3

3장 : Select문

SELECT 문은 SQL 문에서 가장 많이 사용되는 문법으로, 데이터베이스에서 데이터를 구축한 후에 그 내용들을 활용한다. 3장에서는 SELECT 문을 살펴본다.

3-1 : SELECT ~ FROM ~ WHERE

SELECT 문은 구축이 완료된 테이블에서 데이터를 추출하는 기능을 한다. SELECT 문의 가장 기본적인 형식은 **SELECT 열 이름 FROM 테이블 이름 WHERE 조건식** 이다.

실습용 데이터베이스 구축

책에 실습용 데이터베이스를 만드는 방법이 나와있지만 대부분 아직 배우지 않은 SQL이기 때문에 직접 입력하기엔 무리가 있으며 책에 적힌 대로 다운로드 받고, 진행하면 된다.

1. https://www.hanbit.co.kr/support/supplement_survey.html?pcode=B6846155853 에 접속해서 예제 소스를 다운로드하고 압축을 푼다. 그리고 MySQL을 실행해서 현재 열려 있는 쿼리 창을 모두 닫고 **[File]→[Open SQL Script]** 를 선택하고 Open SQL Script 창에서 앞에서 다운로드한 market_db.sql을 선택한 후 열기를 누른다.

2. **Execute the selected portion of the script or everything(번개모양 아이콘)**을 클릭해서 SQL을 실행한다. [Result Grid] 창의 하단에서 회원 테이블(member)과 구매 테이블(buy)을 확인해본다. 이로써 인터넷 마켓 DB 구성도가 완성되었다.

지금 실습한 내용은 언제든지 다시 실행할 수 있으며 market_db가 초기화되는 효과를 갖는다. 앞으로 실습을 진행하다가 market_db를 초기화해야 하는 상황이 많이 발생한다. 그때마다 지금과 같이 초기화시키도록 한다.

market_db.sql 파일 살펴보기

2장에서는 shop_db를 MySQL의 기능을 활용해서 만들었다. 이와 달리 인터넷 마켓 데이터베이스를 만드는 market_db.sql은 모두 SQL로 구성되어 있다. 아직 배우지 않은 내용이기 때문에 이해하기는 어려울 수도 있으나 앞으로 이 내용으로 SQL을 학습하기 때문에 필요한 부분을 위주로 먼저 살펴본다.

```
DROP DATABASE IF EXISTS market_db;   → 1

CREATE DATABASE market_db;           → 2
```

1. DROP DATABASE는 market_db를 삭제하는 문장이다. 이는 market_db.sql을 처음 실행할 때는 필요 없지만 이후 실습에서 다시 실행해야 할 때, 기존의 market_db를 삭제해주는 SQL문이다.

2. 데이터베이스를 새로 만드는 SQL문이다. 2장에서 실습할 때 봤던 것과 동일한 역할을 합니다.

회원 테이블 만들기

```
USE market_db;                                  → 1
CREATE TABLE member                             →2
( mem_id        CHAR(8) NOT NULL PRIMARY KEY,
mem_name        VARCHAR(10) NOT NULL,
mem_number      INT NOT NULL,
addr            CHAR(2) NOT NULL,
phone1      CHAR(3),
phone2          CHAR(8),
height      SMALLINT,
debut_date DATE
);                                              →2
```

1. USE 문은 market_db 데이터베이스를 선택하는 문장이다. 2장에서 우린 스키마 패널에서 shop_db 데이터베이스를 클릭해서 선택했다. 그 동작과 동일한 기능을 한다.

2. member 테이블을 만드는 과정이다. 여기서 열 이름은 영문으로 표현했다. 열 이름 다음에 데이터 타입과 NN, 키 지정을 적었는데 이는 우리가 2장에서 테이블을 만드는 과정과 동일한 과정이다. CHAR, INT는 2장에서 살펴본 내용이지만 VARCHAR, SMALLINT는 처음 보는 타입이다. **VARCHAR**는 데이터가 길거나, 길이가 다양하게 변동할 가능성이 있을 때 사용한다. 반면 CHAR는 고정된 길이로 저장한다. **SMALLINT**는 작은 정수를 저장하기 위한 타입이며 메모리와 저장 공간을 절약할 수 있다. 차이점은 나중에 더 자세히 설명한다.

