SQL

Microsoft SQL Server 2018

Lab 1

CREATE, DROP, ALTER, INSERT, UPDATE, DELETE

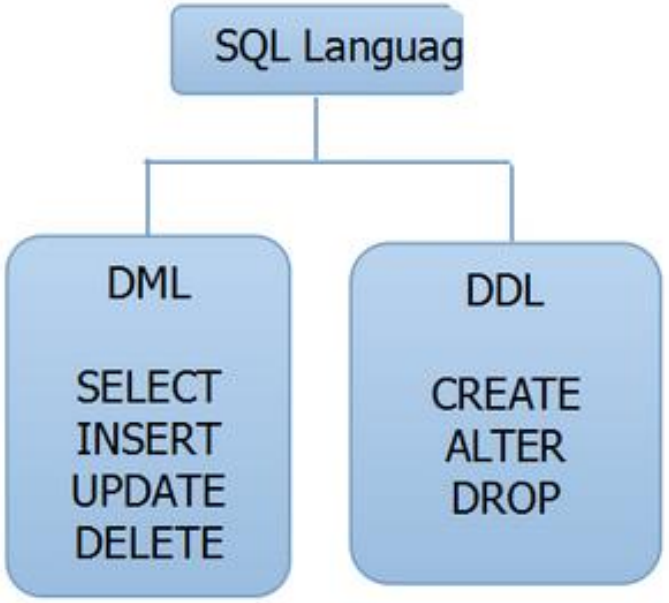
SQL vs. NoSQL

- SQL databases are **relational**, NoSQL databases are **non-relational**.
- SQL databases use **structured** query language and have a **predefined schema**. NoSQL databases have **dynamic schemas** for **unstructured** data.
- SQL databases are **vertically scalable**, while NoSQL databases are **horizontally scalable**.
- SQL databases are **table-based**, while NoSQL databases are **document, key-value, graph, or wide-column stores**.
- SQL databases are better for **multi-row transactions**, while NoSQL is better for **unstructured** data like documents or **JSON**.

DML vs DDL

DDL is Data Definition Language
which is used to define **data structures.**

DML is Data Manipulation Language
which is used to **manipulate data itself**



Database

- Create a database called TestDb
- `CREATE DATABASE TestDb;`

- Remove a database called TestDb
- `DROP DATABASE TestDb;`

- View all databases
- `SELECT name FROM master.sys.databases`

Comment

- `/* … */`

Table

- Create a table
- CREATE TABLE student

```
(stud_id CHAR(5),
fname varCHAR(10) NOT NULL,
lname varCHAR(8),
marks int check (mark>1 and mark<=100),
dep varchar(15)
PRIMARY KEY (stud_id));
```

Remark: column marks only accepts a value between 1 and 100

- Remove a table
- DROP TABLE student

- Create a table
- CREATE TABLE student

```
(stud_id int IDENTITY(1,1) PRIMARY KEY,
fname varCHAR(10) NOT NULL,
lname varCHAR(8),
dep varchar(15) check(dep in ('cs', 'se')),
email varchar(50) UNIQUE);
```

Note: IDENTITY(1,1) means column stud_id starts from 1 and increment by 1 and we don't need to insert stud_ id, the system generates automatically.

- Rename column name

```
SP_RENAME 'STUDENT.fname','firstname','column'
```

Alter

- Change table structure by adding a new column
- `ALTER TABLE student ADD marks int;`
- `ALTER TABLE student ADD description VARCHAR(255);`

- Change table structure by removing a colum
- `ALTER TABLE student DROP COLUMN description;`

- Change table structure by changing column data type(or max string length - 50)
- `ALTER TABLE student ALTER COLUMN lname VARCHAR (50);`

Insert

- Insert values to all columns
- Insert Into student values ('aaa','aaa',25,'math','aaa@gmail.com')
- Insert Into student values ('bbb','bbb',25,'math','bbb@gmail.com')
- Insert Into student values ('ccc','ccc',25,'math','ccc@gmail.com’)

