Databases and SQL

1. Introduction

1.1 What is SQL?

`SQL` stands for `Structured Query Language`. It is a language used to "talk" to a database server. It can be used as frontend to many databases (mysql, oracle, MSSQL, ...).

The SQL language covers three subsystems: - data description - data access - privileges

1.2 When do you need a database?

- Simultaneous access and changes to data (= concurrency)
- share huge data among many people
- web interfaces to data (dynamic data)
- data needs to be structured

1.3 Keys (Primary keys and Foreign Keys)

1.3.1 Primary Key

In order for a table to qualify as a relational table it must have a primary key.

The primary key consists of one or more columns whose data contained within is used to uniquely identify each row in the table.

When a primary key is composed of multiple columns, the data from each column is used to determine whether a row is unique.

In order to be a primary key, several conditions must hold true:

- The columns must be unique
- No value in the columns can be blank or NULL.
- The primary key for each table is stored in an index. The index is used to enforce the uniqueness requirement.

1.3.2 Foreign Key

A foreign key is a set of one or more columns in a table that refers to the primary key in another table.

Unlike primary keys, foreign keys can contain duplicate values. Also, it is OK for them contain NULL values.

Indexes aren't automatically created for foreign keys; however, you can define (and probably should!) them.

A table is allowed to contain more than one foreign key.

Foreign Key Constraints

Some database management systems allow you to set up foreign key constraints. These help to enforce referential integrity. In their simplest form, a foreign key constraint stops you from entering values that aren't found in the related table's primary key.

2. Database structure - Normalization

2.1 Definition

Normalization is a process used to organize a database into tables and columns. The idea is that a table should be about a specific topic and that only those columns which support that topic are included.

For example, a spreadsheet containing information about sales people and customers serves several purposes:

- Identify sales people in your organization
- List all customers your company calls upon to sell product
- Identify which sales people call on specific customers

By limiting a table to one purpose you reduce the number of duplicate data that is contained within your database, which helps eliminate some issues.

2.2 Reasons for normalization

There are three main reasons to normalize a database:

1. Minimize duplicate data
2. Minimize or avoid data modification issues
3. Simplify queries

Here is an example of a table that isn't normalized:

SalesStaff						
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

What are the problems here?

- **Insert**: We can't insert a new sales office without knowing who is the sales person.
- **Update**: If the phone number of the sales office in Chicago changes, we have to update the data in multiple rows. If one row isn't updated, the data becomes inconsistent.
- **Delete**: If John Hunt leaves the office in New York, deleting this row will also delete the data concerning the sales office in New York
- **Search and Sort**: If you want to search for an office with a customer named `Ford`, your query will have to test the values of each `customer` row (if Customer1 == "Ford" OR Customer2 == "Ford" OR Customer3 == "Ford)

2.3 Levels of Normalization

There are three levels of Normalization (`NF` for `Normal Form`):

1. **First Normal Form**: Information is stored in a relational table and each column contains atomic values, and there is no repeating groups of columns
2. **Second Normal Form**: The table is in *first normal form* and all the columns depend on the table's primary key.
3. **Third Normal Form**: the table is in *second normal form* and all of its columns are not transitively dependent on the primary key