구매 테이블 만들기

```
CREATE TABLE buy                                        →1
( num       INT AUTO_INCREMENT NOT NULL PRIMARY KEY,        →2
mem_id      CHAR(8) NOT NULL,
prod_name       CHAR(6) NOT NULL,
group_name      CHAR(4)  ,
price       INT  NOT NULL,
amount      SMALLINT  NOT NULL,
FOREIGN KEY (mem_id) REFERENCES member(mem_id)          →3
);
```

1. 구매 테이블을 생성한다.

2. **AUTO_INCREMENT**는 자동으로 숫자를 입력해준다는 의미이다. 순번을 직접 입력할 필요 없이 1,2,3,... 과 같은 방식으로 자동으로 증가하는 숫자를 만든다. 잠시 후 다시 확인한다.

3. **FOREIGN KEY**도 처음 나왔는데 외래 키라고 한다. 5장에서 살펴본다.

데이터 입력하기

이제 데이터를 입력하는 INSERT문을 살펴본다.

```
INSERT INTO member VALUES('TWC', '트와이스', 9, '서울', '02', '11111111', 167, '2015.10.19');
INSERT INTO member VALUES('BLK', '블랙핑크', 4, '경남', '055', '22222222', 163, '2016.08.08');
INSERT INTO member VALUES('WMN', '여자친구', 6, '경기', '031', '33333333', 166, '2015.01.15');
INSERT INTO member VALUES('OMY', '오마이걸', 7, '서울', NULL, NULL, 160, '2015.04.21');
INSERT INTO member VALUES('GRL', '소녀시대', 8, '서울', '02', '44444444', 168, '2007.08.02');
INSERT INTO member VALUES('ITZ', '잇지', 5, '경남', NULL, NULL, 167, '2019.02.12');
INSERT INTO member VALUES('RED', '레드벨벳', 4, '경북', '054', '55555555', 161, '2014.08.01');
INSERT INTO member VALUES('APN', '에이핑크', 6, '경기', '031', '77777777', 164, '2011.02.10');
INSERT INTO member VALUES('SPC', '우주소녀', 13, '서울', '02', '88888888', 162, '2016.02.25');
INSERT INTO member VALUES('MMU', '마마무', 4, '전남', '061', '99999999', 165, '2014.06.19');

INSERT INTO buy VALUES(NULL, 'BLK', '지갑', NULL, 30, 2);
INSERT INTO buy VALUES(NULL, 'BLK', '맥북프로', '디지털', 1000, 1);
INSERT INTO buy VALUES(NULL, 'APN', '아이폰', '디지털', 200, 1);
INSERT INTO buy VALUES(NULL, 'MMU', '아이폰', '디지털', 200, 5);
INSERT INTO buy VALUES(NULL, 'BLK', '청바지', '패션', 50, 3);
INSERT INTO buy VALUES(NULL, 'MMU', '에어팟', '디지털', 80, 10);
INSERT INTO buy VALUES(NULL, 'GRL', '혼공SQL', '서적', 15, 5);
INSERT INTO buy VALUES(NULL, 'APN', '혼공SQL', '서적', 15, 2);
INSERT INTO buy VALUES(NULL, 'APN', '청바지', '패션', 50, 1);
INSERT INTO buy VALUES(NULL, 'MMU', '지갑', NULL, 30, 1);
INSERT INTO buy VALUES(NULL, 'APN', '혼공SQL', '서적', 15, 1);
INSERT INTO buy VALUES(NULL, 'MMU', '지갑', NULL, 30, 4);
```

위에는 회원테이블의 데이터 입력, 아래는 구매테이블의 데이터 입력이다.