Remark: student table has stud_id column but we don’t need to insert a value because it is IDENTITY

- Insert values to specifically first name and email columns
- INSERT INTO student (firstname,email) VALUES ('ddd', 'ddd@gmail.com')

Select

- View all data in student table
- SELECT * FROM student

Update

- Update dep value from student table if stud_id is 1
- `UPDATE student SET firstname='AAA' WHERE stud_id=1`

- Update all dep values on student table
- `UPDATE student SET dep='CS'`

Delete

- Delete a student information if stud_id is 3
- DELETE FROM student WHERE stud_id = 3

- Delete all values from student table
- DELETE FROM student

Note: if the table has IDENTITY column after DELETE FROM student it looses all the data then when we insert a new row the value of IDENTITY column will continue from the previous one

- Delete the data from student table with table log file
- TRUNCATE TABLE student

Note: if the table has IDENTITY column after TRUNCATE the IDENTITY column values starts from 1

Lab 2

SELECT

Pre request

- Load the file called For students.sql to Microsoft Sql Server 2019
- Execute all queries

Select Syntax

- `SELECT` **{*field-list* | `*` | `ALL` | `DISTINCT` | expression}**

  `FROM` **table-list**

  `WHERE` **expression**

  `GROUP BY` **group-fields**

  `HAVING` **group-expression**

  `ORDER BY` **field-list;**

Select - 1

- Display all customer information
- `SELECT * FROM customers;`

- Display only first 50 customer information
- `SELECT TOP(50) * FROM customers;`

- Display first name and phone of all customers
- `SELECT first_name, phone FROM customers;`

- Display first name and phone of first 50 customers
- `SELECT TOP(50) first_name, phone FROM customers;`

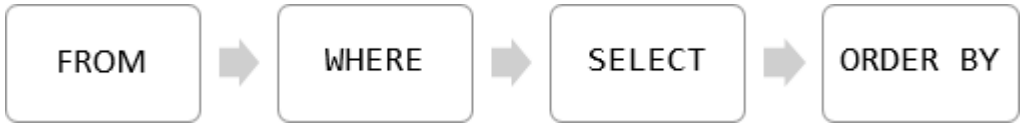
- Display first name and phone of 5 percent of customers
- `SELECT TOP 5 PERCENT first_name, phone FROM customers;`

Select - 2

- Display all customers who are locateed in California (CA)
- `SELECT * FROM customers WHERE state = 'CA';`

- sorts the above result by their first names in ascending order.
- `SELECT * FROM customers WHERE state = 'CA' ORDER BY first_name; /*ASC*/`

- sorts the above result by their first names in descending order.
- `SELECT * FROM customers WHERE state = 'CA' ORDER BY first_name DESC;`



Select - 3

- Returns all the cites of customers located in California and the number of customers in each city.
- `SELECT city, COUNT (*) FROM customers WHERE state = 'CA' GROUP BY city ORDER BY city;`

- Returns the city in California which has more than 10 customers:

```
SELECT city, COUNT (*) AS 'Count' FROM customers WHERE state = 'CA' GROUP BY city
HAVING COUNT (*) > 10 ORDER BY city;
```

Notice that the `WHERE` clause filters rows while the `HAVING` clause filter groups.

Select - 4

- Display city, first name and last and sort the list by the city first and then by the first name.
- `SELECT city, first_name, last_name FROM customers ORDER BY city, first_name;`
- It is possible to sort the result set by a column that does not appear on the select list

- Retrieve a customer list sorted by the length of the first name.
- `SELECT first_name, last_name FROM customers ORDER BY LEN(first_name) DESC;`

- The following statement sorts the customers by first name and city.
- `SELECT first_name, last_name, state FROM customers ORDER BY 1, 3;`
- In this example, 1 means the `first_name` column and 3 means the `state` column

Select - 5

- Returns a distinct cities
- `SELECT DISTINCT city FROM customers ORDER BY city;`
- It is possible to sort the result set by a column that does not appear on the select list

- Display the distinct city and state of all customers.
- `SELECT DISTINCT city, state FROM customers ORDER BY city, state;`
- In this example, the statement used the combination of values in both `city` and `state` columns to evaluate the duplicate.

