

# DB 01

# Database

## 데이터베이스 (DB)

데이터베이스는 **체계화된 데이터**의 모임이다.

여러 사람이 공유하고 사용할 목적으로 통합 관리되는 정보의 집합이다.

논리적으로 연관된 (하나 이상의) 자료의 모음으로

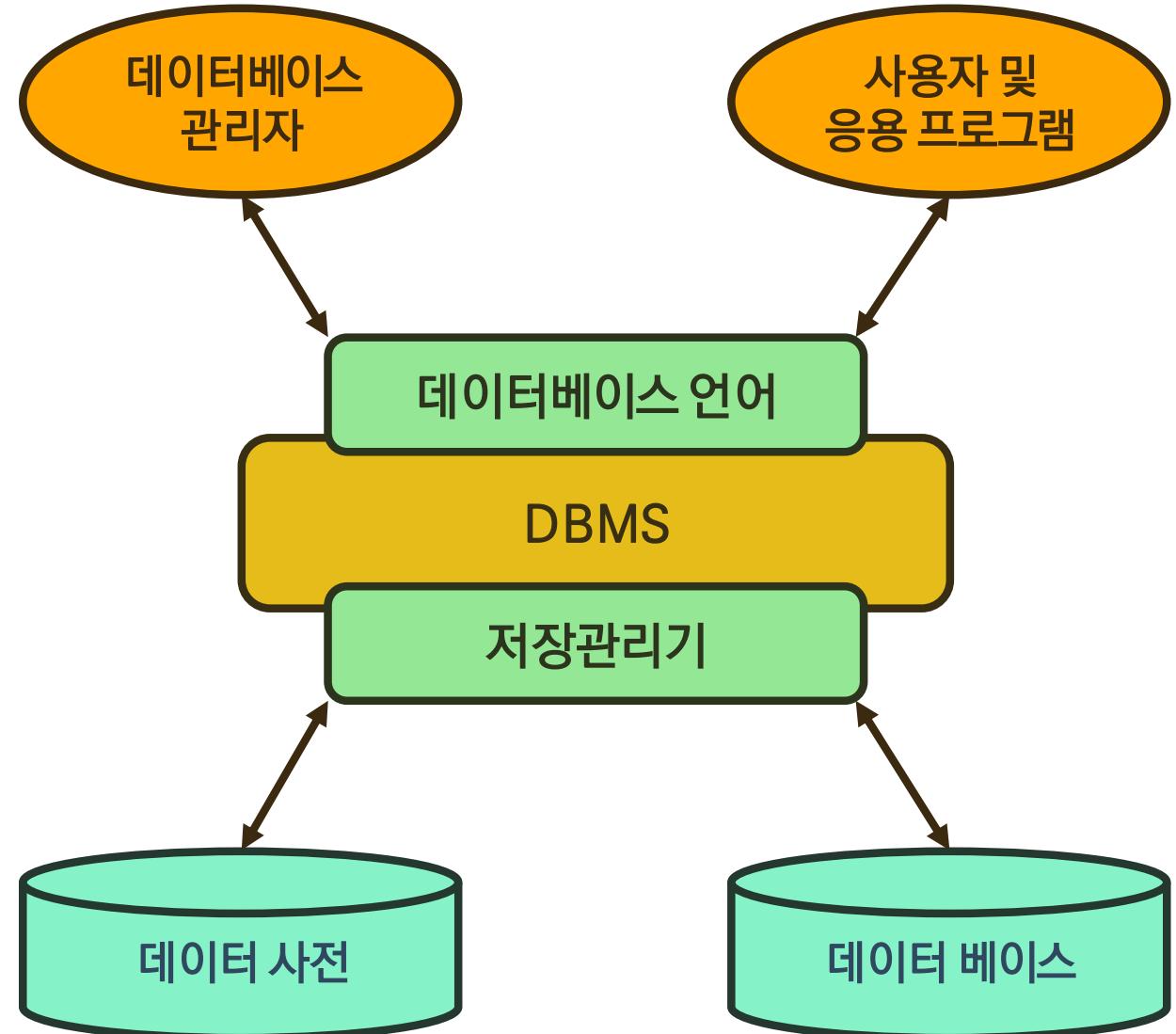
그 내용을 고도로 구조화 검색과 생산의 효율화를 꾀한 것이다.

즉, 몇 개의 자료 파일을 조직적으로 통합하여

자료 항목의 중복을 없애고 자료를 구조화하여 기억시켜 놓은 자료의 집합체

## 데이터베이스로 얻는 장점들

- 데이터 중복 최소화
- 데이터 무결성 (정확한 정보를 보장)
- 데이터 일관성
- 데이터 독립성 (물리적 / 논리적)
- 데이터 표준화
- 데이터 보안 유지



RDB

## 관계형 데이터베이스 (RDB)

- Relational Database
- 키(key)와 값(value)들의 간단한 관계(relation)를 표(table) 형태로 정리한 데이터베이스
- 관계형 모델에 기반

고유 번호	이름	주소	나이
1	홍길동	제주	20
2	김길동	서울	30
3	박길동	독도	40

## 관계형 데이터베이스 용어 정리 (1/5)

- 스키마 (schema) : 데이터베이스에서 자료의 구조, 표현방법, 관계등 전반적인 명세를 기술한 것.

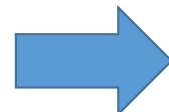
column	datatype
id	INT
name	TEXT
address	TEXT
age	INT

## 관계형 데이터베이스 용어 정리 (2/5)

- 테이블 (table) : 열(컬럼/필드)과 행(레코드/값)의 모델을 사용해 조직된 데이터 요소들의 집합

Schema

column	datatype
id	INT
name	TEXT
address	TEXT
age	INT



Table

id	name	address	age
1	홍길동	제주	20
2	김길동	서울	30
3	박길동	독도	40

## 관계형 데이터베이스 용어 정리 (3/5)

- 열 (Column) : 각 열에는 고유한 데이터 형식이 지정됨.
- 아래의 예시에서는 name이란 필드에 고객의 이름(TEXT) 정보가 저장됨.

열, 컬럼, 필드 등의 이름으로 불림

id	name	address	age
1	홍길동	제주	20
2	김길동	서울	30
3	박길동	독도	40

## 관계형 데이터베이스 용어 정리 (4/5)

- 행 (row) : 실제 데이터가 저장되는 형태
- 아래의 예시에서는 총 3명의 고객정보가 저장되어 있음(레코드가 3개)

id	name	address	age
1	홍길동	제주	20
2	김길동	서울	30
3	박길동	독도	40

행, 로우, 레코드 등의 이름으로 불림

## 관계형 데이터베이스 용어 정리 (5/5)

- 기본키 (Primary Key) : 각 행(레코드)의 고유 값
- 반드시 설정해야 하며, 데이터베이스 관리 및 관계 설정 시 주요하게 활용 됨.