INSERT INTO 테이블이름 을 먼저 적어 어떤 테이블에 데이터를 입력하는지 지정한다. 그 후 데이터를 입력하는데 먼저 **CHAR, VARCHAR, DATE** 타입은 작은 따옴표로 값을 묶어준다. **INT**형은 그냥 넣어주면 된다.

구매 테이블의 첫 번째 열린 순번(NUM)은 위에서 **AUTO_INCREMENT**를 통해 자동으로 입력되므로 그 자리에는 NULL이라고 쓴 것을 알 수 있다.

이렇게 해서 MySQL 워크벤치를 사용해서 데이터베이스를 구축하는 것, 즉 2장에서 데이터베이스를 구축하는 것과 동일한 작업을 SQL로도 진행할 수 있다는 것을 확인했다.

기본 조회하기 : SELECT ~ FROM

이제 본격적으로 인터넷 마켓 DB 구성도를 활용해서 SELECT 문을 배워본다.

USE 문

SELECT 문을 실행하려면 먼저 사용할 데이터베이스를 지정해야 한다. 현재 사용하는 데이터베이스를 지정 또는 변경하는 형식은 **USE DB이름;** 을 사용하면 된다. 이렇게 지정해 놓은 후에 다시 USE 문을 사용하거나 다른 DB를 사용하겠다고 명시하지 않으면 앞으로 모든 SQL문은 현재DB에서 수행된다. MySQL 워크벤치를 재시작하거나 쿼리 창을 새로 열면 다시 USE를 실행해야 한다.

SELECT 문의 기본 형식 (대괄호로 묶인 부분은 생략이 가능하다.)

SELECT 열_이름	테이블에서 데이터를 불러보는 SELECT 문
[FROM 테이블_이름]	데이터를 가져올 테이블 이름
[WHERE 조건식]	특정한 조건을 추가해서 원하는 데이터만 보고싶을 때 사용
[GROUP BY 열_이름]	
[HAVING 조건식]	
[ORDER BY 열_이름]	
[LIMIT 숫자]	

SELECT 다음엔 추출하고 싶은 열 이름을 적으면 된다. 테이블의 모든 열을 추출하고 싶으면 * 을 사용하면 되고, 여러 개의 열을 추출하고 싶을 땐, 열 이름 사이에 쉼표 (,)를 사용하면 된다.

다음으로 FROM이다. 원칙적으로 **SELECT ~ FROM DB이름.테이블_이름**을 적지만 DB의 이름을 생략하면 USE 문으로 지정해 놓은 데이터베이스가 자동으로 선택된다. 따라서

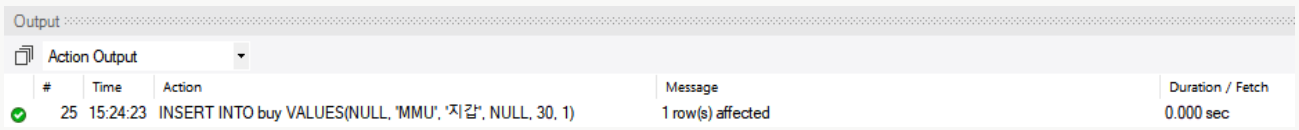
SELECT * FROM MARKET_DB.MEMBER ;

SELECT * FROM MEMBER;

두 쿼리는 동일하다. 대부분 테이블 이름만 사용한다.