- Returns a distinct phone
- `SELECT DISTINCT phone FROM customers ORDER BY phone;`

```
SELECT city FROM customers GROUP BY city ORDER BY city ;
The same as SELECT DISTINCT city FROM customers ORDER BY city;
But use GROUP BY when we want aggregate functions
```

Select - 6

- Retrieves all products with the category id 1
- `SELECT * FROM products WHERE category_id = 1;`

- Retrieves all products with the category id 1 and the model is 2018
- `SELECT * FROM products WHERE category_id = 1 AND model_year = 2018`

- Finds the products that have list price is greater than 300 and model is 2018
- `SELECT * FROM products WHERE list_price > 300 AND model_year = 2018`

- Finds the products that have list prices are between 1,899 and 1,999.99
- `SELECT * FROM products WHERE list_price BETWEEN 1899.00 AND 1999.99`

Select - 7

- Find all products of list price is 299.99 or 466.99 or 489.99.
- `SELECT * FROM products WHERE list_price IN (299.99, 369.99, 489.99)`

- Find all products name that starts with in the string ‘Electra’
- `SELECT * FROM products WHERE product_name LIKE 'Electra%'`

- Find all products name that ends with in the string ‘2018’
- `SELECT * FROM products WHERE product_name LIKE ‘%2018’`

- Find all products name that contains the string ‘Cruiser’
- `SELECT * FROM products WHERE product_name LIKE '%Cruiser%'`

Select - 8

- Returned the customers who don’t have phone information.
- `SELECT first_name, phone FROM customers WHERE phone IS NULL;`

- Returned the customers who have phone information.
- `SELECT first_name, phone FROM customers WHERE phone IS NOT NULL`

- Return product whose brand id is 1 or 2 and list price is larger than 1,000
- `SELECT * FROM products WHERE (brand_id = 1 OR brand_id = 2) AND list_price > 1000`

- Return products that have a quantity greater than or equal to 30 in stock
- `SELECT * FROM products WHERE product_id IN (SELECT product_id FROM stocks WHERE quantity >= 30)`

Select - 9

- Finds the customers where the first character in the last name is the letter in the range ‘A’ through ‘C’
- `SELECT first_name, last_name FROM customers WHERE last_name LIKE '[A-C]%'`

- Finds the customers where the first character in the last name is not the letter in the range ‘A’ through ‘X’
- `SELECT first_name, last_name FROM customers WHERE last_name LIKE ‘[^A-X]%’`

- Find customers where the first character in the first name is not the letter ‘A’
- `SELECT first_name, last_name FROM customers WHERE first_name NOT LIKE 'A%’`

- Display first name and last name in one column
- `SELECT first_name + ' ' + last_name FROM customers ORDER BY first_name;`

Select - 10

- Display first name and last name in one column and rename column name
- SELECT first_name + ' ' + last_name AS 'Full Name' FROM customers ORDER BY first_name;

- Display first name and rename column name
- SELECT first_name FN from customers

- Return the average list price of all products in the products table
- SELECT AVG(list_price) avg_product_price FROM products;

- Return the number of products whose price is greater than 500
- SELECT COUNT(*) product_count FROM products WHERE list_price > 500;

Select - 11

- Return the highest list price of all products
- `SELECT MAX(list_price) max_list_price FROM products;`

- Return the lowest list price of all products
- `SELECT MIN(list_price) min_list_price FROM products;`

- Return the sum of quantities of each product list in stocks table
- `SELECT product_id, SUM(quantity) stock_count FROM stocks GROUP BY product_id;`

Lab 3

JOIN, VIEW, STORED PROCEDURE, TRIGGER, TRANSACTION

Create Table – insert multiple row

- Create table

```
CREATE TABLE candidates(
    id INT PRIMARY KEY IDENTITY,
    fullname VARCHAR(100) NOT NULL
);
```

