Data Collection and Modeling

Course 10 - Normal Forms; DML

Database Normalization

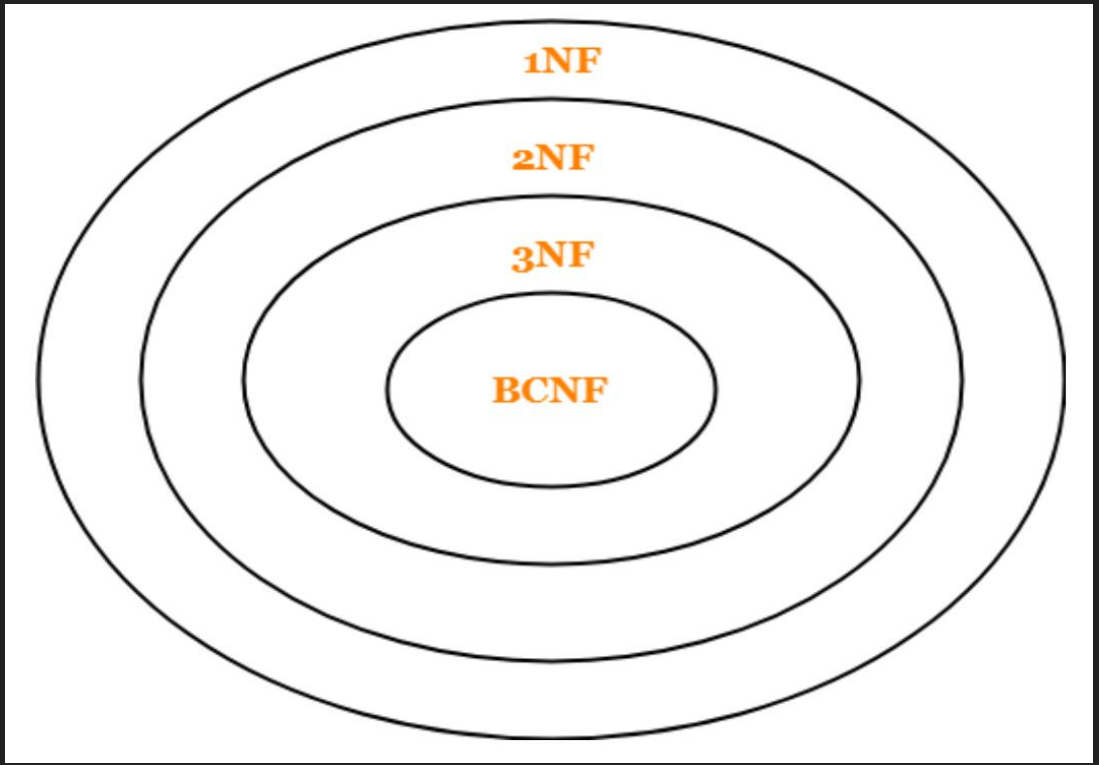
Database Normalization is a method for data organization in the database. It is based on decomposing tables to eliminate data redundancy and anomalies (insert, update and deletion).

It consists of several steps that organize data into tabular form and remove duplicate data from the tables according to rules designed to protect the data and to make the database more flexible by eliminating redundancy and inconsistent dependency. Each rule is called a normal form.

Anomalies

- Insert: inability to add data to the database due to the absence of other data. For example, if we have the following relation: `Students[id, name, section]` and section has a `NOT NULL` restriction, if the student is not given a section, can’t be saved in the database.
- Update: results from data redundancy and a partial update. If a section is recorded erroneously for several students, if it is updated only for a portion of them, there will be inconsistency.
- Delete: unintended loss of data due to deletion of other data. If a section is deleted, all students associated with it will be deleted (will no longer exist in database).

Normal Forms



Source: codingninjas

1st Normal Form

The following rules have to be satisfied:

1. There are only Single Valued Attributes.
2. Attribute Domain does not change.
3. There is a unique name for every Attribute/Column.
4. The order in which data is stored does not matter.

```
Teachers[id, name, courses]
```

2nd Normal Form

The following rules have to be satisfied:

1. It should be in the First Normal form.
2. it should not have Partial Dependency.

`Grades[student_id, course_id, grade, teacher]`

Partial Dependency is when an attribute in a table depends on a part of the primary key and not on the whole key.

3rd Normal Form

The following rules have to be satisfied:

1. It is in the Second Normal form.
2. It doesn't have Transitive Dependency.

`Students[id, name, city, country, age ]`

Transitive Dependency is when a non-prime attribute depends on other non-prime attributes instead of depending on the prime attributes or primary key.

BCNF

Boyce and Codd Normal Form is a stronger version of 3NF.