💬 **OUTPUT 패널**

OUTPUT 패널의 내용 정도는 확인하는 편이 좋다. 의미를 살펴보자



- **초록색 체크 표시** : SQL이 정상적으로 실행됨
- **빨간색 x 표시** : SQL에 오류가 발생
- **#** : 실행한 SQL의 순번
- **Time** : SQL을 실행한 시각이 표시
- **Action** : 실행된 SQL이 표시
- **Message** : SELECT 문이 조회된 행의 개수, 오류 발생 시 오류 번호 및 오류 메시지 표시
- **Duration/Fetch** : Duration은 SQL 문이 실행되는 데 걸린 시간(초), Fetch는 DB에서 가져온 시간

열 이름에 **별칭(alias)**을 지정할 수 있다. 열 이름 다음에 지정하고 싶은 별칭을 입력하면 된다. 별칭에 공백이 있으면 큰따옴표(")로 묶어준다.

SELECT addr, debut_date, mem_name FROM member;

addr	debut_date	mem_name
경기	2011-02-10	에이핑크
경남	2016-08-08	블랙핑크
서울	2007-08-02	소녀시대
경남	2019-02-12	잇지
전남	2014-06-19	마마무
서울	2015-04-21	오마이걸
경북	2014-08-01	레드벨벳
서울	2016-02-25	우주소녀
서울	2015-10-19	트와이스
경기	2015-01-15	여자친구

SELECT addr 주소, debut_date "데뷔 일자", mem_name FROM member;

	주소	데뷔 일자	mem_name
▶	경기	2011-02-10	에이핑크
	경남	2016-08-08	블랙핑크
	서울	2007-08-02	소녀시대
	경남	2019-02-12	잇지
	전남	2014-06-19	마마무
	서울	2015-04-21	오마이걸
	경북	2014-08-01	레드벨벳
	서울	2016-02-25	우주소녀
	서울	2015-10-19	트와이스
	경기	2015-01-15	여자친구

열 이름이 변경되었다.

WHERE 절

WHERE 절은 조회하는 결과에 특정한 조건을 추가해서 원하는 데이터를 추출하는 방법이다. 지금 찾는 이름이(mem_name)이 '블랙핑크' 라면 다음과 같은 조건식을 사용하면 된다. 열_이름 = 값에 해당하는 결과만 출력한다. 이름과 같이 조건이 문자형이면 작은따옴표로 묶어주고, 숫자형이면 작은 따옴표 없이 사용해도 된다.

SELECT * FROM member WHERE mem_name = '블랙핑크';

mem_id	mem_name	mem_number	addr	phone1	phone2	height	debut_date
BLK	블랙핑크	4	경남	055	22222222	163	2016-08-08
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

SELECT * FROM member WHERE mem_number = 4;

	mem_id	mem_name	mem_number	addr	phone1	phone2	height	debut_date
▶	BLK	블랙핑크	4	경남	055	22222222	163	2016-08-08
	MMU	마마무	4	전남	061	99999999	165	2014-06-19
	RED	레드벨벳	4	경북	054	55555555	161	2014-08-01
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

관계 연산자, 논리 연산자의 사용

구분	연산자	설명
비교 연산자	= / > / < / >= / <= / <>	같음 / 보다 큼 / 보다 작음 / 크거나 같음 / 작거나 같음 / 같지 않음
논리 연산자	AND / OR	앞, 뒤 조건 모두 만족 / 하나라도 만족
	NOT	뒤에 오는 조건과 반대
특수 연산자	BETWEEN a AND b / NOT BETWEEN a AND b	a와 b의 값 사이 / a와 b의 값 사이가 아님
	IN (List) / NOT IN (List)	리스트(List) 값 / 리스트(List) 값이 아님
	LIKE '비교문자열'	비교문자열과 같음
	NOT LIKE '비교문자열'	비교문자열이 아님
	IS NULL	NULL과 같음
	IS NOT NULL	NULL이 아님
산술 연산자	+ , - , * , /	덧셈, 뺄셈, 곱셉, 나눗셈
집합 연산자	UNION	2개 이상 테이블 중복된 행 제거 하여 집합(* 열 개수와 데이터 타입 일치)
	UNION ALL	2개 이상 테이블 중복된 행 제거 없이 집합(* 열 개수와 데이터 타입 일치)

다음과 같은 연산자들이 존재하는데 대부분 직관적으로 이해할 수 있다. 여기서 몇 가지만 좀더 살펴보자.