- Insert multiple row at one

```
INSERT INTO
    candidates(fullname)
VALUES
    ('John Doe'),
    ('Lily Bush'),
    ('Peter Drucker'),
    ('Jane Doe');
```

- Create table

```
CREATE TABLE employees(
    id INT PRIMARY KEY IDENTITY,
    fullname VARCHAR(100) NOT NULL
);
```

- Insert multiple row at one

```
INSERT INTO
    employees(fullname)
VALUES
    ('John Doe'),
    ('Jane Doe'),
    ('Michael Scott'),
    ('Jack Sparrow');
```

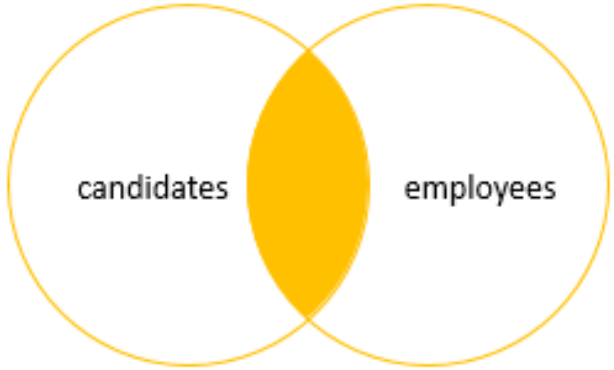
Join / Inner Join

```
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    INNER JOIN employees e
        ON e.fullname = c.fullname;
```

Note: ‘candidate c’ means renaming column

Note: ‘c.id’ because id column found on both employees table and candidates table, c.id means id column found on candidate table

candidate_id	candidate_name	employee_id	employee_name
1	John Doe	1	John Doe
4	Jane Doe	2	Jane Doe

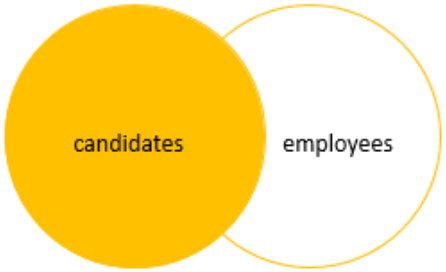


Remark: INNER JOIN produces a data set that includes rows from the left table which have matching rows from the right table.

Left Join

```
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    LEFT JOIN employees e
        ON e.fullname = c.fullname;
```

candidate_id	candidate_name	employee_id	employee_name
1	John Doe	1	John Doe
2	Lily Bush	NULL	NULL
3	Peter Drucker	NULL	NULL
4	Jane Doe	2	Jane Doe

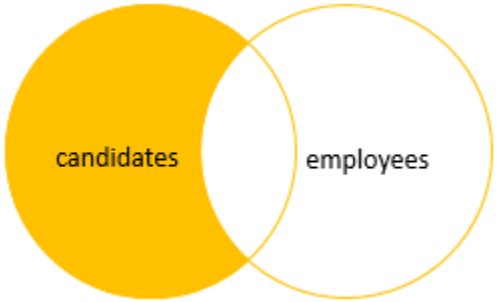


Note: LEFT JOIN selects data starting from the left table and matching rows in the right table. The left join returns all rows from the left table and the matching rows from the right table. If a row in the left table does not have a matching row in the right table, the columns of the right table will have nulls.

Left Join – Only the left side

```
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    LEFT JOIN employees e
        ON e.fullname = c.fullname
WHERE
    e.id IS NULL;
```

candidate_id	candidate_name	employee_id	employee_name
2	Lily Bush	NULL	NULL
3	Peter Drucker	NULL	NULL

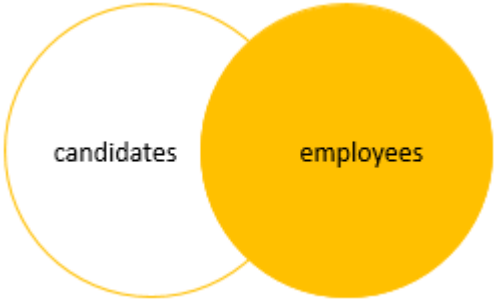


Note: To get the rows that available only in the left table but not in the right table, you add a `WHERE` clause to the above query.

