE a.ordernumber = @ordernumber;
```

MySQL Transaction Documentation

- http://dev.mysql.com/doc/refman/5.0/en/commit.html

Control Flow Functions

Commit

Transaction

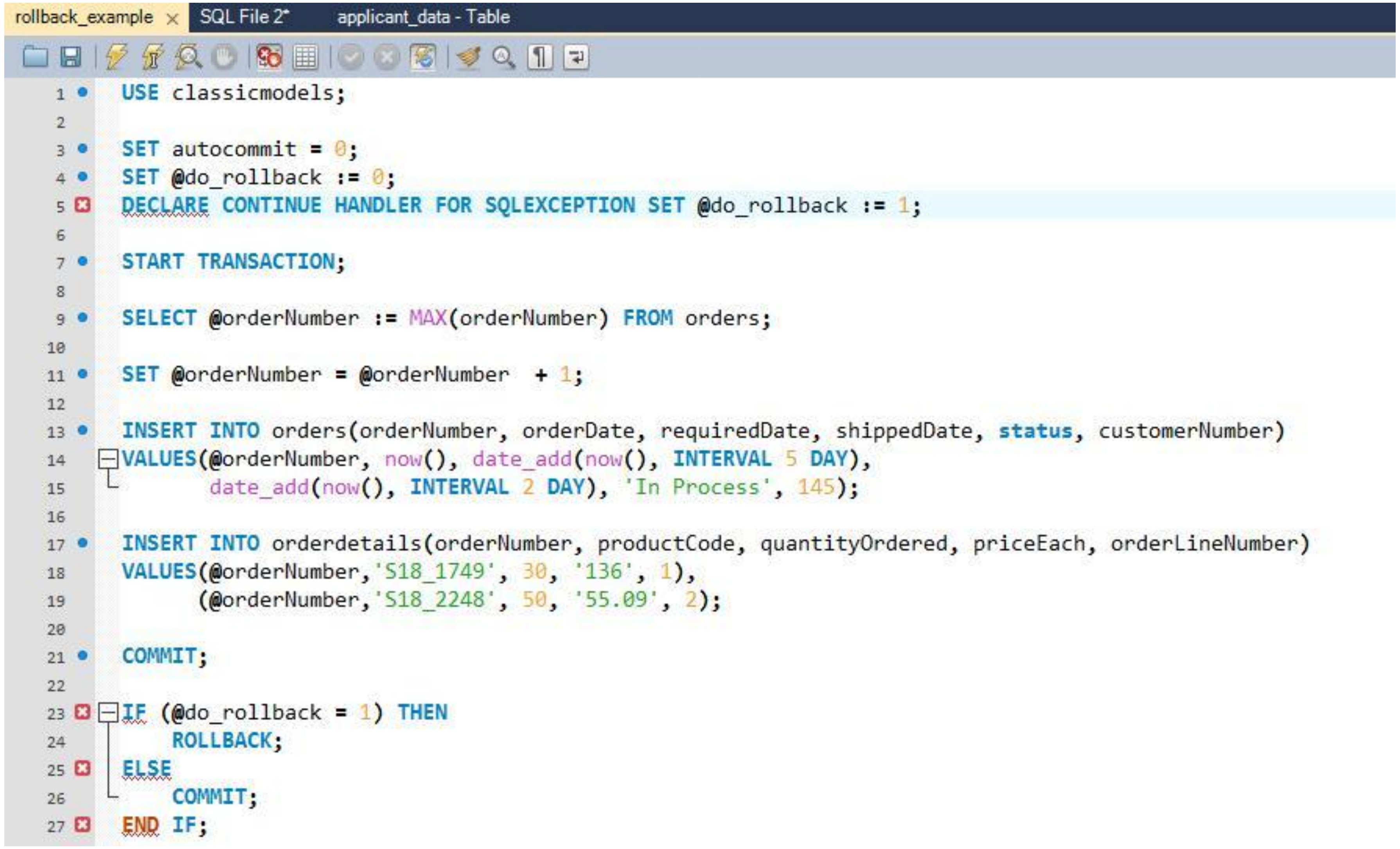
Rollback

Triggers

Rollback

- Rollback allows you to revert/undo everything that was done between commits.
- Even if you don’t explicitly tell your transaction to rollback, it will do so automatically (eventually).
- For better memory management, you should explicitly rollback failed transactions.

Rollback