PK, 기본키

id	name	address	age
1	홍길동	제주	20
2	김길동	서울	30
3	박길동	독도	40

# RDBMS

## 관계형 데이터베이스 관리 시스템 (RDBMS)

- Relational Database Management System
- 관계형 모델을 기반으로 하는 데이터베이스 관리시스템을 의미
- 예시)
  - MySQL
  - SQLite
  - PostgreSQL
  - ORACLE
  - MS SQL



## SQLite



서버 형태가 아닌 파일 형식으로 응용 프로그램에 넣어서 사용하는 **비교적 가벼운 데이터베이스**

구글 안드로이드 운영체제에 기본적으로 탑재된 데이터베이스이며, 임베디드 소프트웨어에도 많이 활용됨

로컬에서 간단한 DB 구성을 할 수 있으며, 오픈소스 프로젝트이기 때문에 자유롭게 사용 가능

## SQLite 설치하기

- “sqlite3 설치” 문서 확인
  - <https://abit.ly/ssafy-document>

# SQL

## SQL (Structured Query Language)

- 관계형 데이터베이스 관리시스템의 데이터 관리를 위해 설계된 특수 목적으로 프로그래밍 언어
- 데이터베이스 스키마 생성 및 수정
- 자료의 검색 및 관리
- 데이터베이스 객체 접근 조정 관리

## SQL 분류

분류	개념	예시
DDL – 데이터 정의 언어 (Data Definition Language)	관계형 데이터베이스 구조(테이블, 스키마)를 정의하기 위한 명령어	CREATE DROP ALTER
DML – 데이터 조작 언어 (Data Manipulation Language)	데이터를 저장, 조회, 수정, 삭제 등을 하기 위한 명령어	INSERT SELECT UPDATE DELETE
DCL – 데이터 제어 언어 (Data Control Language)	데이터베이스 사용자의 권한 제어를 위해 사용하는 명령어	GRANT REVOKE COMMIT ROLLBACK

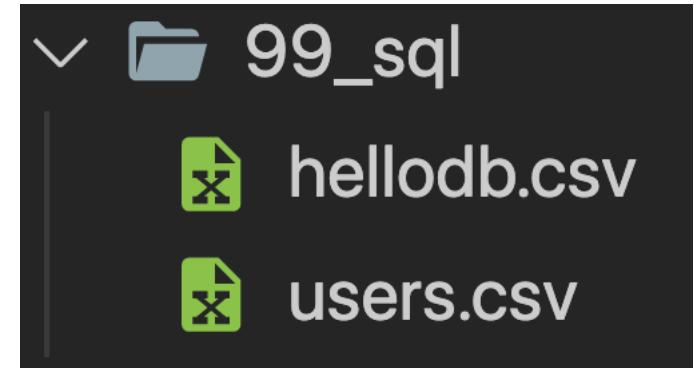
## SQL Keywords – Data Manipulation Language

- INSERT : 새로운 데이터 삽입(추가)
- SELECT : 저장되어있는 데이터 조회
- UPDATE : 저장되어있는 데이터 갱신
- DELETE : 저장되어있는 데이터 삭제

# 테이블 생성 및 삭제

## csv 파일 다운로드

- <https://abit.ly/ssafy-db> – SQL 풀더
- SQL 실습을 위한 별도의 풀더(99\_sql) 생성 및 csv 파일 다운로드



## 진행 TIP

- vscode 화면을 좌우로 나누어서 진행
  - 1. 좌측 (sql 확장자 파일)
    - 명령어 기록 및 누적 (개념정리)
    - 좌측에서 먼저 명령어 입력 후 우측으로 복붙
  - 2. 우측 (터미널, db.sqlite3)
    - 명령어 작성 및 결과 확인
- View – Appearance – Move Panel Right

The screenshot shows the Visual Studio Code (VS Code) interface with a dark theme. The left side of the screen features a sidebar with icons for file, folder, search, and other extensions. The main area is split into two panes. The left pane is a code editor containing an SQL file named '01\_intro.sql' with the following content:

```
-- 전체 조회
SELECT * FROM users_user;
```

The right pane is a terminal window titled '01\_intro.sql — sqllite'. It displays the command 'sqlite> SELECT \* FROM users\_user;' and the response 'Ln 2, Col 26'. At the bottom of the terminal window, there are status indicators for 'Spaces: 2', 'UTF-8', 'LF', and 'SQLite'.

## 데이터베이스 생성하기

```
$ sqlite3 tutorial.sqlite3  
sqlite> .database
```



‘.’은 sqlite 프로그램의 기능을 실행하는 것

## csv 파일을 table로 만들기

```
sqlite> .mode csv  
sqlite> .import hellodb.csv examples  
sqlite> .tables  
examples
```

## SELECT

```
SELECT * FROM examples;
```



; 까지 하나의 명령 (SQL Query)로 간주 됨!

## SELECT 확인하기

```
sqlite> SELECT * FROM examples;  
1, "길동", "홍", 600, "충청도", 010-2424-1232
```

SELECT 문은 특정 테이블의 레코드(행) 정보를 반환!

## (Optional) 터미널 view 변경하기

```
sqlite> SELECT * FROM examples;
1,"길동","홍",600,"충청도",010-2424-1232
sqlite> .headers on
sqlite> SELECT * FROM examples;
id,first_name,last_name,age,country,phone
1,"길동","홍",600,"충청도",010-2424-1232
sqlite> .mode column
sqlite> SELECT * FROM examples;
id      first_name  last_name   age      country    phone
-----  -----  -----  -----  -----
1          길동        홍       600    충청도  010-2424-1232
```

## 테이블 생성 및 삭제 statement

- **CREATE TABLE**
  - 데이터베이스에서 테이블 생성
- **DROP TABLE**
  - 데이터베이스에서 테이블 제거

## CREATE

```
CREATE TABLE classmates (
    id INTEGER PRIMARY KEY,
    name TEXT
);
```

## CREATE – 테이블 생성 및 확인하기

```
sqlite> CREATE TABLE classmates (
...> id INTEGER PRIMARY KEY,
...> name TEXT
...> );
sqlite> .tables
classmates examples
```

CREATE는 테이블을 생성!

## 특정 테이블의 schema 조회

방금 전 생성한  
classmates  
테이블의 스키마

```
sqlite> .schema classmates
CREATE TABLE classmates (
    id INTEGER PRIMARY KEY,
    name TEXT
);
```

## DROP

```
DROP TABLE classmates;
```

## DROP

```
sqlite> DROP TABLE classmates;  
sqlite> .tables  
examples
```

DROP은 테이블을 삭제!

## 테이블 생성 실습 (1/2)

- 다음과 같은 스키마(schema)를 가지고 있는 classmates 테이블을 만들고 스키마를 확인해보세요.

column	datatype
name	TEXT
age	INT
address	TEXT

## 테이블 생성 실습 (2/2)

SQL

```
CREATE TABLE classmates (
    name TEXT,
    age INT,
    address TEXT
);
```

터미널 창 확인

```
sqlite> .schema classmates
CREATE TABLE classmates (
    name TEXT,
    age INT,
    address TEXT
);
```

# CRUD

**CREATE**

## CREATE

- **INSERT**

- inserting a single row into a table
- 테이블에 단일 행 삽입

## INSERT

```
INSERT INTO 테이블이름 (컬럼1, 컬럼2, ...) VALUES (값1, 값2, ...);
```

**INSERT는 특정 테이블에 레코드(행)를 삽입(생성)!**

## ■ INSERT 사용해보기 (1/5)

```
INSERT INTO 테이블이름 (컬럼1, 컬럼2, ...) VALUES (값1, 값2, ...);
```

## INSERT 사용해보기 (2/5)

```
INSERT INTO classmates (name, age) VALUES ('홍길동', 23);
```

Q. `classmates` 테이블에 이름이 홍길동이고 나이가 23인 데이터를 넣어봅시다.

그리고 `SELECT`문을 통해 확인해보세요.

```
sqlite> SELECT * FROM classmates;
name          age          address
-----        -----
홍길동          23
```

## ■ INSERT 사용해보기 (3/5)

Q. classmates 테이블에 이름이 홍길동이고, 나이가 300이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 SELECT문을 통해 확인해보세요.

## INSERT 사용해보기 (4/5)

Q. classmates 테이블에 이름이 홍길동이고, 나이가 30이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 SELECT문을 통해 확인해보세요.

```
INSERT INTO classmates VALUES ('홍길동', 30, '서울');
```

모든 열에 데이터가 있는 경우 column을 명시하지 않아도 됨!

