CHAPTER 2

SQL

2.1 Creating schema and tables

Structured Query Language (SQL) transforms a relational schema into a machine comprehensible format so that we can perform operations on an instance of the schema automatically. It is a declarative language. So, we declare what to do rather than how to do it. It is different from procedural languages, such as Java, Python and C.

We can create a schema using the create command, e.g.

```
CREATE SCHEMA Company;
```

Every statement in SQL ends with a semicolon. After creating a schema, we create tables using a similar command.

The attributes in a table have different domains. These include:

- numeric data types, such as:
  - integer numbers `INT`
  - floating point numbers `REAL` or `DECIMAL(m, n)` (`n` digits after the decimal and `m` digits before the decimal).
- character/string data types, such as:
  - fixed length of characters with length `n` `CHAR(n)`
  - variable length of characters with maximum length `n` `VARCHAR(n)`
- bit string data types, such as:
  - fixed length bit strings `BIT(n)`
  - variable length bit strings: `BIT VARYING(n)`
- boolean data type, with values `TRUE`, `FALSE` or `NULL`.
- date data type, e.g. in the format `YYYY-MM-DD`.

The following command will create 3 tables:

```
CREATE TABLE Employee (
    FName           VARCHAR(15)         NOT NULL,
    SSN             CHAR(9)             NOT NULL,
    BDate           DATE,
    Address         VARCHAR(30),
    SUPER_SSN       CHAR(9),
    DNO             INT,
    PRIMARY KEY(SSN)
);
CREATE TABLE Department(
    DName           VARCHAR(15)         NOT NULL,
    DNumber         INT                 NOT NULL,
    Mgr_SSN         CHAR(9),
    Mgr_Start_Date  DATE,
    PRIMARY KEY(DNumber),
    UNIQUE(DName),
    FOREIGN KEY(Mgr_SSN) REFERENCES Employee(SSN)
);
CREATE TABLE Dept_Locations(
    DNumber         INT                 NOT NULL,
    DLocation       VARCHAR(15)         NOT NULL,
    PRIMARY KEY(DNumber, DLocation),
    FOREIGN KEY(DNumber) REFERENCES Department(DNumber)
);
```

Within the create command, each row specifies the attribute (e.g. `SSN`), the domain (e.g. `CHAR(9)`), and any constraints we have (e.g. `NOT NULL`). At the end, we specify any primary and foreign keys.

We can set a default value for an attribute. We can also allow a value to not be `null`. For example,

```
DNO INT NOT NULL DEFAULT 1;
```

We can have a range constraint (called a check clause), e.g.

```
DNumber INT NOT NULL CHECK(DNumber > 0 AND DNumber < 21);
```

We must declare the key constraint within the create command. That is, the primary key value should be listed in the command, and this value must be unique for each tuple. Moreover, the entity integrity constraint requires that the primary key value must not be `NULL`. When we have multiple candidate keys, we use the `UNIQUE` clause. For referential constraints, we need to associate them with the primary key in the corresponding relation, e.g.

```
FOREIGN KEY (Super_SSN) REFERENCES Employee(SSN);
FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN);
```

The database system ensures consistency when we use the keywords `PRIMARY KEY` and `FOREIGN KEY`.

When a tuple is deleted and updated, other tuples that refer to this tuple can either:

- get the value `NULL` (command: `ON UPDATE SET NULL`),
- get a default value (command: `ON UPDATE SET <value>`), or
- get updated themselves (command: `ON UPDATE CASCADE`).

We can use the command explicitly to choose what happens when a tuple gets updated/deleted. A database management system must ensure that deletion/update does not make the relational schema inconsistent- it transforms from one consistent state to another.

Now, consider the following SQL create command.

```
CREATE TABLE Employee (
    ...
    DNO         INT         NOT NULL        DEFAULT 1,
    CONSTRAINT EMPPK PRIMARY KEY (SSN),
    CONSTRAINT EMPSUPERFK
        FOREIGN KEY(DNO) REFERENCES Employee(SSN) ON DELETE
        SET NULL ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
        FOREIGN KEY(DNO) REFERENCES Department(DNumber) ON DELETE
        SET DEFAULT ON UPDATE CASCADE
);
CREATE TABLE Department (
    ...
    Mgr_SSN     CHAR(9)     NOT NULL        DEFAULT "888665555",
    CONSTRAINT DEPTPK PRIMARY KEY(DNumber),
    CONSTRAINT DEPTSK UNIQUE(DName),
    CONSTRAINT DEPTMGRFK
        FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN) ON DELETE
        SET DEFAULT ON UPDATE CASCADE
);
CREATE TABLE Dept_Locations(
    ...
    PRIMARY KEY(DNumber, DLocation),
    FOREIGN KEY(DNumber) REFERENCES DEPARTMENT(DNumber) ON DELETE
    CASCADE ON UPDATE CASCADE
);
```

We have labelled each constraint, e.g. `EMPSUPERKEY`. This is optional, but helps identify them. Moreover, we have set foreign key constraints. For example, when we delete a department tuple, the employee gets assigned to the default department with `DNO = 1`, and the department location gets deleted.