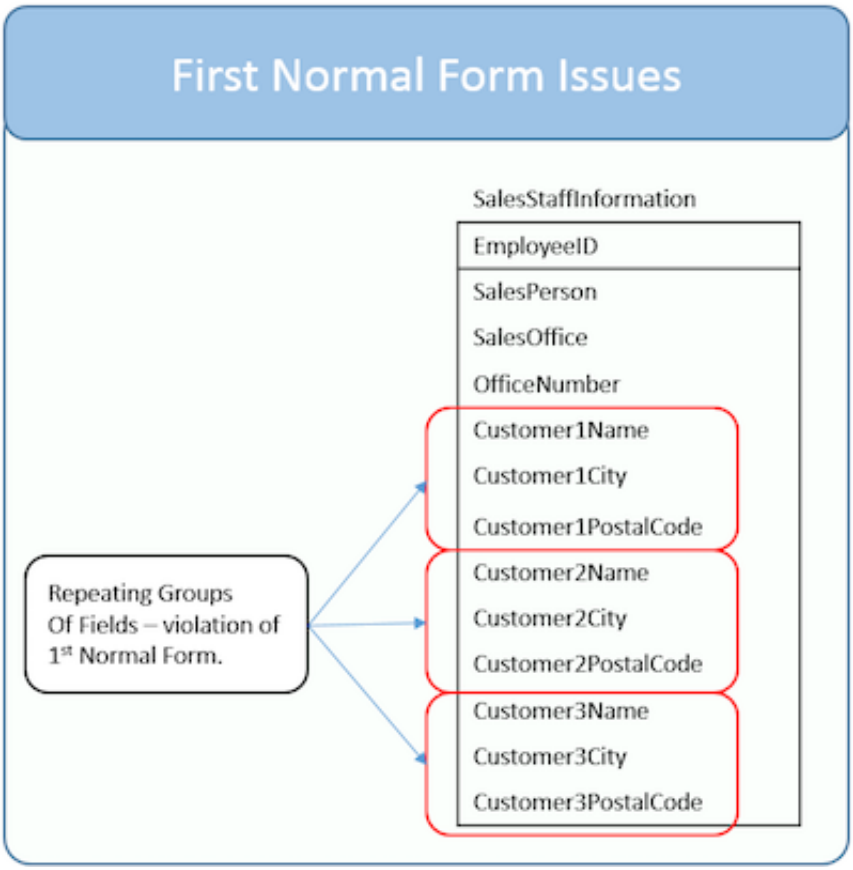
2.3.1 1NF - First Normal Form definition

- The data must be in a database table
  Ok, I think there is nothing to say here...

- Each column contains atomic values, and there are not repeating groups of columns
  `Each column contains atomic values`
  A column cannot contain multiple values, e.g.: `SalesPerson` cannot have the value `Mary Smith, John Doe`
  `there are not repeating groups of columns`
  A table should not contain repeating groups of columns such as `Customer1`, `Customer2`, and `Customer3`.

Issues with our data structure

The table should not contain repeating groups of columns such as Customer1, Customer2, and Customer3.



Solution

The repeating groups of columns now become separate rows in the Customer table linked by the EmployeeID foreign key.

SalesStaffInformation			
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**
1003	Mary Smith	Chicago	312-555-1212
1004	John Hunt	New York	212-555-1212
1005	Martin Hap	Chicago	312-555-1212

Note: Primary Key: EmployeeID

Customer				
CustomerID	**EmployeeID**	**CustomerName**	**CustomerCity**	**PostalCode**
C1000	1003	Ford	Dearborn	48123
C1010	1003	GM	Detroit	48213
C1020	1004	Dell	Austin	78720
C1030	1004	HP	Palo Alto	94303
C1040	1004	Apple	Cupertino	95014
C1050	1005	Boeing	Chicago	60601

1. The original design limited each SalesStaffInformation entry to three customers. In the new design, the number of customers associated to each design is practically unlimited.
2. It was nearly impossible to Sort the original data by Customer (not easily anway...). Now, it is simple to sort customers.
3. The same holds true for filtering on the customer table. It is much easier to filter on one customer name related column than three.
4. The insert and deletion anomalies for Customer have been eliminated. You can delete all the customer for a SalesPerson without having to delete the entire SalesStaffInformaiton row.

2.3.2 2NF - Second Normal Form definition

- The table is in 1st normal form
- All the non-key columns are dependent on the table's primary key.

A table must have a single purpose, it must be used to describe a single type of entity. Also, the primary key provides a means to uniquely identify each row in a table.

Once you identify a table's purpose, then look at each of the table's columns and ask yourself, "Does this column serve to describe what the primary key identifies?"