- **(NOT) BETWEEN a AND B**는 범위가 있는 값을 구하는 경우에 사용할 수 있다. 또한, 데이터는 수치형인데 Type이 문자형인 경우에도 작은 따옴표로 조건을 묶어 주면 사용가능하다.

  ex)WHERE height BETWEEN '2' AND '10' ;

- **IN()**은 조건식에서 여러 문자 중 포함되는 문자가 있으면 추출해주는 조건문이다.

  ex) WHERE addr IN ('경기', '전남', '경남') ;

- **LIKE**는 문자열의 일부 글자를 검색할 때 사용한다. 예를 들어 이름(mem_name)의 첫 글짜가 '우'로 시작하는 회원은 다음과 같이 검색할 수 있다. 이 조건은 제일 앞 글자가 '우' 이고 그 뒤는 무엇이든(%) 허용한다는 의미이다.

  ex) WHERE mem_name LIKE '우%';

  한 글자와 매치하기 위해선 언더바 (_)를 사용한다. 다음 SQL은 이름(mem_name)의 앞 두 글자는 상관없고 뒤는 '핑크'인 회원을 검색한다.

  ex) WHERE mem_name LIKE '__핑크' ; → 언더바 2개

💬 **서브 쿼리**

SELECT 안에 또 다른 SELECT가 들어갈 수 있다. 이것을 서브 쿼리 또는 하위 쿼리라고 부른다.

여기서 이름(mem_name)이 '에이핑크'인 회원의 평균 키(height)보다 큰 회원을 검색하고 싶다고 생각해보자.

우선 에이핑크의 평균 키를 알아내야 한다.

SELECT height FROM member WHERE mem_name = '에이핑크';

에이핑크의 키가 164인 것을 알아 냈다. 이제는 164보다 평균 키가 큰 회원을 조회하면 된다.

SELECT mem_name,height FROM member WHERE height > 164;

를 사용하면 결과를 얻을 수 있다. 이 두개의 SQL을 하나로 만들려면 두 번째 SQL의 164 위치에 에이핑크의 평균 키를 조회하는 SQL을 쓰면 된다.

```
SELECT mem_name, height FROM member
    WHERE height > (SELECT height FROM member WHERE mem_name = '에이핑크) ;
```

세미콜론이 하나이므로 이 SQL은 하나의 문장이다. 서브 쿼리의 장점은 2개의 SQL을 하나로 만듦으로써 하나의 SQL만 관리하면 되므로 더 간단하게 만들어 준다는 것이다.

3-2 : SELECT 문 심화

SELECT 문에서는 결과의 정렬을 위한 **ORDER BY**, 결과의 개수를 제한하는 **LIMIT**, 중복된 데이터를 제거하는 **DISTINCT** 등을 사용할 수 있다. 그리고 **GROUP BY** 절은 지정한 열의 데이터들을 같은 데이터끼리 묶어서 결과를 추출한다. 주로 그룹으로 묶는 경우는 합계, 평균, 개수 등을 처리할 때 사용하므로 집계 함수와 함께 사용된다. GROUP BY절에도 HAVING 절을 통해 조건식을 추가할 수 있다. HAVING 절은 WHERE 절과 비슷해보이지만, GROUP BY절과 함께 사용되는 것이 차이점이다.

ORDER BY 절

ORDER BY절은 결과의 값이나 개수에 대해서 영향을 미치지 않지만, 결과가 출력되는 순서를 조절한다. 회원 테이블에서 mem_id, mem_name, debut_date 를 추출할 때 debut_date 가 빠른 순서대로 뽑고 싶으면 다음과 같은 SQL을 작성하면 된다.