## INSERT 사용해보기 (5/5)

Q. classmates 테이블에 이름이 홍길동이고, 나이가 30이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 SELECT문을 통해 확인해보세요.

```
sqlite> SELECT * FROM classmates;  
name          age          address  
-----        -----  
홍길동        23  
홍길동        30          서울
```

## 2가지 의문점

```
sqlite> SELECT * FROM classmates;
```

name	age	address
-----	-----	-----
홍길동	23	
홍길동	30	서울

정상동작은 하지만 2가지 의문점 발생

## 1. id는 어디로 갔을까?

```
sqlite> SELECT * FROM classmates;
```

name	age	address
-----	-----	-----
홍길동	23	
홍길동	30	서울

## 사실 SQLite가 관리해주고 있었다 !

rowid 를 포함해서 조회

```
sqlite> SELECT rowid, * FROM classmates;
```

rowid	name	age	address
1	홍길동	23	
2	홍길동	30	서울

SQLite는 따로 PRIMARY KEY 속성의 컬럼을 작성하지 않으면

값이 자동으로 증가하는 PK 옵션을 가진 **rowid** 컬럼을 정의

## 2. 비어있는 1호 홍길동씨의 주소

```
sqlite> SELECT rowid, * FROM classmates;  
rowid      name       age      address  
-----  -----  -----  -----  
1          홍길동     23        
2          홍길동     30      서울
```

1호 홍길동씨의 주소가 비어있는 상황

이대로 괜찮을까?

NULL

```
sqlite> SELECT rowid, * FROM classmates;  
rowid      name       age      address  
-----  -----  -----  -----  
1          홍길동     23        
2          홍길동     30      서울
```

NO!

주소가 꼭 필요한 정보라면 공백으로 비워두면 안된다. (NOT NULL 설정 필요)

## 지우고 새로 만들기

```
sqlite> DROP TABLE classmates;  
sqlite> CREATE TABLE classmates (  
...> id INTEGER PRIMARY KEY,  
...> name TEXT NOT NULL,  
...> age INT NOT NULL,  
...> address TEXT NOT NULL  
...> );
```

## 다시 INSERT 사용해보기

Q. classmates 테이블에 이름이 홍길동이고, 나이가 30이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 SELECT문을 통해 확인해보세요.

```
INSERT INTO classmates VALUES ('홍길동', 30, '서울');
```

## 실패

Q. classmates 테이블에 이름이 홍길동이고, 나이가 30이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 SELECT문을 통해 확인해보세요.

```
sqlite> INSERT INTO classmates VALUES ('홍길동', 30, '서울');  
Error: table classmates has 4 columns but 3 values were supplied
```

스키마에 id를 직접 작성했기 때문에

입력할 column들을 명시하지 않으면 자동으로 입력되지 않음

## 첫번째 방법

Q. classmates 테이블에 이름이 홍길동이고, 나이가 30이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 SELECT문을 통해 확인해보세요.

```
INSERT INTO classmates VALUES (1, '홍길동', 30, '서울');
```

1. id를 포함한 모든 value를 작성

## 두번째 방법

Q. classmates 테이블에 이름이 홍길동이고, 나이가 30이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 SELECT문을 통해 확인해보세요.

```
INSERT INTO classmates (name, age, address) VALUES ('홍길동', 30, '서울');
```

2. 각 value에 맞는 column들을 명시적으로 작성

## 성공

Q. `classmates` 테이블에 이름이 홍길동이고, 나이가 30이고, 주소가 서울인 데이터를 넣어봅시다.

그리고 `SELECT`문을 통해 확인해보세요.

```
sqlite> INSERT INTO classmates VALUES (1, '홍길동', 30, '서울');
sqlite> INSERT INTO classmates (name, age, address) VALUES ('홍길동', 30, '서울');
sqlite> SELECT * FROM classmates;
id      name       age      address
-----  -----      -----  -----
1       홍길동     30      서울
2       홍길동     30      서울
```

이번 실습에서는 `rowid`를 사용해서 편하게 공부하자

## 지우고 새로 만들기

```
sqlite> DROP TABLE classmates;  
sqlite> CREATE TABLE classmates (  
...>     name TEXT NOT NULL,  
...>     age INT NOT NULL,  
...>     address TEXT NOT NULL  
...> );
```

이번 실습에서는 **rowid**를 사용해서 편하게 진행하자

## ■ INSERT 직접 해보기

- 방금 새롭게 생성한 테이블에 다음과 같은 정보를 저장하고 확인해봅시다.
- 각 정보는 이름 / 나이 / 주소로 구분됩니다.
  - 홍길동 / 30 / 서울
  - 김철수 / 30 / 대전
  - 이싸피 / 26 / 광주
  - 박삼성 / 29 / 구미
  - 최전자 / 28 / 부산

## INSERT 직접 해보기 – 해설

SQL

```
INSERT INTO classmates VALUES
('홍길동', 30, '서울'),
('김철수', 30, '대전'),
('이싸피', 26, '광주'),
('박삼성', 29, '구미'),
('최전자', 28, '부산');
```

터미널 창 확인

```
sqlite> SELECT * FROM classmates;
name      age      address
-----  -----  -----
홍길동      30      서울
김철수      30      대전
이싸피      26      광주
박삼성      29      구미
최전자      28      부산
```

**READ**

## SELECT statement

- **SELECT**
  - to query data from a table
  - 테이블에서 데이터를 조회
  - SELECT 문은 SQLite에서 가장 복잡한 문이며 다양한 절(clause)와 함께 사용
    - ORDER BY, DISTINCT, WHERE, LIMIT, GROUP BY ...

## SELECT와 함께 사용하는 clause (1/2)

- **LIMIT**

- to constrain the number of rows returned by a query.
- 쿼리에서 반환되는 행 수를 제한
- 특정 행부터 시작해서 조회하기 위해 **OFFSET** 키워드와 함께 사용하기도 함

- **Where**

- to specify the search condition for rows returned by the query.
- 쿼리에서 반환된 행에 대한 특정 검색 조건을 지정

## SELECT와 함께 사용하는 clause (2/2)

- **SELECT DISTINCT**

- to remove duplicate rows in the result set.
- 조회 결과에서 중복 행을 제거
- DISTINCT 절은 SELECT 키워드 바로 뒤에 작성해야 함

## SELECT statement

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름;
```

모든 컬럼 값이 아닌 특정 컬럼만 조회하기

## SELECT statement

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름;
```

Q. `classmates` 테이블에서 `id`, `name` 컬럼 값만 조회하세요.

## SELECT statement

```
SELECT rowid, name FROM classmates;
```

Q. classmates 테이블에서 id, name 컬럼 값만 조회하세요.

```
sqlite> SELECT rowid, name FROM classmates;  
rowid      name  
-----  -----  
1          홍길동  
2          김철수  
3          이싸피  
4          박삼성  
5          최전자
```

## LIMIT

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름 LIMIT 숫자;
```

원하는 수 만큼 데이터 조회하기

## LIMIT

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름 LIMIT 숫자;
```

Q. `classmates` 테이블에서 `id`, `name` 컬럼 값을 하나만 조회하세요.

## LIMIT

```
SELECT rowid, name FROM classmates LIMIT 1;
```

Q. classmates 테이블에서 id, name 컬럼 값을 하나만 조회하세요.

```
sqlite> SELECT rowid, name FROM classmates LIMIT 1;
rowid      name
-----  -----
1          홍길동
```

## OFFSET keyword

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름 LIMIT 숫자 OFFSET 숫자;
```

특정 부분에서 원하는 수 만큼 데이터 조회하기

## OFFSET keyword

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름 LIMIT 숫자 OFFSET 숫자;
```

Q. `classmates` 테이블에서 `id, name` 컬럼 값을 세번째에 있는 하나만 조회하세요.

## OFFSET keyword

```
SELECT rowid, name FROM classmates LIMIT 1 OFFSET 2;
```

Q. classmates 테이블에서 id, name 컬럼 값을 세번째에 있는 하나만 조회하세요.

```
sqlite> SELECT rowid, name FROM classmates LIMIT 1 OFFSET 2;
rowid      name
-----  -----
3          이싸피
```

## WHERE

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름 WHERE 조건;
```

특정 데이터(조건) 조회하기

## WHERE

```
SELECT 컬럼1, 컬럼2, ... FROM 테이블이름 WHERE 조건;
```

Q. `classmates` 테이블에서 `id`, `name` 컬럼 값 중에 주소가 서울인 경우의 데이터를 조회하세요.

## WHERE

```
SELECT rowid, name FROM classmates WHERE address='서울';
```

Q. classmates 테이블에서 id, name 컬럼 값 중에 주소가 서울인 경우의 데이터를 조회하세요.