- `yes` : the column depends on the primary key and belongs to the table
- `no` : the column should be moved different table

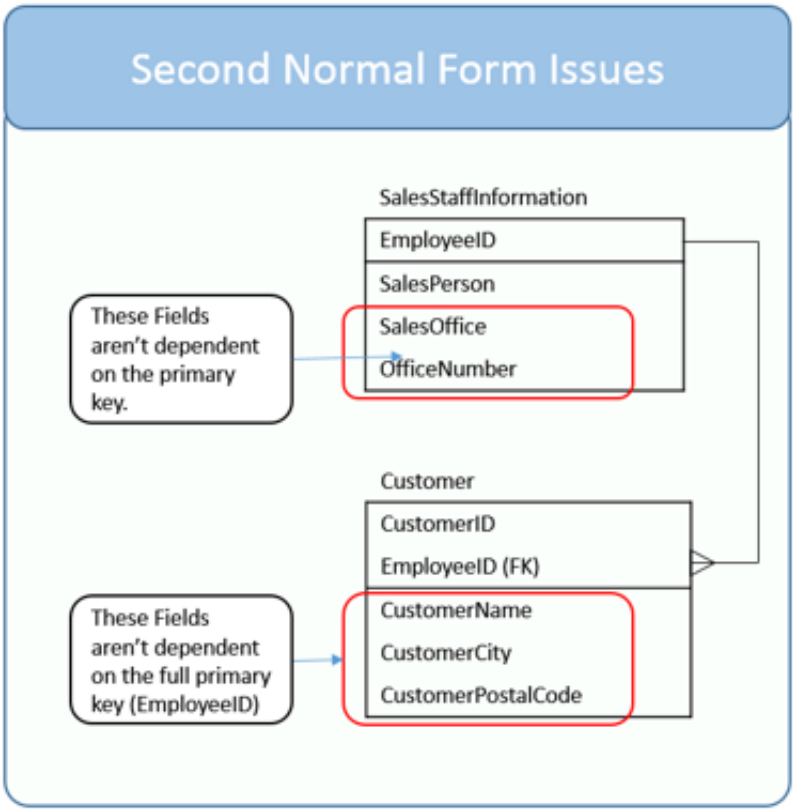
Issues with our data structure

`SalesStaffInformation` table has two columns which aren't dependent on the `EmployeeID` . Though they are used to describe which office the SalesPerson is based out of, the SalesOffice and OfficeNumber columns themselves don't serve to describe who the employee is.

The second issue is that there are several attributes which don't completely rely on the entire `Customer` table primary key. For a given customer, it doesn't make sense that you should have to know both the CustomerID and EmployeeID to find the customer.
The `Customer` table currently has two purposes:

- To indicate which customers are called upon by each employee
- To identify customers and their locations.

It stands to reason you should only need to know the CustomerID. Given this, the Customer table isn't in 2nd normal form as there are columns that aren't dependent on the full primary key. They should be moved to another table.



Solution

Customer			
CustomerID	**CustomerName**	**CustomerCity**	**CustomerPostalCode**
C1000	Ford	Dearborn	48123
C1010	GM	Detroit	48213
C1020	Dell	Austin	78720
C1030	HP	Palo Alto	94303
C1040	Apple	Cupertino	95014
C1050	Boeing	Chicago	60601

SalesStaffCustomer	
CustomerID	**EmployeeID**
C1000	1003
C1010	1003
C1020	1004
C1030	1004
C1040	1004
C1050	1005

SalesStaffInformation		
EmployeeID	**SalesPerson**	**SalesOffice**
1003	Mary Smith	S10
1004	John Hunt	S20
1005	Martin Hap	S10

SalesOffice		
SalesOfficeID	**SalesOffice**	**OfficeNumber**
S10	Chicago	312-555-1212
S20	New York	212-555-1212

2.3.3 3NF - Third Normal Form definition

- A table is in 2nd normal form.
- It contains only columns that are non-transitively dependent on the primary key

Transitive

When something is transitive, then if something applies from the beginning to the end, it also applies from the middle to the end.

Example
In general, if `A` is greater than `B`, and `B` is greater than `C`, then it follows that `A` is greater than `C`. In this case, the "greater than" (`>`) comparison is `transitive`.

Dependance

An object has a dependence on another object when it relies upon it. In the case of databases, when we say that a column has a dependence on another column, we mean that the value can be derived from the other. For example, my age is dependent on my birthday.