SELECT mem_id, mem_name, debut_date FROM member ORDER BY debut_date;

가장 늦은 순서대로 정렬하려면 **ORDER BY debut_date DESC** 를 적어주면 된다. 명시를 안하면 기본값인 오름차순(ASC)로 정렬된다.

ORDER BY 정렬 원하는 열 ASC(default) or DESC

정렬 기준은 1개 열이 아니라 **여러 개의 열로 지정할 수 있다.** 우선 첫 번째 지정 열로 정렬했을 때 동일한 경우엔 다음 지정 열로 정렬할 수 있다.

SELECT mem_id, mem_name, debut_date, height FROM member WHERE height ≥ 164

ORDER BY height DESC, debut_date ASC;

해당 SQL을 살펴보면 키가 164 이상인 행의 mem_id,mem_name,debut_date를 키가 큰 순서대로 정렬한 후에 키가 동일한 경우 데뷔 일자가 빠른 순서로 정렬한다.

> 💬 WHERE 문과 ORDER BY 문을 같이 사용할 때 먼저 WHERE 문을 사용해야 한다.

LIMIT : 출력 개수 제한

LIMIT 은 출력하는 개수를 제한한다. 아무 제한 없이 정해진 개수만 뽑는 경우는 별로 없고 **ORDER BY절과 함께 사용**하여 상위, 하위 몇 개씩 뽑는 방식으로 사용한다. LIMIT의 형식은 **LIMIT 시작, 개수**이다.

LIMIT 3 이라고 적으면 **LIMIT 0, 3**과 동일한 결과를 준다. 즉 **0번째부터 3건**이라는 의미이다. 필요하다면 **LIMIT 3, 2**과 같이 사용해서 3번째부터 2건을 조회하는 것처럼 **중간부터 출력**도 가능하다.

SELECT mem_name, height FROM member ORDER BY height DESC LIMIT 3, 2 ;

DISTINCT : 중복된 결과 제거

DISTNCT는 조회된 결과에서 중복된 데이터를 1개만 남긴다. 즉 **unique value를 확인**할 때 사용하면 된다. 유용하게 사용되는 구문이므로 반드시 기억하고 다음과 같이 사용된다.

SELECT DISTINCT addr FROM member;

GROUP BY

GROUP BY 문은 말 그대로 그룹을 묶어주는 역할을 한다. 3-1에서 불러온 구매 테이블과 회원 테이블을 생각하자. 구매 테이블(buy)에서 회원(mem_id)이 구매한 물품의 개수를 구하고 싶다고 가정하자.

SELECT mem_id, amount FROM buy ORDER BY mem_id

mem_id	amount
APN	1
APN	2
APN	1
APN	1
BLK	2
BLK	1
BLK	3
GRL	5
MMU	5
MMU	10
MMU	1
MMU	4

위의 SQL을 통해 회원들이 구매한 물품의 개수를 구했다. 회원별로 여러 건의 물건 구매가 있었고, APN 회원의 경우엔 1+2+1+1=5 개의 물건이 있었다. 합계를 이렇게 암산으로 할 필요없이 **GROUP BY 절과 집계함수**를 같이 사용하면 바로 구매 물품 개수를 구할 수 있다.

집계함수

GROUP BY와 함께 주로 사용되는 집계 함수는 다음 표와 같다.

함수명	함수표기	설 명
SUM	SUM()	합계를 구한다.
AVG	AVG()	평균을 구한다.
MIN	MIN()	최솟값을 구한다.
MAX	MAX()	최댓값을 구한다.
COUNT	COUNT()	행의 개수를 센다.
COUNT DISTINCT	COUNT(DISTINCT)	행의 개수를 센다. (중복은 1개만 인정)

COUNT(*)은 모든 행의 개수를 새고 COUNT(열 이름)은 해당 열의 값이 NULL인 것을 제외한 행의 개수를 센다.