2.2 Select, from, where

We will use select-from-where clause to find tuples that match a criterion. The syntax is:

```sql
SELECT  <attributes>
FROM    <tables>
WHERE   <condition>;
```

So, we first declare what to retrieve. Then, we say from which tables to retrieve this data. Finally, we state the condition that a tuple must satisfy in order for it to be retrieved. This condition must evaluate to either `TRUE`, `FALSE` or `NULL` (also called `UNKNOWN`). However, we are not stating how to implement this, e.g. how do we search and check if the tuple satisfies this condition.

We will now look at some queries. First, consider the following relational schema.

```
Employee(Name, SSN, Address, Salary, Super_SSN, DNO)
Department(DName, DNumber, Mgr_SSN, Mgr_Start_Date)
Product(PName, PNumber, PLocation, DNum)
```

We will look at some commands to retrieve relevant tuples from the relational schema.

- To select the address of employees in the department 4 or their salary is less than 31 000, we have the command:

  ```sql
  SELECT  Address
  FROM    Employee
  WHERE   DNO = 4 OR salary < 31000;
  ```

- To select the name and address of all employees who work in the Research department, we have the command:

  ```sql
  SELECT  Name, Address
  FROM    Employee, Department
  WHERE   DName = "Research" AND DNO = DNumber;
  ```

  The condition `DName = "Research"` is called a selection condition, while the condition `DNO = DNumber` is a join condition.

- To select the project number, the controlling department number and the department manager's name for a project located in "Stafford, TX", we have the command:

  ```sql
  SELECT  PNumber, DNumber, Name
  FROM    Employee, Department, Project
  WHERE   PLocation = "Stafford, TX" AND Mgr_SSN = SSN AND
              DNO = DNumber AND DNum = DNumber;
  ```

- To select the name of an employee and their supervisor, we have the following command:

  ```sql
  SELECT  E.Name, S.Name
  FROM    Employee AS E, EMPLOYEE AS S
  WHERE   E.Super_SSN = S.SSN;
  ```

  We are using variables `E` (for employee) and `S` (for supervisor) because the relation here is recursive.

- To select everything about the employees working in department number 5, we have the following command:

  ```
  SELECT   *
  FROM     Employee
  WHERE    DNO = 5;
  ```

- To select the distinct salary of each employee, we have the following command:

  ```
  SELECT   DISTINCT Salary
  FROM     Employee;
  ```

  The outcome of an SQL query is another relation. It is either a set (with distinct elements), or a multiset (with duplicates). To make it a set, we use the `DISTINCT` command.

- To select all the product numbers for those that involve employees whose name is "John", either as workers or managers of departments controlling these products, we have the following command:

  ```
  (SELECT DISTINCT PNumber
  FROM     Employee, Department, Product
  WHERE    Name = "John" AND Mgr_SSN = SSN AND DNum = DNumber)
  UNION
  (SELECT DISTINCT PNumber
  FROM     Employee, Department, Product
  WHERE    Name = "John" AND DNO = DNumber AND DNum = DNumber);
  ```

  Here, we have broken the command into 2 commands- one where the employee manages that department, and the other where the employee works for a department that controls the products. Note that the operations `UNION`, `EXCEPT` and `INTERSECT` are only defined in sets.

If we do not include the where clause with one table, we get the original table. Instead, if we have multiple tables, then this returns the Cartesian product of the two tables. This will most likely have fictitious tuples. We should make use of the foreign key constraint to return all the possible valid combinations.

2.3 Three-valued logic

SQL is a three-valued logic. So, there are 3 boolean values: `TRUE` (1), `FALSE` (0) and `UNKNOWN` (0.5). Note that every `NULL` value is different to other `NULL` values. In particular, any value compared to `NULL` evaluates to `UNKNOWN`, e.g.