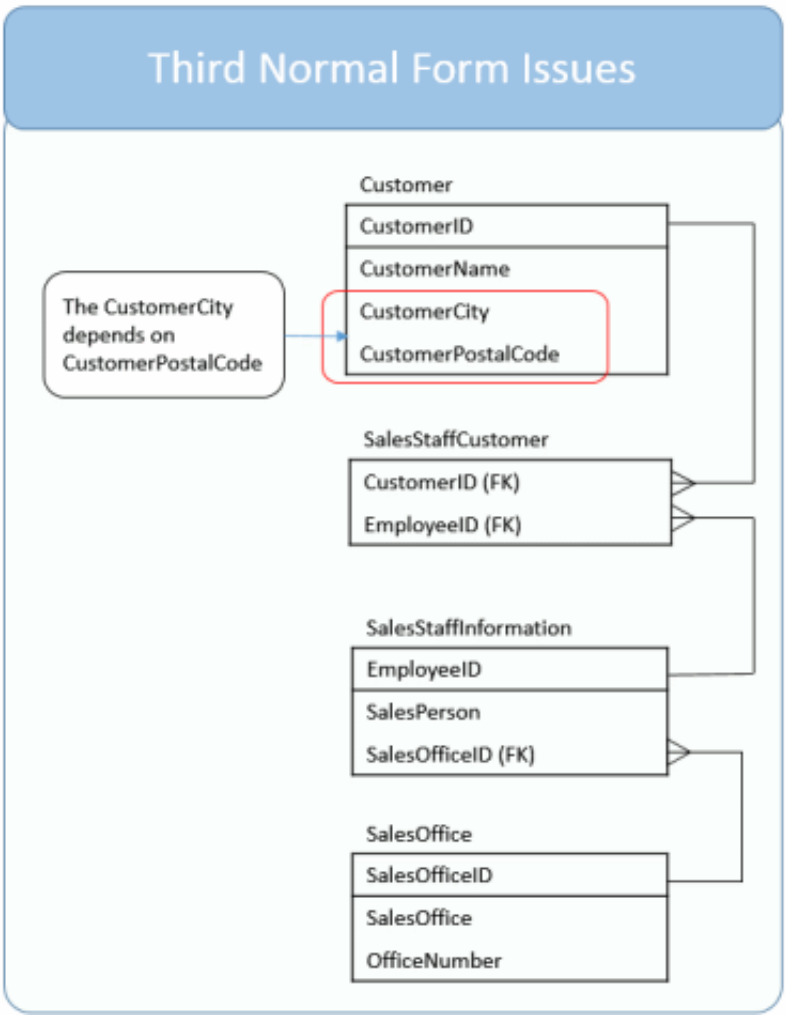
Transitive Dependence

It is simplest to think of `transitive dependence` to mean a column's value relies upon another column through a second intermediate column.

Example
Consider three columns: `AuthorNationality`, `Author`, and `Book`. Column values for `AuthorNationality` and `Author` rely on the `Book`; once the book is known, you can find out the `Author` or `AuthorNationality`. But also notice that the `AuthorNationality` relies upon `Author`. That is, once you know the `Author`, you can determine their nationality. In this sense then, the `AuthorNationality` relies upon `Book`, via `Author`.

Issues with our data structure

Generally speaking a postal code applies to one city. Although all the columns are dependent on the primary key, CustomerID, there is an opportunity for an update anomaly as you could update the CustomerPostalCode without making a corresponding update to the CustomerCity.



Solution

In order for our model to be in third normal form, we need to remove the transitive dependencies. As we stated our dependency is:
`CustomerCity relies on CustomerPostalCode which relies on CustomerID`.

Customer		
CustomerID	**CustomerName**	**CustomerPostalCode**
C1000	Ford	48123
C1010	GM	48213
C1020	Dell	78720
C1030	HP	94303
C1040	Apple	95014
C1050	Boeing	60601

PostalCode	
PostalCode	**City**
48123	Dearborn
48213	Detroit
60601	Chicago
78720	Austin
94303	Palo Alto
95014	Cupertino

2.3.4 Can Normalization Get out of Hand?

Sometimes itsn't worth the time and effort to fully normalize a database. In our example you could argue to keep the database in second normal form, that the CustomerCity to CustomerPostalCode dependency isn't a deal breaker.