A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

1. R must be in 3rd Normal Form
2. For each functional dependency (X → Y), X should be a super Key.

`Classes [student_id, course, teacher]`

A

DML - Select

`SELECT` statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

```
SELECT column1, column2, ...
```

```
FROM table_name;
```

`column1, column2, ...` are the field names (attributes) of the table data is selected from. When selecting all the fields available in the table, the simplified syntax is:

```
SELECT * FROM table_name;
```

DML - Select

```
SELECT [DISTINCT] select_list FROM source
[ WHERE condition(s) ]
[ GROUP BY expression ]
[ HAVING condition ]
[ ORDER BY expression ] ;
```

- `select_list` specifies the fields (or column names) to retrieve.
- `DISTINCT` is used to discard duplicate records and retrieve only the unique records for the specified columns.
- `FROM` clause is the only required clause in the SELECT statement. It specifies the tables to retrieve data from.
- `WHERE` clause filters the rows such that the result contains records that meet some particular conditions.
- `GROUP BY` clause specifies the column list used to aggregate rows.
- `HAVING` clause specifies the specific conditions to group by.
- `ORDER BY` clause specifies a column list to order by (ascending or descending).

Select (PostgreSQL)

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
        [ * | expression [ [ AS ] output_name ] [, ...] ]
        [ FROM from_item [, ...] ]
        [ WHERE condition ]
        [ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
        [ HAVING condition ]
        [ WINDOW window_name AS ( window_definition ) [, ...] ]
        [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
        [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
LAST } ] [, ...] ]
        [ LIMIT { count | ALL } ]
        [ OFFSET start [ ROW | ROWS ] ]
        [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES }
]
        [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [,
...] ] [ NOWAIT | SKIP LOCKED ] [...] ]
```

Select DISTINCT

```
Students[id, name, group, country_origin, age]

-- return all groups that have students, each group once
SELECT DISTINCT group FROM Students;

-- return all countries of origin there are students from
SELECT DISTINCT country_origin FROM Students;

-- return all countries for students in each group
SELECT DISTINCT group, country_origin FROM Students;
```

Select - WHERE

```
Students[id, name, group, country_origin, age]

-- return all Romanian students
SELECT * FROM Students WHERE country_origin='Romania';

-- return all Romanian students that are younger than 20
SELECT * FROM Students WHERE country_origin='Romania' AND
age<20;

-- return all groups that have Romanian students
SELECT DISTINCT group FROM Students WHERE
country_origin='Romania' OR age>21;
```

Select - WHERE

```
Students[id, name, group, country_origin, age]

-- return all Romanian students that have age specified
SELECT * FROM Students WHERE country_origin='Romania' AND age IS NOT NULL;

-- return all Romanian students that are younger than 20 and older than --
17
SELECT * FROM Students WHERE country_origin='Romania' AND age<20 AND
age>17;

-- return all Romanian students that are younger than 20 and older than --
17
SELECT * FROM Students WHERE country_origin='Romania' AND age BETWEEN 18
AND 19;
```

Select - WHERE

```
Students[id, name, group, country_origin, age]

-- return all students that have name starting with 'AB'
SELECT * FROM Students WHERE name LIKE 'AB%';

-- return all students that have name containing 'AB'
SELECT * FROM Students WHERE name LIKE '%AB%';

-- return all students that have name starting with any
-- letter followed 'AB'
SELECT * FROM Students WHERE name LIKE '_AB%';
```

Select - ORDER BY

```
Students[id, name, group, country_origin, age]

-- return all Romanian students ordered by age
SELECT * FROM Students WHERE country_origin='Romania' ORDER BY
age;

-- return all students that are older than 20 order by group
and, -- within each group by name
SELECT * FROM Students WHERE age>20 ORDER BY group ASC, age ASC;