```
sqlite> SELECT rowid, name FROM classmates WHERE address='서울';
rowid      name
-----
1          홍길동
```

## DISTINCT

```
SELECT DISTINCT 컬럼 FROM 테이블이름;
```

특정 컬럼을 기준으로 중복없이 가져오기

## DISTINCT

```
SELECT DISTINCT 컬럼 FROM 테이블이름;
```

Q. **classmates** 테이블에서 age값 전체를 중복없이 조회하세요.

## DISTINCT

```
SELECT DISTINCT age FROM classmates;
```

Q. classmates 테이블에서 age값 전체를 중복없이 조회하세요.

```
sqlite> SELECT DISTINCT age FROM classmates;  
age  
-----  
30  
26  
29  
28
```

**DELETE**

## ■ DELETE statement

- **DELETE**
  - to remove rows from a table.
  - 테이블에서 행을 제거

## DELETE

```
DELETE FROM 테이블이름 WHERE 조건;
```

조건을 통해 특정 레코드 삭제하기

## 삭제하기 전에 (1/2)

```
DELETE FROM 테이블이름 WHERE 조건;
```

그럼 어떤 기준으로 데이터를 삭제하면 좋을까?

## 삭제하기 전에 (2/2)

```
DELETE FROM 테이블이름 WHERE 조건;
```

그럼 어떤 기준으로 데이터를 삭제하면 좋을까?

**중복 불가능한(UNIQUE) 값인 rowid를 기준으로 삭제하자!**

## DELETE

```
DELETE FROM 테이블이름 WHERE 조건;
```

Q. classmates 테이블에 id가 5인 레코드를 삭제하세요.

## DELETE

```
DELETE FROM classmates WHERE rowid=5;
```

Q. classmates 테이블에 id가 5인 레코드를 삭제하세요.

## | DELETE

```
DELETE FROM classmates WHERE rowid=5;
```

Q. classmates 테이블에 id가 5인 레코드를 삭제하세요.

```
sqlite> DELETE FROM classmates WHERE rowid=5;
sqlite> SELECT rowid, * FROM classmates;
      rowid      name       age      address
-----  -----
        1    홍길동      30      서울
        2    김철수      30      대전
        3    이싸피      26      광주
        4    박삼성      29      구미
```

## 삭제하고 나서 (1/3)

데이터 삭제는 잘 되었는데, 지워진 id(5)는 어떻게 되는걸까?

데이터를 다시 추가해서 확인해보자

```
INSERT INTO classmates VALUES ('최전자', 28, '부산');
```

## 삭제하고 나서 (2/3)

```
sqlite> DELETE FROM classmates WHERE rowid=5;  
sqlite> SELECT rowid, * FROM classmates;  
rowid      name       age      address  
-----  
1          홍길동     30      서울  
2          김철수     30      대전  
3          이싸피     26      광주  
4          박삼성     29      구미  
sqlite> INSERT INTO classmates VALUES ('최전자', 28, '부산');  
sqlite> SELECT rowid, * FROM classmates;  
rowid      name       age      address  
-----  
1          홍길동     30      서울  
2          김철수     30      대전  
3          이싸피     26      광주  
4          박삼성     29      구미  
5          최전자     28      부산
```

**SQLite는 기본적으로 id를 재사용**

## 삭제하고 나서 (3/3)

```
sqlite> DELETE FROM classmates WHERE rowid=5;  
sqlite> SELECT rowid, * FROM classmates;  
rowid      name       age      address  
-----  
1          홍길동     30      서울  
2          김철수     30      대전  
3          이싸피     26      광주  
4          박삼성     29      구미  
sqlite> INSERT INTO classmates VALUES ('최전자', 28, '부산');  
sqlite> SELECT rowid, * FROM classmates;  
rowid      name       age      address  
-----  
1          홍길동     30      서울  
2          김철수     30      대전  
3          이싸피     26      광주  
4          박삼성     29      구미  
5          최전자     28      부산
```

재사용 없이 사용하지 않은 다음 행 값을  
사용하게 하려면 어떻게 해야할까?

## AUTOINCREMENT (1/2)

- Column attribute
- SQLite가 사용되지 않은 값이나 이전에 삭제된 행의 값을 재사용하는 것을 방지

## AUTOINCREMENT (2/2)

```
CREATE TABLE 테이블이름 (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    ...
);
```

테이블을 생성하는 단계에서

AUTOINCREMENT를 통해 설정 가능

(이번 실습에서는 사용하지 않지만, django에서 기본 값으로 사용되는 개념!)

**UPDATE**

## ■ UPDATE statement

- **UPDATE**
  - to update data of existing rows in the table.
  - 기존 행의 데이터를 수정
  - **SET** clause에서 테이블의 각 열에 대해 새로운 값을 설정

## UPDATE

```
UPDATE 테이블이름 SET 컬럼1=값1, 컬럼2=값2, ... WHERE 조건;
```

조건을 통해 특정 레코드 수정하기

중복 불가능한(UNIQUE) 값인 rowid를 기준으로 수정하자!

## UPDATE

```
UPDATE 테이블이름 SET 컬럼1=값1, 컬럼2=값2, ... WHERE 조건;
```

Q. `classmates` 테이블에 id가 5인 레코드를 수정하세요.

이름을 홍길동으로, 주소를 제주도로 바꿔주세요!

# UPDATE

```
UPDATE classmates SET name='홍길동', address='제주도' WHERE rowid=5;
```

Q. `classmates` 테이블에 id가 5인 레코드를 수정하세요.

이름을 홍길동으로, 주소를 제주도로 바꿔주세요!

```
sqlite> UPDATE classmates SET name='홍길동', address='제주도' WHERE rowid=5;
sqlite> SELECT * FROM classmates;
name      age       address
-----  -----  -----
홍길동      30        서울
김철수      30        대전
이싸피      26        광주
박삼성      29        구미
홍길동      28        제주도
```

## CRUD 정리하기

	구문	예시
C	INSERT	<b>INSERT INTO</b> 테이블이름 (컬럼1, 컬럼2, ...) <b>VALUES</b> (값1, 값2);
R	SELECT	<b>SELECT *</b> <b>FROM</b> 테이블이름 <b>WHERE</b> 조건;
U	UPDATE	<b>UPDATE</b> 테이블이름 <b>SET</b> 컬럼1=값1, 컬럼2=값2 <b>WHERE</b> 조건;
D	DELETE	<b>DELETE FROM</b> 테이블이름 <b>WHERE</b> 조건;

WHERE

## Table users 생성

```
CREATE TABLE users (
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    age INTEGER NOT NULL,
    country TEXT NOT NULL,
    phone TEXT NOT NULL,
    balance INTEGER NOT NULL
);
```

## csv파일 정보를 테이블에 적용하기

```
sqlite> .mode csv  
sqlite> .import users.csv users  
sqlite> .tables  
classmates examples users
```

## WHERE 복습

```
SELECT * FROM 테이블이름 WHERE 조건;
```

특정 조건으로 데이터 조회하기

## ■ WHERE 활용

Q. users 테이블에서 age가 30 이상인 유저의 모든 컬럼 정보를 조회하려면?

## WHERE 활용

Q. users 테이블에서 age가 30 이상인 유저의 모든 컬럼 정보를 조회하려면?

```
SELECT * FROM users WHERE age >= 30;
```

## WHERE 활용

Q. users 테이블에서 age가 30 이상인 유저의 이름만 조회하려면?

## WHERE 활용

Q. users 테이블에서 age가 30 이상인 유저의 이름만 조회하려면?