There are times when you'll intentionally denormalize data. If you need to present summarized or complied data to a user, and that data is very time consuming or resource intensive to create, it may make sense to maintain this data separately.

3. SQL

3.1 Types

SQL supports a very large number of different formats for internal storage of information.

- Numeric
  - INTEGER, SMALLINT, BGINT
  - NUMBERIC (w,d) `width and decimal places`
  - REAL, DOUBLE PRECISION
  - FLOAT(p) `floating point with p binay digits of precision
- Character
  - CHARACTER(L) `fixed-length of length L`
  - ...
- Binary
  - BLOB
  - ...
- Temporal
  - DATE
  - TIME

- TIMESTAMP

3.2 CREATE

```
CREATE TABLE kids(id CHAR(6),
                  race SMALLINT,
                  age DECIMAL(6,3),
                  height DECIMAL(7,3),
                  weight DECIMAL(7,3),
                  sex SMALLINT);
```

3.3 INSERT

```
INSERT INTO kids VALUES(100011,2,10.346,
                        148.5,38.95,1);
```

```
LOAD DATA INFILE 'kids.tab'
          INTO TABLE kids
          FIELDS TERMINATED BY '\t';
```

3.4 SELECT

3.4.1 Comparison Operators used in conditions

- Usual logical operators: `<` `>` `<=` `>=` `=` `<>`
- `BETWEEN` used to test for a range
- `IN` used to test group membership
- Keyword `NOT` used for negation
- `LIKE` operator allows wildcards
  - _ means single character
  - `%` means anything
  - `SELECT salary WHERE name LIKE 'Fred %';`
- `RLIKE` operator allows regular expressions
- Use `AND` and `OR` to combine conditions

3.4.2 Summaries and Computations

- COUNT()
- AVG()
- SUM()
- MIN()

- MAX()

Other functions may also be available:

- ABS()
- FLOOR()
- ROUND()
- SQRT()
- ...

3.4.3 SELECT statement

3.4.3.1 Syntax

```
SELECT columns or computations
   FROM table
   WHERE condition
   GROUP BY columns
   HAVING condition
   ORDER BY column  [ASC | DESC]
   LIMIT offset,count;
```

3.4.3.2 Examples

Find the first 10 tallest kids heavier than 80 kg and shorter than 1.5m:

```
SELECT age,race,height,weight
FROM kids
WHERE
   weight > 80
   AND height < 150
ORDER BY height DESC
LIMIT 1,10
```

Get the tallest kid aged between 10 and 11 with the race `1` (whatever it means)

```
SELECT MAX(height)
FROM kids
WHERE age BETWEEN 10 AND 11
    AND race = 1;
```

By combining with the GROUP BY command, useful summaries can be obtained.
Find the average BMI (`body mass index` , = weight / (height * height) * 10000) by sex and race:

```
SELECT sex,race,count(*) AS n,
     AVG(weight/(height*height)*10000) AS bmi
FROM kids
GROUP BY sex, race;
```

Summaries cannot be used in `WHERE` clause, but in `HAVING` clause.
Find all the IDs in the kids database for which there were less than 2 observations:

```
SELECT id /* in this case the id field is not unique */
FROM kids
GROUP BY id
HAVING COUNT(*) < 2;
```

3.4.3.3 Subqueries

Find all information about the observations with maximum weight:

```
SELECT *
FROM kids
WHERE weight = (
    SELECT MAX(weight)
    FROM kids
);
```

`Aliases` and the use of a `subquery as a table`:

```
SELECT k.id ,k.sex, k.race, k.age, k.weight, k.height
FROM
    kids AS k,
    (SELECT sex,race,max(weight) AS weight from kids) AS m
WHERE
    k.sex=m.sex
    AND k.race=m.race
    AND k.weight=m.weight;
```

3.4.3.4 Making a table from a query

You can dynamically create and store the data based on other tables:

```
CREATE TABLE young LIKE kids;
INSERT INTO young SELECT * FROM kids
     WHERE age < 15;
```

3.5 UPDATE

```
UPDATE kids SET weight=weight + 1
            WHERE id='101311' AND
            age BETWEEN 9 and 10;
```

Be careful with `UPDATE`, because if you don't provide a `WHERE` clause, all the rows of the table will be changed!