```
USE classicmodels;

SET autocommit = 0;
SET @do_rollback := 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET @do_rollback := 1;

START TRANSACTION;

SELECT @orderNumber := MAX(orderNumber) FROM orders;

SET @orderNumber = @orderNumber  + 1;

INSERT INTO orders(orderNumber, orderDate, requiredDate, shippedDate, status, customerNumber)
VALUES(@orderNumber, now(), date_add(now(), INTERVAL 5 DAY),
       date_add(now(), INTERVAL 2 DAY), 'In Process', 145);

INSERT INTO orderdetails(orderNumber, productCode, quantityOrdered, priceEach, orderLineNumber)
VALUES(@orderNumber,'S18_1749', 30, '136', 1),
      (@orderNumber,'S18_2248', 50, '55.09', 2);

COMMIT;

IF (@do_rollback = 1) THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;
```

Control Flow Functions

Commit

Transaction

Rollback

Triggers

Triggers

- Trigger is a set of SQL statements stored in the database catalog.
- Trigger is executed or fired whenever an event associated with a table occurs e.g., insert, update or delete.

Triggers

- Trigger is a special type of stored procedure.
- It is not called directly like a stored procedure.
  - Trigger is called automatically when a data modification event is made against a table
  - SP must be called explicitly.

Advantages of Triggers

- Triggers provide an alternative way to check the integrity of data.
- Can catch errors in business logic in the database layer.
- Provide an alternative way to run scheduled tasks.
- Very useful to audit the changes of data in tables.

Disadvantages of Triggers

- Triggers only can provide an extended validation and they cannot replace all the validations.
  - Some simple validations have to be done in the application layer. For example, you can validate user’s inputs in the client side by using JavaScript or in the server side using server side scripting languages such as JSP, PHP, ASP.NET, Perl, etc.
- SQL triggers are invoked and executed invisibly from client-applications therefore it is difficult to figure out what happen in the database layer.
- SQL triggers may increase the overhead of the database server.

Trigger Activation

- BEFORE INSERT – activated before data is inserted into the table.
- AFTER INSERT– activated after data is inserted into the table.
- BEFORE UPDATE – activated before data in the table is updated.
- AFTER UPDATE – activated after data in the table is updated.
- BEFORE DELETE – activated before data is removed from the table.
- AFTER DELETE – activated after data is removed from the table.

Create Trigger

```
CREATE TRIGGER trigger_name trigger_time trigger_event
    ON table_name  FOR EACH ROW
 BEGIN