```
SELECT first_name FROM users WHERE age>=30;
```

## WHERE 활용

Q. users 테이블에서 age가 30 이상이고 성이 ‘김’인 사람의  
나이와 성만 조회하려면?

## WHERE 활용

Q. users 테이블에서 age가 30 이상이고 성이 ‘김’인 사람의  
나이와 성만 조회하려면?

```
SELECT age, last_name FROM users WHERE age>=30 AND last_name='김';
```

# Sqlite Aggregate Functions

## Overview of SQLite aggregate functions

- COUNT
  - 그룹의 항목 수를 가져옴
- AVG
  - 값 집합의 평균 값을 계산
- MAX
  - 그룹에 있는 모든 값의 최대값을 가져옴
- MIN
  - 그룹에 있는 모든 값의 최소값을 가져옴
- SUM
  - 모든 값의 합을 계산

<https://www.sqlitetutorial.net/sqlite-avg/>  
<https://www.sqlitetutorial.net/sqlite-count-function/>  
<https://www.sqlitetutorial.net/sqlite-max/>  
<https://www.sqlitetutorial.net/sqlite-min/>  
<https://www.sqlitetutorial.net/sqlite-sum/>

## COUNT

```
SELECT COUNT(컬럼) FROM 테이블이름;
```

레코드의 개수 조회하기

## COUNT

Q. users 테이블의 레코드 총 개수를 조회한다면?

## COUNT

Q. users 테이블의 레코드 총 개수를 조회한다면?

```
SELECT COUNT(*) FROM users;
```

## AVG, SUM, MIN, MAX (1/7)

```
SELECT AVG(컬럼) FROM 테이블이름;  
SELECT SUM(컬럼) FROM 테이블이름;  
SELECT MIN(컬럼) FROM 테이블이름;  
SELECT MAX(컬럼) FROM 테이블이름;
```

위 함수들은 기본적으로 해당 컬럼이 숫자(INTEGER)일 때만 사용 가능

## AVG, SUM, MIN, MAX (2/7)

Q. 30살 이상인 사람들의 평균 나이는?

## AVG, SUM, MIN, MAX (3/7)

Q. 30살 이상인 사람들의 평균 나이는?

```
SELECT AVG(age) FROM users WHERE age>=30;
```

## AVG, SUM, MIN, MAX (4/7)

Q. 계좌 잔액(balance)이 가장 높은 사람과 그 액수를 조회하려면?

## AVG, SUM, MIN, MAX (5/7)

Q. 계좌 잔액(balance)이 가장 높은 사람과 그 액수를 조회하려면?

```
SELECT first_name, MAX(balance) FROM users;
```

## AVG, SUM, MIN, MAX (6/7)

Q. 나이가 30 이상인 사람의 계좌 평균 잔액을 조회하려면?

## AVG, SUM, MIN, MAX (7/7)

Q. 나이가 30 이상인 사람의 계좌 평균 잔액을 조회하려면?

```
SELECT AVG(balance) FROM users WHERE age>=30;
```

LIKE

## LIKE operator

- to query data based on pattern matching
- 패턴 일치를 기반으로 데이터를 조회하는 방법
- sqlite는 패턴 구성을 위한 2개의 wildcards를 제공
  - % (percent sign)
    - 0개 이상의 문자
  - \_ (underscore)
    - 임의의 단일 문자

## LIKE statement

```
SELECT * FROM 테이블 WHERE 컬럼 LIKE '와일드카드패턴';
```

패턴을 확인하여 해당하는 값을 조회하기

## wildcards

### 와일드 카드 2가지 패턴

%

(percent sign)

이 자리에 문자열이  
있을 수도, 없을 수도 있다.

-

(underscore)

반드시 이 자리에  
한 개의 문자가 존재해야 한다.

## wildcards 사용 예시

```
SELECT * FROM 테이블 WHERE 컬럼 LIKE '와일드카드패턴';
```

와일드카드패턴	의미
2%	2로 시작하는 값
%2	2로 끝나는 값
%2%	2가 들어가는 값
_2%	아무 값이 하나 있고 두번 째가 2로 시작하는 값
1___	1로 시작하고 총 4자리인 값
2_%_ / 2__%	2로 시작하고 적어도 3자리인 값

## wildcards 실습 (1/8)

Q. users 테이블에서 나이가 20대인 사람만 조회한다면?

## wildcards 실습 (2/8)

Q. users 테이블에서 나이가 20대인 사람만 조회한다면?

```
SELECT * FROM users WHERE age LIKE '2_';
```

## wildcards 실습 (3/8)

Q. users 테이블에서 지역 번호가 02인 사람만 조회한다면?

## wildcards 실습 (4/8)

Q. users 테이블에서 지역 번호가 02인 사람만 조회한다면?

```
SELECT * FROM users WHERE phone LIKE '02-%';
```

## wildcards 실습 (5/8)

Q. users 테이블에서 이름이 ‘준’으로 끝나는 사람만 조회한다면?

## wildcards 실습 (6/8)

Q. users 테이블에서 이름이 ‘준’으로 끝나는 사람만 조회한다면?

```
SELECT * FROM users WHERE first_name LIKE '%준';
```

## wildcards 실습 (7/8)

Q. users 테이블에서 중간 번호가 5114인 사람만 조회한다면?

## wildcards 실습 (8/8)

Q. users 테이블에서 중간 번호가 5114인 사람만 조회한다면?

```
SELECT * FROM users WHERE phone LIKE '%-5114-%';
```

# ORDER BY

## ORDER BY clause

- **ORDER BY**
  - to sort a result set of a query
  - 조회 결과 집합을 정렬
  - SELECT 문에 추가하여 사용
  - 정렬 순서를 위한 2개의 keyword 제공
    - ASC – 오름차순 (default)
    - DESC – 내림차순

## ORDER BY

```
SELECT * FROM 테이블 ORDER BY 컬럼 ASC;  
SELECT * FROM 테이블 ORDER BY 컬럼1, 컬럼2 DESC;
```

특정 컬럼을 기준으로 데이터를 정렬해서 조회하기

ASC : 오름차순 (default)

DESC : 내림차순

## ORDER BY 실습 (1/6)

Q. users에서 나이 순으로 오름차순 정렬하여 상위 10개만 조회한다면?

## ORDER BY 실습 (2/6)

Q. users에서 나이 순으로 오름차순 정렬하여 상위 10개만 조회한다면?

```
SELECT * FROM users ORDER BY age ASC LIMIT 10;
```

## ORDER BY 실습 (3/6)

Q. 나이 순, 성 순으로 오름차순 정렬하여 상위 10개만 조회한다면?

## ORDER BY 실습 (4/6)

Q. 나이 순, 성 순으로 오름차순 정렬하여 상위 10개만 조회한다면?

```
SELECT * FROM users ORDER BY age, last_name ASC LIMIT 10;
```

## ORDER BY 실습 (5/6)

Q. 계좌 잔액 순으로 내림차순 정렬하여 해당 유저의 성과 이름을  
10개만 조회한다면?

## ORDER BY 실습 (6/6)

Q. 계좌 잔액 순으로 내림차순 정렬하여 해당 유저의 성과 이름을  
10개만 조회한다면?