Right Join

```
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    RIGHT JOIN employees e
        ON e.fullname = c.fullname;
```

candidate_id	candidate_name	employee_id	employee_name
1	John Doe	1	John Doe
4	Jane Doe	2	Jane Doe
NULL	NULL	3	Michael Scott
NULL	NULL	4	Jack Sparrow

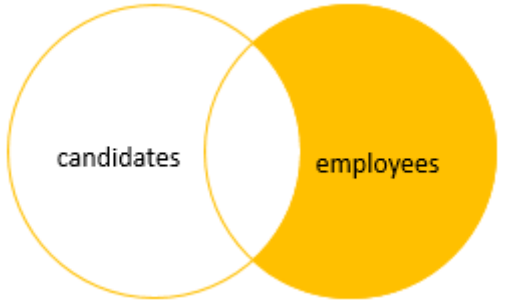


Note: The RIGHT JOIN returns a result set that contains all rows from the right table and the matching rows in the left table. If a row in the right table that does not have a matching row in the left table, all columns in the left table will contain nulls.

Right Join – only the right side

```
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    RIGHT JOIN employees e
        ON e.fullname = c.fullname
WHERE
    c.id IS NULL;
```

candidate_id	candidate_name	employee_id	employee_name
NULL	NULL	3	Michael Scott
NULL	NULL	4	Jack Sparrow

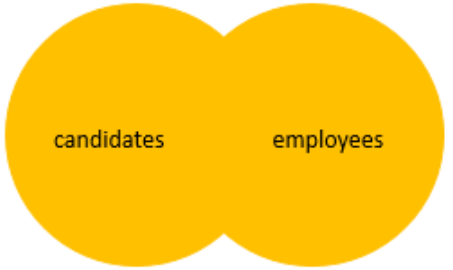


Note: you can get rows that are available only in the right table by adding a `WHERE` clause to the above query

Full Join

```
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    FULL JOIN employees e
        ON e.fullname = c.fullname;
```

candidate_id	candidate_name	employee_id	employee_name
1	John Doe	1	John Doe
2	Lily Bush	NULL	NULL
3	Peter Drucker	NULL	NULL
4	Jane Doe	2	Jane Doe
NULL	NULL	3	Michael Scott
NULL	NULL	4	Jack Sparrow

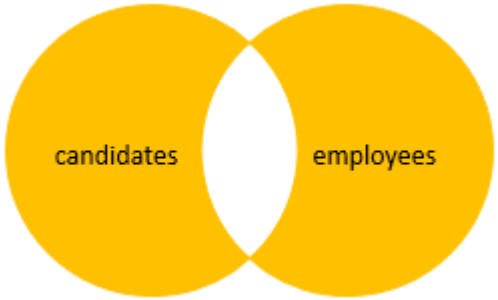


Note: The FULL OUTER JOIN or FULL JOIN returns a result set that contains all rows from both left and right tables, with the matching rows from both sides where available. In case there is no match, the missing side will have NULL values.

Full Join – exclude common rows

```
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    FULL JOIN employees e
        ON e.fullname = c.fullname
WHERE
    c.id IS NULL OR
    e.id IS NULL;
```

candidate_id	candidate_name	employee_id	employee_name
2	Lily Bush	NULL	NULL
3	Peter Drucker	NULL	NULL
NULL	NULL	3	Michael Scott
NULL	NULL	4	Jack Sparrow



Note: To select rows that exist either left or right table, you exclude rows that are common to both tables by adding a `WHERE` clause

View

- Create view

```
CREATE VIEW view_candidate_join_employee
AS
SELECT
    c.id candidate_id,
    c.fullname candidate_name,
    e.id employee_id,
    e.fullname employee_name
FROM
    candidates c
    INNER JOIN employees e
        ON e.fullname = c.fullname;
```