3.6 DELETE

3.6.1 Syntax

```
DELETE
FROM   tableName
WHERE  searchCondition;
```

3.6.2 Examples

Remove all kids:

```
DELETE FROM kids;
```

Remove all kids taller than 1.50m

```
DELETE FROM kids
WHERE  height > 150;
```

Delete all the kids that have the maximum weight:

```
DELETE FROM kids
WHERE weight = (
    SELECT MAX(weight)
    FROM kids
);
```

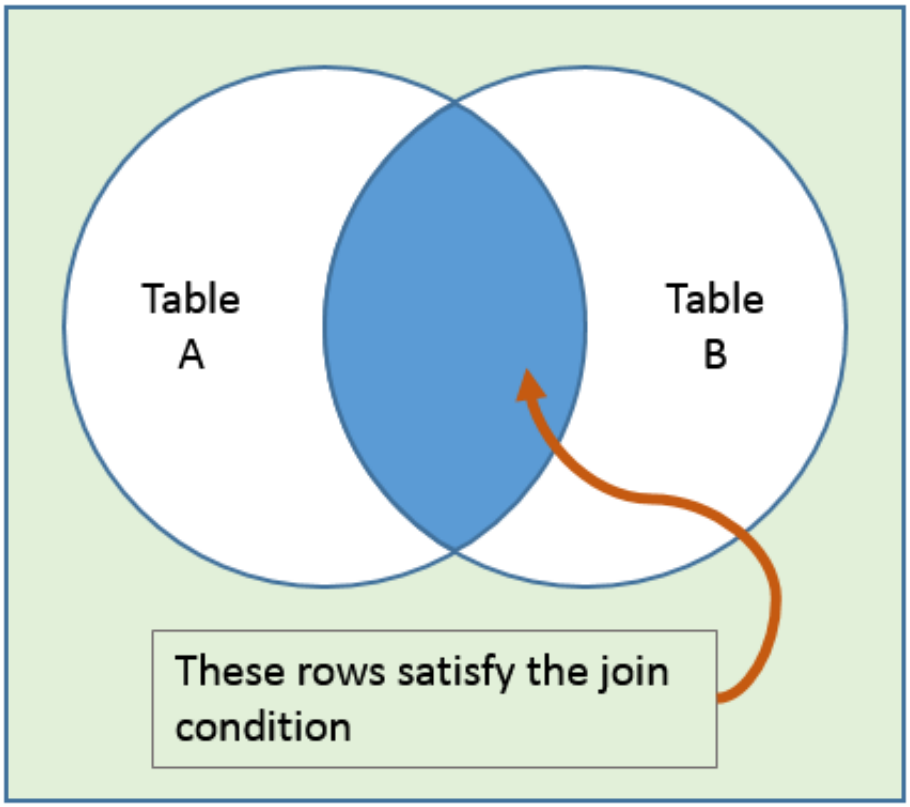
3.7 JOINS

3.7.1 What is a join?

Database joins are used to match rows between tables to allow us to stitch the database back together to make it easy to read and use. In most cases we're matching a column value from one table with another.

3.7.2 Mechanics of a Join

When broken down the mechanics of a join are pretty straightforward. To perform a join you need two items: two tables and a join condition. The tables contain the rows to be combined, and the join condition the instructions to match rows together.