```
SELECT last_name, first_name FROM users ORDER BY balance DESC LIMIT 10;
```

# GROUP BY

## GROUP BY clause

- **GROUP BY**
  - to make a set of summary rows from a set of rows.
  - 행 집합에서 요약 행 집합을 만듦
  - SELECT 문의 optional 절
  - 선택된 행 그룹을 하나 이상의 열 값으로 요약 행으로 만듦
  - 문장에 WHERE 절이 포함된 경우 반드시 WHERE 절 뒤에 작성해야 함

## GROUP BY 예시

```
SELECT 컬럼1, aggregate_function(컬럼2) FROM 테이블 GROUP BY 컬럼1, 컬럼2;  
aggregate 함수(앞에서 배운 COUNT, SUM, MAX, MIN 등등)
```

지정된 기준에 따라 행 세트를 그룹으로 결합  
데이터를 요약하는 상황에 주로 사용

## GROUP BY 실습 (1/2)

Q. users에서 각 성(last\_name)씨가 몇 명씩 있는지 조회한다면?

## GROUP BY 실습 (2/2)

Q. users에서 각 성(last\_name)씨가 몇 명씩 있는지 조회한다면?

```
SELECT last_name, COUNT(*) FROM users GROUP BY last_name;
```

## GROUP BY

Q. users에서 각 성(last\_name)씨가 몇 명씩 있는지 조회한다면?

```
SELECT last_name, COUNT(*) AS name_count FROM users GROUP BY last_name;
```

**AS** 를 활용해서 COUNT에 해당하는 컬럼 명을 바꿔서 조회할 수 있음

# ALTER TABLE

## 잠깐 복습 (1/4)

Q. title과 content라는 컬럼을 가진  
articles라는 이름의 table을 새롭게 만들어보세요!  
(두 컬럼 모두 비어 있으면 안되며, rowid를 사용합니다.)

## 잠깐 복습 (2/4)

Q. title과 content라는 컬럼을 가진  
articles라는 이름의 table을 새롭게 만들어보세요!  
(두 컬럼 모두 비어 있으면 안되며, rowid를 사용합니다.)

```
CREATE TABLE articles (
    title TEXT NOT NULL,
    content TEXT NOT NULL
);
```

## 잠깐 복습 (3/4)

Q. articles 테이블에 값을 추가해보세요!  
(title은 ‘1번제목’, content는 ‘1번내용’)

## 잠깐 복습 (4/4)

Q. articles 테이블에 값을 추가해보세요!  
(title은 ‘1번제목’, content는 ‘1번내용’)

```
INSERT INTO articles VALUES ('1번제목', '1번 내용');
```

## ALTER TABLE statement

ALTER TABLE의 3가지 기능

1. table 이름 변경
2. 테이블에 새로운 column 추가
3. [참고] column 이름 수정 (new in sqlite 3.25.0)

```
ALTER TABLE table_name  
RENAME COLUMN current_name TO new_name;
```

## ALTER TABLE 실습 (1/4)

방금 만든 테이블의 이름을 변경해보자

```
ALTER TABLE 기존테이블이름 RENAME TO 새로운테이블이름;
```

## ■ ALTER TABLE 실습 (2/4)

방금 만든 테이블의 이름을 변경해보자

```
sqlite> ALTER TABLE articles RENAME TO news;  
sqlite> .tables  
classmates examples news users
```

## ALTER TABLE 실습 (3/4)

방금 만든 테이블에 새로운 컬럼을 추가해보자

```
ALTER TABLE 테이블이름 ADD COLUMN 컬럼이름 데이터타입설정 ;
```

## ALTER TABLE 실습 (4/4)

새로운 컬럼 이름은 created\_at이며, TEXT 타입에 NULL 설정!

```
ALTER TABLE news ADD COLUMN created_at TEXT NOT NULL;
```

## 는 실패

실패... Error를 읽어보자!

```
sqlite> ALTER TABLE news ADD COLUMN created_at TEXT NOT NULL;  
Error: Cannot add a NOT NULL column with default value NULL
```

테이블에 있던 기존 레코드들에는 새로 추가할 필드에 대한 정보가 없다.  
그렇기 때문에 NOT NULL 형태의 컬럼은 추가가 불가능!

해결 방법 2가지

1. NOT NULL 설정 없이 추가하기
2. 기본 값(DEFAULT) 설정하기

## 1. NOT NULL 설정 없이 추가하기

```
sqlite> ALTER TABLE news ADD COLUMN created_at TEXT;  
sqlite> INSERT INTO news VALUES ( '제목', '내용', datetime( 'now' ) );  
sqlite> SELECT * FROM news;  
title          content        created_at  
-----          -----        -----  
1번제목        1번 내용      비어있음  
제목          내용          2021-06-03
```

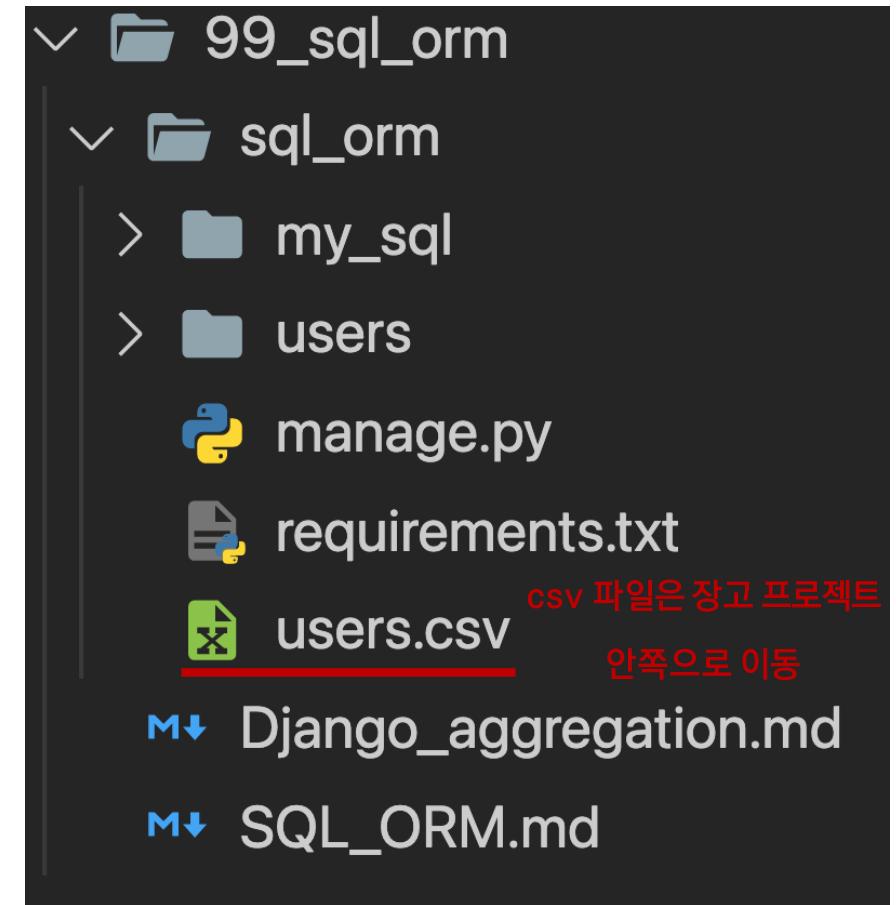
## 2. 기본 값(DEFAULT) 설정하기

```
sqlite> ALTER TABLE news ADD COLUMN subtitle TEXT NOT NULL DEFAULT '소제목';
sqlite> SELECT * FROM news;
title      content      created_at      subtitle
-----      -----      -----      -----
1번제목      1번 내용      소제목
제목          내용      2021-06-03      소제목      기본 값으로 설정 됨
```

# SQL & ORM

## SQL & ORM 실습 준비하기

- <http://abit.ly/ssafy-db> – SQL\_ORM 폴더
- DB 실습을 위한 별도의 폴더(99\_sql\_orm) 생성 및 관련 파일 다운로드



## Django Project Setting

1. 다운로드한 django 프로젝트에서 가상 환경 적용하기
2. requirements.txt 파일로 실습 환경 구성하기

```
$ pip install -r requirements.txt
```

3. migrate 진행하기

```
$ python manage.py migrate
```

```
$ python manage.py sqlmigrate users 0001
```

4. db.sqlite3 실행하기

```
$ sqlite3 db.sqlite3
```

## sqlite 확인 및 .headers on 설정

## csv 파일을 데이터 베이스에 적용하기