- Later, you can used the view in the SELECT statement like a table as follows:
- `select * from view_candidate_join_employee`

Stored Procedure

- Create user defined stored procedure

```
CREATE PROCEDURE uspProductList
AS
BEGIN
    SELECT
        product_name,
        list_price
    FROM
        products
    ORDER BY
        product_name;
END;
```

- Execute my stored procedure

```
EXEC uspProductList
EXECUTE uspProductList
```

- Execute system created stored procedure

```
EXEC sp_databases;
```

Note: display all databases found in the system

Update Stored Procedure

- Alter stored procedure (change structure)

```
ALTER PROCEDURE uspProductList
AS
BEGIN
    SELECT
        product_name,
        list_price
    FROM
        production.products
    ORDER BY
        list_price
END;
```

- Remove stored procedure

```
DROP PROCEDURE uspProductList;
```

- Both PROCEDURE and PROC are the same

```
DROP PROC uspProductList;
```

Stored Procedure – parameter

- Parameterized stored procedure

```
CREATE PROCEDURE uspFindProducts(
        @min_list_price AS DECIMAL
)
AS
BEGIN
    SELECT product_name, list_price
    FROM products
    WHERE
        list_price >= @min_list_price
    ORDER BY list_price;
END;
```

- Execute procedure with parameter value

```
EXEC uspFindProducts 100;
```

First, we added a parameter named @min_list_price to the uspFindProducts stored procedure. Every parameter must start with the @ sign. The AS DECIMAL keywords specify the data type of the @min_list_price parameter. The parameter must be surrounded by the opening and closing brackets.

Second, we used @min_list_price parameter in the WHERE clause of the SELECT statement to filter only the products whose list prices are greater than or equal to the @min_list_price.

Stored Procedure – multiple parameter

- Multiple parameter in stored procedure

```
ALTER PROCEDURE uspFindProducts(
    @min_list_price AS DECIMAL
    ,@max_list_price AS DECIMAL
)
AS
BEGIN
    SELECT product_name, list_price
    FROM products
    WHERE
        list_price >= @min_list_price AND
        list_price <= @max_list_price
    ORDER BY list_price;
END;
```

- Execute procedure with multiple parameter value

```
EXECUTE uspFindProducts 900, 1000;
```

Remark: 900 for @min_list_price and 1000 for @max_list_price

- Pass value with their name

```
EXECUTE uspFindProducts
    @min_list_price = 900,
    @max_list_price = 1000;
```

Stored Procedure – string parameter

- String (VARCHAR) parameter

```
ALTER PROCEDURE uspFindProducts(
    @min_list_price AS DECIMAL
    ,@max_list_price AS DECIMAL
    ,@name AS VARCHAR(max)
)
AS
BEGIN
    SELECT product_name, list_price
    FROM products
    WHERE
        list_price >= @min_list_price AND
        list_price <= @max_list_price AND
        product_name LIKE '%' + @name + '%'
    ORDER BY list_price;
END;
```

- Execute procedure with multiple parameter value

```
EXECUTE uspFindProducts
    @min_list_price = 900,
    @max_list_price = 1000,
    @name = 'Electra';
```

Stored Procedure – optional parameter

- Optional parameter value in stored procedure

```
ALTER PROCEDURE uspFindProducts(
    @min_list_price AS DECIMAL = 0
    ,@max_list_price AS DECIMAL = 999999
    ,@name AS VARCHAR(max)
)
AS
BEGIN
    SELECT product_name, list_price
    FROM products
    WHERE
        list_price >= @min_list_price AND
        list_price <= @max_list_price AND
        product_name LIKE '%' + @name + '%'
    ORDER BY list_price;
END;
```

- Execute procedure

```
EXECUTE uspFindProducts
    @name = 'Electra’;
```

Remark: @min_list_price is 0 and @max_list_price is 99999

- Execute procedure

```
EXECUTE uspFindProducts
    @min_list_price = 6000,
    @name = 'Electra';
```