-- return all distinct student names ordered lexicographically
in -- descending order
SELECT DISTINCT name FROM Students ORDER BY name DESC;
```

Select - IN

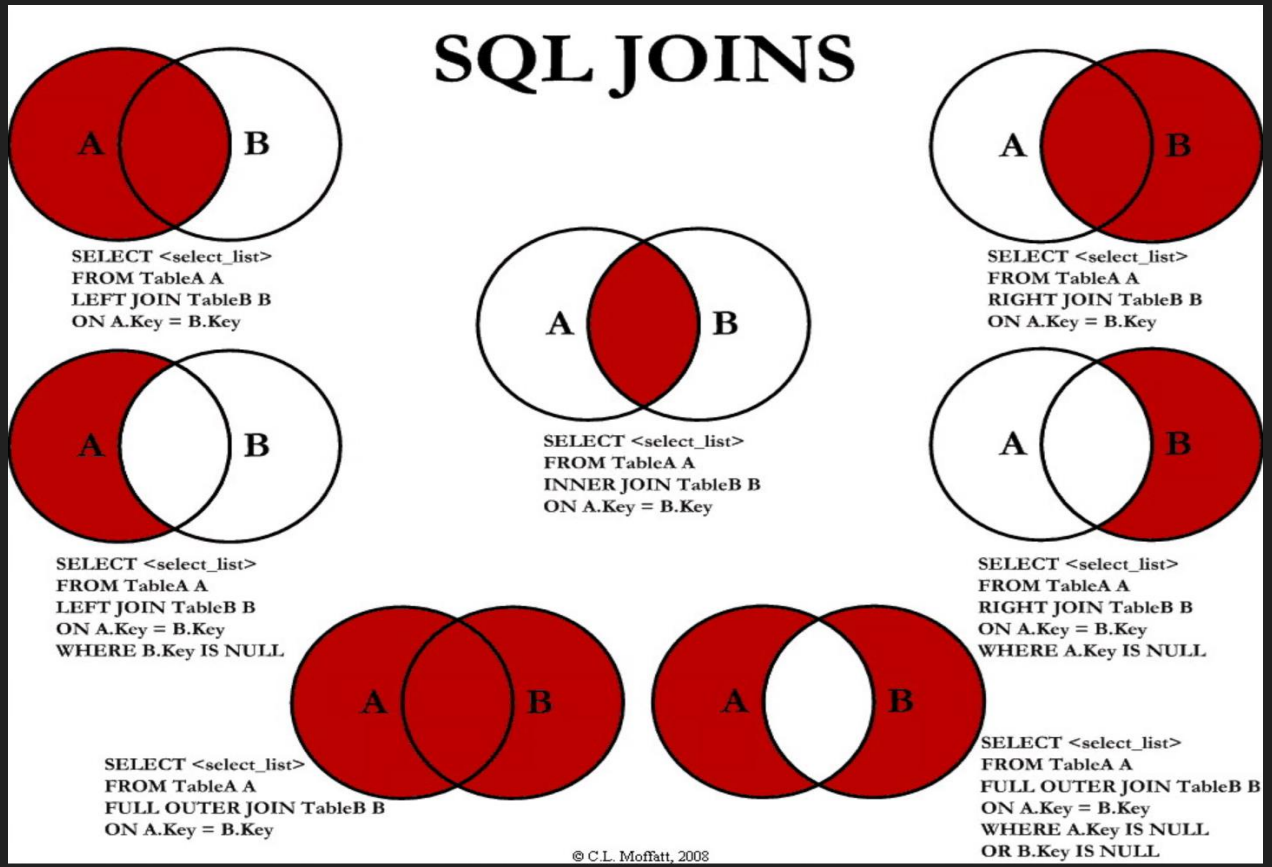
```
Students[id, name, group, country_origin, age]

-- return all Romanian and French students
SELECT * FROM Students WHERE country_origin IN ('Romania',
'France');

-- return all students that are 20 and there have the same
-- name with students being 18 years old
SELECT * FROM Students WHERE age=20 AND name IN (SELECT name
FROM Students WHERE age=18);
```

SQL - JOINS

`JOIN` is a command clause that combines records from two or more tables in a database based on related columns/column expressions between them.

- INNER JOIN: records that have matching values for join condition in both tables
- LEFT [OUTER] JOIN: all records from the left table and matching values from the right table
- RIGHT [OUTER] JOIN: all records from the right table and matching values from the left table
- [FULL | OUTER] JOIN: all records from the right table and left table, once for the matching values

SQL - JOINS



source : codeproject.com

INNER JOIN

```
Students[id, name, group_id, country_origin, age]
Groups[id, code, description]

-- return student names, their age and group codes for
-- Romanian students
SELECT Students.name, Students.age, Groups.code FROM Students, Groups
WHERE Students.group_id=Groups.id AND Student.country_origin='Romania';

SELECT s.name, s.age, g.code FROM Students s, Groups g where
s.group_id=g.id AND s.country_origin='Romania';

SELECT s.name, s.age, g.code FROM Students s INNER JOIN Groups g ON
s.group_id=g.id AND s.country_origin='Romania';

SELECT s.name, s.age, g.code FROM Students s INNER JOIN Groups g ON
s.group_id=g.id WHERE s.country_origin='Romania';
```

LEFT JOIN

```
Students[id, name, group_id, country_origin, age]
Groups[id, code, description]

-- return student names, their age and group codes (if
-- present) for Romanian students, even if they are not
-- registered in a group
SELECT s.name, s.age, g.code FROM Students s LEFT JOIN Groups
g ON s.group_id=g.id AND s.country_origin='Romania';
```

RIGHT JOIN