```
sqlite> .mode csv  
sqlite> .import users.csv users_user  
sqlite> SELECT * FROM users_user;  
1,"정호","유",40,"전라북도",016-7280-2855,370  
2,"경희","이",36,"경상남도",011-9854-5133,5900  
3,"정자","구",37,"전라남도",011-4177-8170,3100  
4,"미경","장",40,"충청남도",011-9079-4419,250000  
5,"영환","차",30,"충청북도",011-2921-4284,220  
...
```

## 스키마 확인

```
sqlite> .schema users_user
CREATE TABLE IF NOT EXISTS "users_user" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "first_name" varchar(10) NOT NULL,
    "last_name" varchar(10) NOT NULL,
    "age" integer NOT NULL,
    "country" varchar(10) NOT NULL,
    "phone" varchar(15) NOT NULL,
    "balance" integer NOT NULL);
```

## .headers on

```
sqlite> .headers on
sqlite> SELECT * FROM users_user;
id|first_name|last_name|age|country|phone|balance
1|정호|유|40|전라북도|016-7280-2855|370
2|경희|이|36|경상남도|011-9854-5133|5900
3|정자|구|37|전라남도|011-4177-8170|3100
4|미경|장|40|충청남도|011-9079-4419|250000
5|영환|차|30|충청북도|011-2921-4284|220
...
```

## shell 정리 및 종료

- in sqlite
  - shell clear
  - .exit
- in django\_shell\_plus
  - clear
  - exit

# CRUD

## 진행 TIP

- vscode 터미널을 좌우로 나누어서 진행
  1. 좌측 (sqlite shell)
    - SQL 명령어 작성 및 결과 확인
  2. 우측 (Django shell\_plus)
    - ORM 명령어 작성 및 결과 확인

The screenshot shows a VS Code interface with two terminal windows side-by-side. The left terminal window has a title bar with tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE, with TERMINAL being the active tab. It displays a command-line session starting with 'edu junho@DESKTOP-UUASRJ4 MINGW64 ~/Desktop/sql\_orm/sql\_orm' followed by a prompt '\$'. Below the terminal window is a dark sidebar with icons for file operations. The right terminal window also has a title bar with the same tabs, with DEBUG CONSOLE being the active tab. It displays a command-line session starting with 'edu junho@DESKTOP-UUASRJ4 MINGW64 ~/Desktop/sql\_orm/sql\_orm' followed by a prompt '\$'. Below the right terminal window is a dark sidebar with icons for file operations. Both terminal windows have red boxes highlighting the command lines '\$ sqlite3 db.sqlite3' in the left window and '\$ python manage.py shell\_plus' in the right window.

**READ**

## 모든 user 레코드 조회

모든 유저의 정보를 조회해보자

## 모든 user 레코드 조회 – ORM

```
User.objects.all()
```

## 모든 user 레코드 조회 – SQL

```
SELECT * FROM users_user;
```

**CREATE**

## | user 레코드 생성

새로운 유저의 정보를 등록해보자

## user 레코드 생성 – ORM

```
User.objects.create(  
    first_name='길동',  
    last_name='홍',  
    age=100,  
    country='제주도',  
    phone='010-1234-5678',  
    balance=10000  
)
```

## 방금 생성한 레코드 확인하기 – SQL

```
SELECT * FROM users_user LIMIT 1 OFFSET 100;
```

## user 레코드 생성 및 확인 – SQL

```
INSERT INTO users_user VALUES (102, '길동', '김', 100, '경상북도', '010-1234-1234', 100);
SELECT * FROM users_user LIMIT 2 OFFSET 100;
```

## user 레코드 생성 실패 예시 1 – SQL

```
sqlite> INSERT INTO users_user VALUES ('길동', '홍', 100, '제주도', '010-1234-4567', 100000);
Error: table users_user has 7 columns but 6 values were supplied
```

VALUES에 id를 작성하지 않은 상황

## user 레코드 생성 실패 예시 2 – SQL

```
sqlite> INSERT INTO users_user VALUES (102, '길동', '홍', 100, '제주도', '010-1234-5678', 10000);
Error: UNIQUE constraint failed: users_user.id
```

이미 존재하는 id를 작성한 상황

## user 레코드 생성 실패 예시 3 – SQL

```
sqlite> INSERT INTO users_user (id, first_name, last_name, age, country, phone)
VALUES (103, '길동', '홍', 100, '제주도', '010-1234-5678');
Error: NOT NULL constraint failed: users_user.balance
```

balance(NOT NULL)를 작성하지 않은 상황

## NOT NULL constraint failed를 ORM에서 확인해보자

```
In [3]: User.objects.create(  
...:     first_name='길동',  
...:     last_name='홍',  
...:     age=100,  
...:     country='제주도',  
...:     phone='010-1234-5678'  
...: )  
  
IntegrityError: NOT NULL constraint failed: users_user.balance
```

**READ**

## 특정 user 레코드 조회

id가 102인 유저의 정보를 조회해보자

## 특정 user 레코드 조회 – ORM

```
User.objects.get(pk=102)
```

## 특정 user 레코드 조회 – SQL

```
SELECT * FROM users_user WHERE id=102;
```

# UPDATE

## 특정 user 레코드 수정

id가 102인 유저의 정보를 수정해보자

## 특정 user 레코드 수정 – ORM

```
In [5]: user = User.objects.get(pk=102)
In [6]: user.last_name = '김'
In [7]: user.save()
In [8]: user.last_name
Out[8]: '김'
```

## 특정 user 레코드 수정 및 확인 – SQL

```
UPDATE users_user SET first_name='철수' WHERE id=102;  
SELECT * FROM users_user WHERE id=102;
```

**DELETE**

## 특정 user 레코드 삭제

id가 102인 유저의 정보를 삭제해보자

## 특정 user 레코드 삭제 – ORM

```
User.objects.get(pk=102).delete()
```

## 특정 user 레코드 삭제 – SQL

```
DELETE FROM users_user WHERE id=101;  
SELECT * FROM users_user WHERE id=101;
```

# SQL & ORM 활용하기

## 들어가기 전에

- 아래 2개의 문서를 참고하여 진행
- <https://docs.djangoproject.com/en/3.2/topics/db/queries/>
- <https://docs.djangoproject.com/en/3.2/ref/models/querysets/>

## 전체 유저의 수를 조회

전체 유저의 수를 조회해보자

## 전체 유저의 수를 조회 – ORM

```
User.objects.count()
```

## 전체 유저의 수를 조회 – SQL

```
SELECT COUNT(*) FROM users_user;
```

## 조건에 따른 쿼리문

특정 조건으로 데이터를 조회하기

## 내 나이가 어때서

나이가 30살인 사람들의 이름을 조회해보자

## 나이가 30살인 사람들의 이름 – ORM

```
User.objects.filter(age=30).values('first_name')
```

## 나이가 30살인 사람들의 이름 – SQL

```
SELECT first_name FROM users_user WHERE age=30;
```

ORM 객체에서 쿼리문도 확인해보자!

## 나이가 30살인 사람들의 이름 – 쿼리문 확인하기

```
print(User.objects.filter(age=30).values('first_name').query)
```

```
SELECT "users_user"."first_name" FROM "users_user" WHERE "users_user"."age" = 30
```

## 조건에 따른 쿼리문

### 대/소 관계 비교 조건

`__gte, __gt, __lte, __lt`

## 조건에 따른 쿼리문

나이가 30살 이상인 사람의 인원 수

## 나이가 30살 이상인 사람의 인원 수 – ORM

```
User.objects.filter(age__gte=30).count()
```

## 나이가 30살 이상인 사람의 인원 수 – SQL

```
SELECT COUNT(*) FROM users_user WHERE age>=30;
```

## 조건에 따른 쿼리문

나이가 20살 이하인 사람의 인원 수

## 나이가 20살 이하인 사람의 인원 수 – ORM