Remark: @min_list_price is 6000 and @max_list_price is 99999

Trigger

- Create table

```
CREATE TABLE dep (
    dep_id INT IDENTITY PRIMARY KEY,
    dep_name VARCHAR(5)
);
```

Remark: IDENTITY in dep_id is the same as IDENTITY(1,1)

- Disable all triggers on dep table

```
DISABLE TRIGGER ALL ON dep;
```

- Enable all triggers on dep table

```
ENABLE TRIGGER ALL ON dep;
```

- Create trigger on dep table after insert operation takes

```
CREATE TRIGGER trg_dep_insert
ON dep
AFTER INSERT
AS
BEGIN
    PRINT 'A new member has been inserted';
END;
```

Note: when a data is inserted in dep table a message is printed

Trigger – insert

- Create a table called dep_audit used as a log table for dep

```
CREATE TABLE dep_audit (
    change_id INT IDENTITY PRIMARY KEY,
    dep_id INT NOT NULL,
    dep_name VARCHAR(255) NOT NULL,
    updated_at DATETIME NOT NULL,
    operation CHAR(3) NOT NULL,
    CHECK(operation = 'INS' or operation='DEL')
);
```

Remark: column operation only accepts INS and DEL

- Update trigger

```
ALTER TRIGGER trg_dep_insert
ON dep
AFTER INSERT
AS
BEGIN
   SET NOCOUNT ON;
   INSERT INTO dep_audit(
   dep_id, dep_name, updated_at, operation)
   SELECT i.dep_id, dep_name, GETDATE(), 'INS'
    FROM inserted i
END;
```

Remark: when a data is inserted in dep it will also inserted in dep_audit table

Trigger – delete

- Create trigger

```
CREATE TRIGGER trg_dep_delete
ON dep
AFTER DELETE
AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO dep_audit(
    dep_id, dep_name, updated_at, operation)
    SELECT d.dep_id, dep_name, GETDATE(), 'DEL'
    FROM  deleted d;
END
```

Remark: when a data is deleted from dep, the deleted data will be inserted to dep_audit table

Operation	*deleted* Table	*inserted* Table
INSERT	(not used)	Contains the rows being inserted
DELETE	Contains the rows being deleted	(not used)
UPDATE	Contains the rows as they were before the UPDATE statement	Contains the rows as they were after the UPDATE statement

Note: inside the body of the trigger, you set the SET NOCOUNT to ON to suppress the number of rows affected messages from being returned whenever the trigger is fired.

Trigger – instead of

- Create instead of trigger

```
CREATE TRIGGER trg_insted_of_dep
ON dep
INSTEAD OF INSERT
AS
BEGIN
    SET NOCOUNT ON;
    SELECT * FROM dep
END
```

Note: An INSTEAD OF trigger is a trigger that allows you to skip an INSERT, DELETE, or UPDATE statement to a table or a view and execute other statements defined in the trigger instead. The actual insert, delete, or update operation does not occur at all.

Note: In other words, an INSTEAD OF trigger skips a DML statement and execute other statements.

Transaction

- Create table

```
CREATE TABLE Books (
    id INT,
    name VARCHAR(50) NOT NULL,
    price INT NOT NULL
)
```

- Insert multiple row

```
INSERT INTO Books VALUES
    (1, 'Book1', 1800),
    (2, 'Book2', 1500);
```

Note: Transactions in SQL Server are used to execute a set of SQL statements in a group. With transactions, either all the statements in a group execute or none of the statements execute.

Without Transaction

- Without Transaction

```
INSERT INTO Books VALUES (15, 'Book15', 2000)

UPDATE Books SET price = ‘25 Hundred' WHERE id = 15

DELETE from Books WHERE id = 15
```

Note: In the script above, we execute three queries. The first query inserts a new record in the Books table where the id of the record is 15. The second query updates the price of the book with id 15. Finally, the third query deletes the record with id 15. If you execute the above query, you should see the following error:

```
(1 row affected)
Msg 245, Level 16, State 1, Line 38
Conversion failed when converting the varchar value '25 Hundred' to data type int.

Completion time: 2019-12-07T13:02:19.8746024+09:00
```

The error is pretty self-explanatory. It says that we cannot assign the string value ‘25 Hundred’ to the ‘id’ column, which is of integer type. Hence, the second query fails to execute. However, the problem with the above script is that while the second query fails, the first query still executes.