- `Address = NULL` evaluates to `UNKNOWN`,
- `Address <> NULL` evaluates to `UNKNOWN`,
- `NULL = NULL` evaluates to `UNKNOWN`.

We should instead check using the commands `IS NULL` and `IS NOT NULL`. In a query, a tuple is retrieved if and only if the condition evaluates to `TRUE`. If it evaluates to `UNKNOWN` or `FALSE`, the tuple doesn't get retrieved.

The following is the logic table for the `AND` operation.

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

It can be thought of as the minimum operation, e.g. `TRUE AND UNKNOWN` is `UNKNOWN` since $\min(1, 0.5) = 0.5$. The following is the logic table for the `OR` operation.

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

It can be thought of as the maximum operation. The following is the logic table for the `NOT` operation.

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

It can be thought of as $1 -$ `VALUE`.

We can use the `null` condition to retrieve the names of all employees who do not have supervisors.

```sql
SELECT Name
FROM Employee
WHERE Super_SSN IS NULL;
```

Note that the following does not work.

```sql
SELECT Name
FROM Employee
WHERE Super_SSN = NULL;
```

As we discussed above, under this command, every tuple will evaluate to `UNKNOWN`, meaning that we do not retrieve any tuple.

2.4 Nested queries

We can nest queries within queries, i.e. a select, from, where block within the where clause. The reason we can do this is that a query returns a relation. The nested query's output is the input to the outer query's where clause. This involves using one of the three operations: `IN`, `ALL` and `EXISTS`. So, the outer query always depends on the inner query.

In a nested uncorrelated query, the inner query does not depend on the outer query. In that case, we first execute the nested query and then execute the outer query, using the inner query's output. So, it is executed once for a given command.

In a nested correlated query, the inner query also depends on the outer query. Here, we execute the nested query for each tuple of the outer query individually and then execute the outer query. So, it is executed once for each tuple of the outer query.

The operator `IN` checks whether a value belongs to a set (or a mutiset). For example, to select the SSN of those employees working in the product numbers 1, 2 or 3, we have the command:

```
SELECT SSN
FROM Employee, Department, Product
WHERE DNO = DNumber AND DNUm = DNumber AND SSN IN (1, 2, 3);
```

Now, we look at a nested version of the `IN` operator. To select the names of the employees working in the department "Research", we have the command:

```
SELECT Name
FROM Employee
WHERE DNO IN (
    SELECT DNumber
    FROM Department
    WHERE DName = "Research"
);
```

This is a nested uncorrelated query since we do not make use of the actual employee in the inner query.

The operator `ALL` compares a value with all the values from the inner query's output set using one of the following: `>`, `>=`, `<`, `<=`, `=`, `<>`. For example, to select the name of those employees whose salary is greater than the salary of all the employees in Department 5, we have the command:

```
SELECT Name
FROM Employee
WHERE Salary > ALL (
    SELECT Salary
    FROM Employee
    WHERE DNO = 5
);
```

This too is a nested uncorrelated query.

Now, to select the name of each employee who has a dependent with the same first name as the employee, we have the command:

```
SELECT E.Name
FROM Employee AS E
WHERE E.SSN IN (
    SELECT D.ESSN
    FROM Dependent AS D
    WHERE E.Name = D.Dependent_Name
);
```

This is a nested correlated query. There is a global scope (outer query), and a local scope (inner query). We have to execute the inner query for each employee. In the inner query, we compute the SSN of the employee whose dependent shares the name with the original employee. In the outer query, we check whether the original employee is part of this set. We can write this as an unnested query as well.

```
SELECT Name
FROM Employee AS E, Dependent AS D
WHERE E.SSN = D.ESSN AND E.Name = D.Dependent_Name;
```

In fact, every correlated query using `IN` can be collapsed into a single block of unnested query.

The operator `EXISTS` checks whether the inner query's output is an empty set or not. For example, to select all the employees that are working at some department, we have the command:

```
SELECT E.Name
From EMPLOYEE AS E
WHERE NOT EXISTS (
    SELECT *
    FROM Department AS D
    WHERE E.DNO = D.DNumber
);
```

Now, to select the name of all employees that work in the department Research (as a nested correlated query), we have the following command.

```
SELECT E.Name
FROM Employee AS E
WHERE EXISTS (
    SELECT *
    FROM Department AS D
    WHERE E.DNO = D.DNumber AND D.DName = "Research"
);
```