```
Students[id, name, group_id, country_origin, age]
Groups[id, code, description]

-- return student names, their age and group codes (if
-- present) for Romanian students, even if they are not
-- registered in a group
SELECT s.name, s.age, g.code FROM Groups g RIGHT JOIN
Students s ON s.group_id=g.id AND s.country_origin='Romania';
```

FULL JOIN

```
Students[id, name, group_id, country_origin, age]
Groups[id, code, description]

-- return student names, their age and group codes (if
-- present) for Romanian students, even if they are not
-- registered in a group and also group codes with no
-- students registered
SELECT s.name, s.age, g.code FROM Groups g FULL JOIN Students
s ON s.group_id=g.id AND s.country_origin='Romania';
```

Self JOIN

A (regular) join where a table is joined with itself

```
Students[id, name, group, country_origin, age]

-- return all students with same name and different age
SELECT * FROM Students s1, Students s2 WHERE s1.name=s2.name
AND s1.age<>s2.age;

-- return all students with same name and different age
SELECT * FROM Students s1 INNER JOIN Students s2 ON
s1.name=s2.name AND s1.age<>s2.age;
```

SQL - UNION

Operator that combines two or more result sets into one; some conditions have to be met:

- Every result set must have the same number of columns
- Corresponding columns must have compatible data types
- Each result set has to have same column names in same order

There are two versions: `UNION` and `UNION ALL`

- Simple form of the operator selects distinct rows
- The version with `ALL` selects all rows

SQL - UNION

```
Students[id, name, group, country_origin, age]
Teachers[id, name, department, age]

-- return persons
SELECT name, age FROM Students
UNION
SELECT name, age FROM Teachers;

-- return all persons
SELECT name, age FROM Students
UNION ALL
SELECT name, age FROM Teachers;
```

SQL - GROUP BY

A clause that is used to generate summary rows based on grouping rows using identical values one or several columns; the result will contain one row per group (set of distinct values). The restriction is that any column in the SELECT list that is not part of an aggregate expression must be included in the GROUP BY list.

Usual aggregate functions used:

- COUNT()
- MIN(), MAX()
- SUM(), AVG()

SQL - GROUP BY

```
Students[id, name, group, country_origin, age]

-- return total number of students
SELECT count(id) FROM Students;
SELECT count(*) as total_no FROM Students;

-- return number of students from each group
SELECT group, count(id) FROM Students GROUP BY group;

-- return number of students from each group, by country of
origin
SELECT group, country_origin, count(id) FROM Students GROUP
BY group, country_origin;
```

SQL - GROUP BY … HAVING

Is a clause that is similar to where but applied on the grouping result; WHERE clause can be used only before GROUP BY

```
Students[id, name, group, country_origin, age]

-- return number of non Romanian students from each group
SELECT group, count(id) FROM Students WHERE
country_origin<>'Romania' GROUP BY group;

-- return number of non Romanian students from each group, by
country of origin
SELECT group, country_origin, count(id) FROM Students GROUP
BY group, country_origin HAVING country_origin<>'Romania';
```

Self JOIN GROUP BY - Exercise

Students[id, name, group, final_grade]

Using Self JOIN and GROUP BY, determine the ranking for each group of students based on their final_grade; the ranking should start with highest final grade, and, when the same grade, the order should be alphabetical.

Self JOIN GROUP BY - Exercise

`Students[id, name, group, final_grade]`

Using `Self JOIN` and `GROUP BY`, determine the ranking for each group of students based on their `final_grade`; the ranking should start with highest final grade, and, when the same grade, the order should be alphabetical.

Self JOIN GROUP BY - Exercise

```
Students[id, name, group, final_grade]

-- obtaining ranking of grades by group
SELECT s1.group, s1.final_grade, count(*)
FROM Students s1 INNER JOIN Students s2
on s1.group=s2.group and s1.final_grade <= s2.final_grade
group by s1.group, s1.final_grade
order by 1,3
-- one more time Students table is needed for final result
```

Self JOIN GROUP BY - Exercise

```
Students[id, name, group, final_grade]

-- obtaining final ranking
SELECT s4.group, s4.name, s4.final_grade, s3.pos FROM
(SELECT s1.group, s1.final_grade, count(*) as pos
FROM Students s1 INNER JOIN Students s2
on s1.group=s2.group and s1.final_grade <= s2.final_grade
group by s1.group, s1.final_grade
) s3
INNER JOIN Students s4 on s3.group=s4.group and
s3.final_grade=s4.final_grade
order by 1,4,2
```