With Transaction

- With Transaction

```
BEGIN TRANSACTION

  INSERT INTO Books VALUES (20, 'Book5', 2000)

  UPDATE Books SET price = 'Hundred' WHERE id = 20

  DELETE from Books WHERE id = 20

COMMIT TRANSACTION
```

Note: To start a transaction, the BEGIN TRANSACTION statement is used, followed by the set of queries that you want to execute inside the transaction. To mark the end of a transaction, the COMMIT TRANSACTION statement can be used.

In the script above, we execute the same three SQL queries that we did in the last section. However, this time the queries have been executed inside a transaction. Again, the first query will execute successfully and an error will occur while executing the second query. Since the queries are being executed inside a transaction, the failure of the second query will cause all the previously executed queries to rollback. Now, if you select all the records from the Books table, you will not see the new record with id 20, inserted by the first query inside the transaction.

Foreign Key

- Create table

```
CREATE TABLE vendor_groups (
    group_id INT IDENTITY PRIMARY KEY,
    group_name VARCHAR (100) NOT NULL
);
```

- Create table

```
CREATE TABLE vendors (
    vendor_id INT IDENTITY PRIMARY KEY,
    vendor_name VARCHAR(100) NOT NULL,
    group_id INT NOT NULL,
);
```

- Note: Consider the following vendor_groups and vendors tables:
- Each vendor belongs to a vendor group and each vendor group may have zero or more vendors. The relationship between the vendor_groups and vendors tables is one-to-many.
- For each row in the vendors table, you can always find a corresponding row in the vendor_groups table
- However, with the current tables setup, you can insert a row into the vendors table without a corresponding row in the vendor_groups table. Similarly, you can also delete a row in the vendor_groups table without updating or deleting the corresponding rows in the vendors table that results in orphaned rows in the vendors table.

Foreign Key – add

- To enforce the link between data in the vendor_groups and vendors tables, you need to establish a foreign key in the vendors table.
- A foreign key is a column or a group of columns in one table that uniquely identifies a row of another table (or the same table in case of self-reference).
- To create a foreign key, you use the FOREIGN KEY constraint.

- Remove table

```
DROP TABLE vendors;
```

- Create table with foreign key

```
CREATE TABLE vendors (
    vendor_id INT IDENTITY PRIMARY KEY,
    vendor_name VARCHAR(100) NOT NULL,
    group_id INT NOT NULL,
    CONSTRAINT fk_group FOREIGN KEY (group_id)
    REFERENCES vendor_groups(group_id)
);
```

Foreign Key - add

- The vendor_groups table now is called the **parent table** that is the table to which the foreign key constraint references. The vendors table is called the **child table** that is the table to which the foreign key constraint is applied.
- In the statement above, the following clause creates a FOREIGN KEY constraint named fk_group that links the group_id in the vendors table to the group_id in the vendor_groups table:

```
CONSTRAINT fk_group FOREIGN KEY (group_id) REFERENCES vendor_groups (group_id)
```

Foreign Key - add

- First, insert some rows into the vendor_groups table:

```
INSERT INTO vendor_groups(group_name)
VALUES('Third-Party Vendors'),
      ('Interco Vendors'),
      ('One-time Vendors');
```

- Second, insert a new vendor with a vendor group into the vendors table:

```
INSERT INTO vendors(vendor_name, group_id)
VALUES('ABC Corp',1);
```

- Third, try to insert a new vendor whose vendor group does not exist in the vendor_groups table:

```
INSERT INTO vendors(vendor_name, group_id)
VALUES('XYZ Corp',4);
```

Note: In this example, because of the FOREIGN KEY constraint, SQL Server rejected the insert and issued an error.