3.7.3 Cross Join

Definition

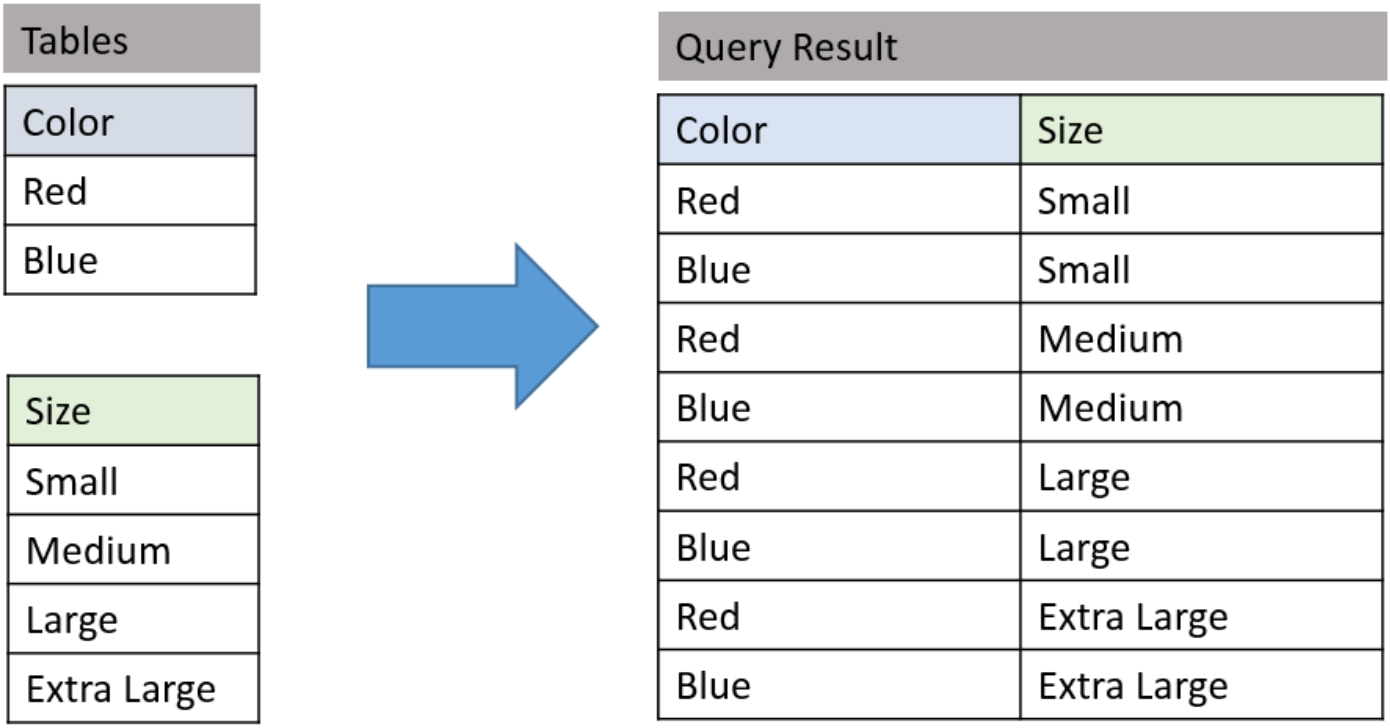
A cross join is used when you wish to create combination of every row from two tables. All row combinations are included in the result; this is commonly called cross product join.

Syntax

```
SELECT columnlist
FROM maintable
CROSS JOIN secondtable
```

Example

```
SELECT c.Color, s.Size
FROM Color c
CROSS JOIN Size s
```



3.7.4 Inner Join

Definition

An inner join is used when you need to match rows from two tables. Rows that match remain in the result, those that don't are rejected. The condition must be met on **both** sides.

Syntax

```
SELECT columnlist
FROM maintable
INNER JOIN secondtable ON join condition
```

Let's say we have these two tables (a person can have 0, 1 or multiple phone numbers):

```
persons
|--------------------------|
| id | firstname | lastname |
|--------------------------|
| 1  | Mike      | Dam      |
| 2  | Foon      | Birthday |
| 3  | Dominic   | SwissGuy | <-- this guy has no phone number
|--------------------------|
              |
              |
phones       / \
|------------------------------|
| id | person_id | number      |
|------------------------------|
| 1 | 1          | 11111111    |
| 2 | 1          | 22222222    |
| 3 | 2          | 33333333    |
| 4 | NULL       | 44444444    | <-- this phone number belongs to nobody
|------------------------------|
```