```
User.objects.filter(age__lte=20).count()
```

## 나이가 20살 이하인 사람의 인원 수 – SQL

```
SELECT COUNT(*) FROM users_user WHERE age<=20;
```

AND

나이가 30살이면서 성이 김씨인 사람의 인원 수

## 나이가 30살이면서 성이 김씨인 사람의 인원 수 – ORM

```
User.objects.filter(age=30, last_name='김').count()
```

```
User.objects.filter(age=30).filter(last_name='김').count()
```

## 나이가 30상이면서 성이 김씨인 사람의 인원 수 – SQL

```
SELECT COUNT(*) FROM users_user WHERE age = 30 AND last_name = '김';
```

OR

나이가 30살이거나 성이 김씨인 사람의 인원 수

## 나이가 30살이거나 성이 김씨인 사람의 인원 수 – ORM

```
from django.db.models import Q
User.objects.filter(Q(age=30) | Q(last_name='김'))
```

- [참고] OR 을 활용하고 싶다면, **Q object** 를 활용해야 함
  - encapsulate a collection of keyword arguments.
  - ‘|’(OR) 및 ‘&’(AND)와 같은 연산자를 사용하여 조건을 정의 및 재사용
  - <https://docs.djangoproject.com/en/3.2/topics/db/queries/#complex-lookups-with-q-objects>

## 나이가 30살이거나 성이 김씨인 사람의 인원 수 – SQL

```
SELECT * FROM users_user WHERE age=30 OR last_name='김';
```

## LIKE

지역번호가 02인 사람의 인원 수

## 지역번호가 02인 사람의 인원 수 – ORM

```
User.objects.filter(phone__startswith='02-').count()
```

## 지역번호가 02인 사람의 인원 수 – SQL

```
SELECT COUNT(*) FROM users_user WHERE phone LIKE '02-%';
```

## 특정 컬럼 데이터만 조회하기

주소가 강원도이면서 성이 황씨인 사람의 이름

## 주소가 강원도이면서 성이 황씨인 사람의 이름 – ORM

```
User.objects.filter(country='강원도', last_name='황').values('first_name')
```

## 주소가 강원도이면서 성이 황씨인 사람의 이름 – SQL

```
SELECT first_name FROM users_user WHERE country='강원도' AND last_name='황';
```

## 정렬, LIMIT, OFFSET

나이가 많은 사람순으로 10명만 조회

## 나이가 많은 사람순으로 10명 – ORM

```
User.objects.order_by(' -age ')[:10]
```

## 나이가 많은 사람순으로 10명 – SQL

```
SELECT * FROM users_user ORDER BY age DESC LIMIT 10;
```

## 정렬, LIMIT, OFFSET

잔액이 적은 사람순으로 10명만 조회

## 잔액이 적은 사람순으로 10명 – ORM

```
User.objects.order_by('balance')[ :10 ]
```

## 잔액이 적은 사람순으로 10명 – SQL

```
SELECT * FROM users_user ORDER BY balance ASC LIMIT 10;
```

## 정렬, LIMIT, OFFSET

잔액이 적고, 나이가 많은순으로 10명만 조회

## 잔액이 적고, 나이가 많은순으로 10명 – ORM

```
User.objects.order_by('balance', '-age')[:10]
```

## 잔액이 적고, 나이가 많은순으로 10명 – SQL

```
SELECT * FROM users_user ORDER BY balance, age DESC LIMIT 10;
```

## 정렬, LIMIT, OFFSET

성, 이름 내림차순으로 5번째 있는 유저 정보 조회

## 성, 이름 내림차순으로 5번째 있는 유저 정보 조회 – ORM

```
User.objects.order_by('-last_name', '-first_name')[4]
```

## 성, 이름 내림차순으로 5번째 있는 유저 정보 조회 – SQL

```
SELECT * FROM users_user ORDER BY last_name DESC, first_name DESC LIMIT 1 OFFSET 4;
```

# Django Aggregation

## 들어가기 전에

- 아래 문서를 참고하여 진행
- <https://docs.djangoproject.com/en/3.2/topics/db/aggregation/>

## aggregate( )

- Returns a dictionary of aggregate values (averages, sums, etc.) calculated over the QuerySet.
- ‘무언가를 종합, 집합, 합계’ 등의 사전적 의미
- 특정 필드 전체의 합, 평균, 개수 등을 계산할 때 사용

## aggregate 사용하기

전체 유저의 평균 나이

## 전체 유저의 평균 나이 – SQL

```
SELECT AVG(age) FROM users_user;
```

## aggregate 사용하기

성이 김씨인 유저들의 평균 나이

## 성이 김씨인 유저들의 평균 나이 – ORM

```
from django.db.models import Avg  
User.objects.filter(last_name='김').aggregate(Avg('age'))
```

## 성이 김씨인 유저들의 평균 나이 – SQL

```
SELECT AVG(age) FROM users_user WHERE last_name = '김';
```

## aggregate 사용하기

지역이 강원도인 유저들의 평균 계좌 잔고

## 지역이 강원도인 유저들의 평균 계좌 잔고 – ORM

```
from django.db.models import Avg  
User.objects.filter(country='강원도').aggregate(Avg('balance'))
```

## 지역이 강원도인 유저들의 평균 계좌 잔고 – SQL

```
SELECT AVG(balance) FROM users_user WHERE country='강원도';
```

## aggregate 사용하기

계좌의 잔고 중 가장 높은 값

## 계좌의 잔고 중 가장 높은 값 – ORM

```
from django.db.models import Max  
User.objects.aggregate(Max('balance'))
```

## 계좌의 잔고 중 가장 높은 값 – SQL

```
SELECT MAX(balance) FROM users_user;
```

## aggregate 사용하기

계좌 잔고의 총 합

## 계좌 잔고의 총 합 – ORM

```
from django.db.models import Sum  
User.objects.aggregate(Sum('balance'))
```

## 계좌 잔고의 총 합 – SQL

```
SELECT SUM(balance) FROM users_user;
```

## | **annotate( )**

- Annotates each object in the QuerySet with the provided list of query expressions
- ‘주석을 달다’라는 사전적 의미
- 마치 컬럼 하나를 추가하는 것과 같음
  - 특정 조건으로 계산된 값을 가진 컬럼을 하나 만들고 추가하는 개념
- annotate()에 대한 각 인자는 반환되는 QuerySet의 각 객체에 추가될 주석임

## Annotate 사용하기 (1/2) – “지역별 인원 수”

- Annotate는 새로운 컬럼(주석)을 만들어 냄 (원본 테이블이 변하는 것이 아님)

```
from django.db.models import Count

# 1
User.objects.values('country').annotate(Count('country'))
# <QuerySet [{'country': '강원도', 'country__count': 14}, {'country': '경기도', 'country__count': 9}, ...]

# 2
User.objects.values('country').annotate(num_countries=Count('country'))
# <QuerySet [{'country': '강원도', 'num_countries': 14}, {'country': '경기도', 'num_countries': 9}, ...]

print(User.objects.values('country').annotate(Count('country')).query)
# SELECT "users_user"."country", COUNT("users_user"."country") AS "country__count" FROM "users_user" GROUP BY "users_user"."country"

# 3
User.objects.values('country').annotate(Count('country'), avg_balance=Avg('balance'))
# <QuerySet [{'country': '강원도', 'country__count': 14, 'avg_balance': 157895.0}, ...]
```

## Annotate 사용하기 (2/2) – “지역별 인원 수”

- Annotate는 새로운 컬럼(주석)을 만들어 냄 (원본 테이블이 변하는 것이 아님)

```
SELECT country, COUNT(country) FROM users_user GROUP BY country;
```

-- 실행 결과  
강원도|14  
경기도|9  
경상남도|9  
경상북도|15  
전라남도|10  
전라북도|11  
제주특별자치도|9  
충청남도|9  
충청북도|14