 END
```

Create Trigger

- You put the trigger name after the CREATE TRIGGER statement.
  - The trigger name should follow the naming convention [trigger time]_[table name]_[trigger event], for example before_employees_update.
- Trigger activation time can be BEFORE or AFTER.
  - Must specify the activation time when you define a trigger.
  - Use BEFORE keyword if you want to process action prior to the change is made on the table
  - Use AFTER if you need to process action after the change is made

Create Trigger

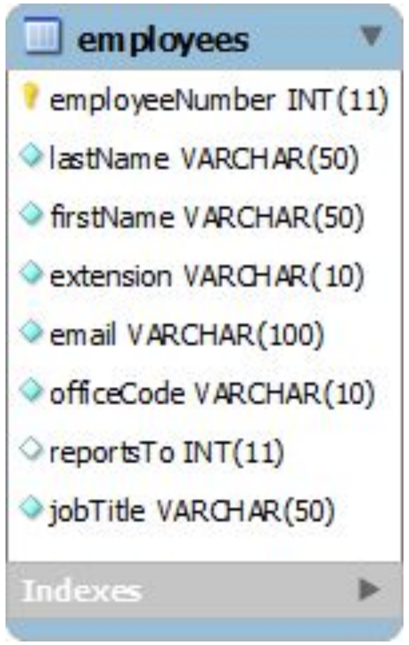
- Trigger event can be INSERT, UPDATE or DELETE.
- This event causes trigger to be invoked. A trigger only can be invoked by one event.
- To define a trigger that is invoked by multiple events, you have to define multiple triggers, one for each event.

Create Trigger

- A trigger must be associated with a specific table.
- The SQL statements are placed between BEGIN and END block.

Trigger Example

- We have employees table in our *classicmodels* database as follows:



employees

- employeeNumber INT(11)
- lastName VARCHAR(50)
- firstName VARCHAR(50)
- extension VARCHAR(10)
- email VARCHAR(100)
- officeCode VARCHAR(10)
- reportsTo INT(11)
- jobTitle VARCHAR(50)

Indexes

Trigger Example

- Create a new table named *employees_audit* to keep the changes of the employee records. The following script creates the *employee_audit* table.

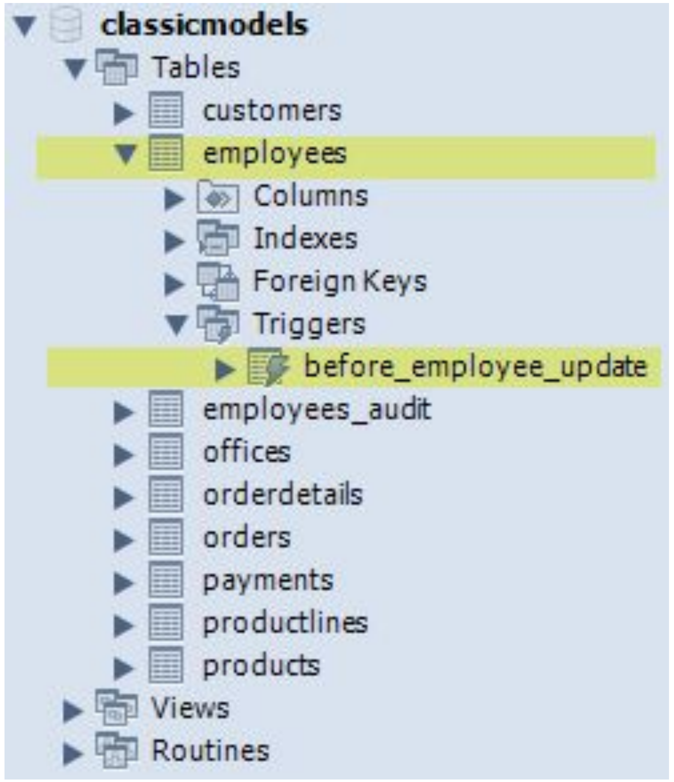
```
CREATE TABLE employees_audit (
        id int(11) NOT NULL AUTO_INCREMENT,
        employeeNumber int(11) NOT NULL,
        lastname varchar(50) NOT NULL,
        changedon datetime DEFAULT NULL,
        action varchar(50) DEFAULT NULL,
        PRIMARY KEY (id)
)
```

Trigger Example

Create a BEFORE UPDATE trigger to be invoked before a change is made to the employees table.

```
DELIMITER $$
CREATE TRIGGER before_employee_update
   BEFORE UPDATE ON employees
   FOR EACH ROW BEGIN

   INSERT INTO employees_audit
   SET action = 'update',
    employeeNumber = OLD.employeeNumber,
      lastname = OLD.lastname,
      changedon = NOW();
END$$
DELIMITER ;
```



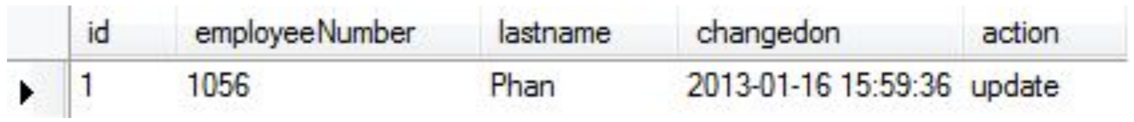
```
▼ classicmodels
  ▼ Tables
    ▶ customers
    ▼ employees
      ▶ Columns
      ▶ Indexes
      ▶ Foreign Keys
      ▼ Triggers
        ▶ before_employee_update
    ▶ employees_audit
    ▶ offices
    ▶ orderdetails
    ▶ orders
    ▶ payments
    ▶ productlines
    ▶ products
  ▶ Views
  ▶ Routines
```

Trigger Example

UPDATE employees SET lastName = 'Phan’ WHERE employeeNumber = 1056

To check if the trigger was invoked by the UPDATE statement, we can query the employees_audit table by using the following query:

*SELECT * FROM employees_audit;*



id	employeeNumber	lastname	changedon	action
1	1056	Phan	2013-01-16 15:59:36	update