Next, to select the name of all the employees that manage a department and have dependents, we have the following command.

```
SELECT E.Name
FROM Employee AS E
WHERE EXISTS (
    SELECT *
    FROM Dependent AS P
    WHERE E.SSN = P.ESSN
) AND EXISTS (
    SELECT *
    FROM Department AS D
    WHERE E.SSN = D.MGR_SSN
);
```

Now, consider the following relational schema:

```
Student(Name, StudentID, Class)
Course(Name, CourseID, Credits, School)
Grades(StudentID, CourseID, Grade)
```

We want to retrieve the names of all students who have a grade of "A" in all of their courses. Then, the command would be:

```
SELECT Name
FROM Student
WHERE StudentID NOT IN (
    SELECT S.StudentID
    FROM Student AS S, Grade AS G
    WHERE G.Grade <> "A" AND S.StudentID = G.StudentID
);
```

This is a nested uncorrelated query. Here, the inner query finds all the students for whom there exists a grade that is not A, and we reject a student if they are present in this set. We can also compute it in the following way.

```
SELECT S.Name
FROM Student AS S
WHERE NOT EXISTS (
    SELECT *
    FROM Grade AS G
    WHERE G.Grade <> "A" AND S.StudentID = G.StudentID
);
```

This is a nested correlated query. Here, the inner query finds all the courses for the student where they got a grade that is not A, and we reject a student if this set is non-empty.

2.5 Inner and Outer Joins

When we join two records, we add to the where clause the condition where the PK value of one record matches the FK value of another one. This is called inner join with the equijoin condition, where it is of the form `R1.PK = R2.FK`.

For example, consider the following schema.

```
Employee(Name, SSN, Address, Salary, Super_SSN, DNO)
Department(DName, DNumber, Mgr_SSN, Mgr_Start_Date)
Product(PName, PNumber, PLocation, DNum)
```

If we want to show the employees who are working in the department "Research", we have the following query.

```sql
SELECT Name, Address
FROM (Employee JOIN Department ON Dno = DNumber)
WHERE DName = "Research";
```

This is equivalent to the query we had seen before.

```sql
SELECT Name, Address
FROM Employee, Department
WHERE DName = "Research" AND Dno = DNumber;
```

Here, `DName = "Research"` is the selection condition, and `DNo = DNumber` is the equi-join condition.

The value of the FK can also be `null`, and we know that any condition involving `null` gets the value `UNKNOWN`. So, if we use inner join, a tuple is retrieved if and only if there exists a matching tuple, i.e. the FK is not `null`. When we have `null` values in the FK, we need to use outer join.

In left outer join, a tuple on the left relation must appear in the result. If there is no matching tuple on the right relation, we just add `null` values for the attributes. We can similarly define right outer join. For example, if we want to find the name of an employee and the name of their supervisor (if there exists), we have the following query:

```sql
SELECT E.Name, S.Name
FROM (Employee AS E LEFT OUTER JOIN Employee AS S
    ON E.Super_SSN = S.SSN);
```

The following is the outcome of the query, using the dataset above.

`E.Name`	`S.Name`
John	Franklin
Franklin	James
Ramesh	Franklin
Joyce	Franklin
Ahmad	Jennifer
James	NULL

By using the left outer join, we guarantee that the every employee appears on the first column. However, it is possible that the FK value is `NULL`, in which case `S.Name` becomes `NULL`. Now, assume we execute the following query.

```sql
SELECT E.Name, S.Name
FROM Employee AS E, Employee AS S
WHERE E.Super_SSN = S.SSN;
```

In this case, we would not get the last cell from the table above.

2.6 Aggregation Functions

An aggregation function is a statistical summary or a value over a group of tuples. The following are built-in aggregation functions over an attribute X:

- `COUNT(*)`, which counts the number of tuples present in the query, e.g. the number of employees are working in department 5.
- `SUM(X)`, which adds the values of the attributes from the tuples present in the query, e.g. the sum of all salaries of employees in department 5.
- `MAX(X)` or `MIN(X)`, which returns a single tuple with the minimum/maximum value in the tuple, e.g. the youngest employee in department 5.
- `AVG(X)`, which computes the average of the values of the attributes from the tuples present in the query, e.g. the average salary of employees working in department 5.
- `CORR(X, Y)`, which computes the correlation coefficient between two attributes, e.g. the correlation between the age and salary of employees working in department 5.

Null values are discarded in aggregate functions, except for `COUNT(*)`. It is also possible to define a new aggregate function.