Example

We want to to retrieve a list of all the employees with their phone number.
If an employee has no phone number, we skip it. If an employee has mulitple phone numbers, one row per number in the results:

```
SELECT persons.firstname,
    persons.lastname,
    phones.number
FROM persons
INNER JOIN phones ON phones.person_id = persons.id
```

Result

```
|---------------------------------|
| firstname | lastname  | number  |
|---------------------------------|
| Mike      | Dam       | 11111111 |
| Mike      | Dam       | 22222222 |
| Foon      | Birthday | 33333333 |
|---------------------------------|
```

3.7.5 Outer Join

There are three types of outer joins:

- `Left Outer Join` – All rows from the left table are included, unmatched rows from the right are replaced with NULL values.
- `Right Outer Join` – All rows from the right table are included, unmatched rows from the left are replaced with NULL values.
- `Full Outer Join` – All rows from both tables are included, NULL values fill unmatched rows.

3.7.5.1 Left Outer Join

Syntax

```
SELECT columnlist
FROM   table
LEFT OUTER JOIN othertable ON join condition
```

Example

```
SELECT persons.id, persons.firstname, persons.lastname, phones.number
FROM persons
LEFT OUTER JOIN phones ON (phones.person_id = persons.id)
```

Result

```
|---------------------------------|
| firstname | lastname | number   |
|---------------------------------|
| Mike      | Dam      | 11111111 |
| Mike      | Dam      | 22222222 |
| Foon      | Birthday | 33333333 |
| Dominic   | SwissGuy | NULL     |
|---------------------------------|
```

3.7.5.2 Right Outer Join

Syntax

```
SELECT columnlist
FROM   table
RIGHT OUTER JOIN othertable ON join condition
```

Example

```
SELECT persons.id, persons.firstname, persons.lastname, phones.number
FROM persons
RIGHT OUTER JOIN phones ON (phones.person_id = persons.id)
```

Result

```
|---------------------------------|
| firstname | lastname | number   |
|---------------------------------|
| Mike      | Dam      | 11111111 |
| Mike      | Dam      | 22222222 |
| Foon      | Birthday | 33333333 |
| NULL      | NULL     | 44444444 |
|---------------------------------|
```

3.7.5.3 Full Outer Join

A full outer join is the combination of results from a left and right outer join. The results returned from this type of join include all rows from both tables. Where matches occur, values are related. Where matched from either table don't, then NULL are returned instead.

Syntax

```
SELECT columnlist
FROM   table
FULL OUTER JOIN othertable ON join condition
```

Example

```
SELECT persons.id, persons.firstname, persons.lastname, phones.number
FROM persons
RIGHT OUTER JOIN phones ON (phones.person_id = persons.id)
```

Example

```
SELECT persons.id, persons.firstname, persons.lastname, phones.number
FROM persons
FULL OUTER JOIN phones ON (phones.person_id = persons.id)
```

Result

```
|---------------------------------|
| firstname | lastname | number   |
|---------------------------------|
| Mike      | Dam      | 11111111 |
| Mike      | Dam      | 22222222 |
| Foon      | Birthday | 33333333 |
| Dominic   | SwissGuy | NULL     |
| NULL      | NULL     | 44444444 |
|---------------------------------|
```

3.8 Transactions

A transaction is a single unit of work. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.

It is important to note that MySQL automatically commits the changes to the database by default.
To force MySQL not to commit changes automatically, you use the following statement:

```
SET autocommit = 0;
```

3.8.1 Commit

```
START TRANSACTION;

DELETE FROM phones WHERE id = 4;
INSERT INTO phones (person_id, number) VALUES (1, '5545544');

COMMIT;
```

The SQL statements have been applied because of the `COMMIT` keyword.

3.8.2 Rollback

```
START TRANSACTION;

DELETE FROM phones WHERE id = 4;
INSERT INTO phones (person_id, number) VALUES (1, '5545544');

ROLLBACK;
```

The SQL statements have been cancelled, and therefore not applied, because of the `ROLLBACK` keyword.