We can use aggregate functions to show the average salary of those employees working in department 5.

```
SELECT AVG(Salary) AS Average_Sal
FROM Employee
WHERE DNO = 5;
```

Group by

We can partition a relation into groups based on grouping attribute. So, we can cluster tuples when they have the same value in the grouping attribute. For example, we can group the employees working on the same department and count it:

```
SELECT DNO, Count(*)
FROM Employees
GROUP BY DNO;
```

The result will contain precisely one tuple for each department number. We must apply an aggregation function to reduce each group to a single tuple.

We will now show the number of employees per department and the average salary for each department.

```
SELECT DNO, COUNT(*), AVG(Salary)
FROM Employee
GROUP BY DNO;
```

We can also show the number of employees per age.

```
SELECT E.AGE, COUNT(*)
FROM Employee AS E
GROUP BY E.Age;
```

We can similarly compute the average salary of the employees with respect to their age.

```
SELECT E.AGE, COUNT(*), AVG(SALARY)
FROM Employee AS E
GROUP BY E.Age;
```

Similarly, we can find out how many employees are working in each project.

```
SELECT P.PName, COUNT(*)
From Project AS P, Works_on AS W
WHERE P.PNumber = W.PNO
GROUP BY P.PName;
```

Having

The having condition selects a group, i.e. it applies to aggregate functions. For example, assume we have the following relational schema:

Project(<u>PNo</u>, PName)
Works_on(<u>ESSN</u>, PNo).

We can show the number of employees per project, only from those projects with more than 2 employees.

```
SELECT P.PName, COUNT(*)
FROM Project AS P, Works_on AS W
WHERE P.PNo = W.PNo
GROUP BY P.PName
HAVING COUNT(*) > 2;
```

Next, consider the following relational schema:

Employee(Name, <u>SSN</u>, Address, Salary, Super_SSN, DNO)
Dependent(<u>ESSN</u>, <u>Dependent_Name</u>, BDate, Relationship).

We will find the employees (SSN and Name) with more than two dependents and list the number of dependents in the query below.

```
SELECT E.SSN, E.Name, COUNT(*)
FROM Employee AS E, Dependent AS D
WHERE E.SSN = D.ESSN
GROUP BY E.SSN
HAVING COUNT(*) > 2;
```

Now, consider the following relational schema:

Employee(Name, <u>SSN</u>, Address, Salary, Super_SSN, DNO)
Department(DName, <u>DNumber</u>, Mgr_SSN)

We will find the name of the managers of those departments with more than 100 employees.

```
SELECT M.Name
FROM Employee AS M, Department AS D
WHERE M.SSN = D.Mgr_SSN AND D.DNumber IN (
    SELECT E.DNO
    FROM Employee AS E
    GROUP BY E.DNO
    HAVING COUNT(*) > 100
);
```

We first identify the departments with more than 100 employees, and then select managers that manage one of these departments. It is a nested uncorrelated query.

Now, assume that for each department with more than 5 employees, we want to find how many of them make more than £40 000. A naive attempt at this might be the following query.

```
SELECT DNO, COUNT(*)
FROM Employee
WHERE Salary > 40000
GROUP BY DNO
HAVING COUNT(*) > 5;
```

This does not give us the right result. It computes the departments which have more than 5 employees that earn £40 000. We need to apply the condition that the employee earns more than £40 000 after we have found the departments with more than 5 employees. The correct query is the following.

```
SELECT DNO, COUNT(*)
FROM Employee
WHERE Salary > 40000 AND DNo IN (
    SELECT E.DNO
    FROM Employee AS E
    GROUP BY E.DNO
    HAVING COUNT(*) > 5
)
GROUP BY DNO;
```

Next, we want to find the department(s) with the maximum number of employees. It is possible that more than one department has the maximum number of employees. The query is the following. First, we have the following intermediate query.

```
SELECT E.DNO, COUNT(*) AS Members
FROM Employee AS E
GROUP BY E.DNo;
```

At this point, we have the department number, along with a count of the employees working in that department, denoted by the attribute members. Call this record `A`. Using this record, we can find the relevant departments using the following query.

```
SELECT DNO, Members
FROM A
WHERE Members = (
    SELECT MAX(Members)
    FROM A
);
```

We can group the two steps together as follows.

```
SELECT DNO, COUNT(*)
FROM Employee
GROUP BY DNO
HAVING COUNT(*) = (
    SELECT MAX(A.Members)
    FROM (
        SELECT E.DNO, COUNT(*) AS Members
        FROM Employee AS E
        GROUP BY E.DNO
    ) AS A
);
```