각 회원별로 구매한 개수를 합쳐서 출력하기 위해서 GROUP BY와 SUM을 사용하면 된다.

SELECT mem_id, SUM(amount) FROM buy GROUP BY mem_id;

	mem_id	SUM(amount)
▶	APN	5
	BLK	6
	GRL	5
	MMU	20

이번에는 회원이 구매한 금액의 총합을 출력해본다.

SELECT mem_id, SUM(price*amount) FROM buy GROUP BY mem_id;

	MEM_ID	SUM(AMOUNT*PRICE)
▶	APN	295
	BLK	1210
	GRL	75
	MMU	1950

HAVING

앞에서 살펴봤던 SUM()으로 회원(mem_id)별 총 구매액을 구해보자

SELECT mem_id ,SUM(price*amount) '총 구매 금액' FROM buy GROUP BY mem_id

	MEM_ID	총 구매 금액
▶	APN	295
	BLK	1210
	GRL	75
	MMU	1950

결과 중에서 총 구매액이 1000 이상인 회원에게만 사은품을 증정하려면 어떻게 해야 할까? 이런 경우 HAVING 절을 사용하면 된다. HAVING과 WHERE 둘 다 조건을 주는 SQL문이지만, **HAVING은 집계함수에 대해서 조건을 제한하는 것**이라고 생각하면 된다. 또한 **HAVING 절은 GROUP BY 절 다음에 나와야 한다.**

SELECT mem_id, SUM(price*amount) FROM buy GROUP BY mem_id HAVING SUM(amount*price) >1000;

	mem_id	SUM(price*amount)
▶	BLK	1210
	MMU	1950

여기다가 만약 총 구매액이 큰 사용자부터 나타내면 HAVING 절 다음에 ORDER BY를 사용하면 된다.

3-3 : 데이터 변경을 위한 SQL 문

데이터베이스와 테이블을 만든 후에는 데이터를 입력/수정/삭제하는 기능이 필요하다. 새로운 데이터를 입력할 땐 **INSERT** 문, 정보를 수정할 때는 **UPDATE** 문, 데이터를 삭제할 때는 **DELETE** 문을 사용한다.

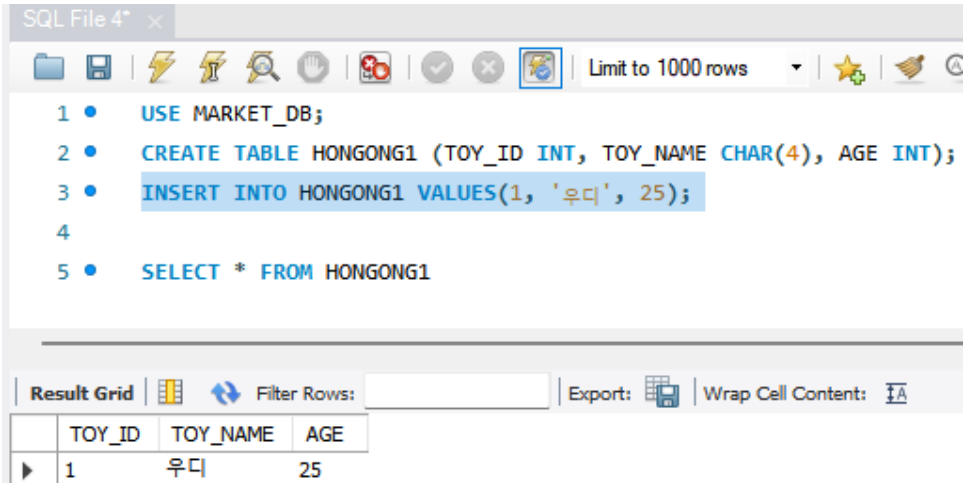
INSERT : 데이터 입력

테이블에 행 데이터를 입력하는 기본적인 SQL 문은 INSERT이다. 기본적인 형식은 다음과 같다.

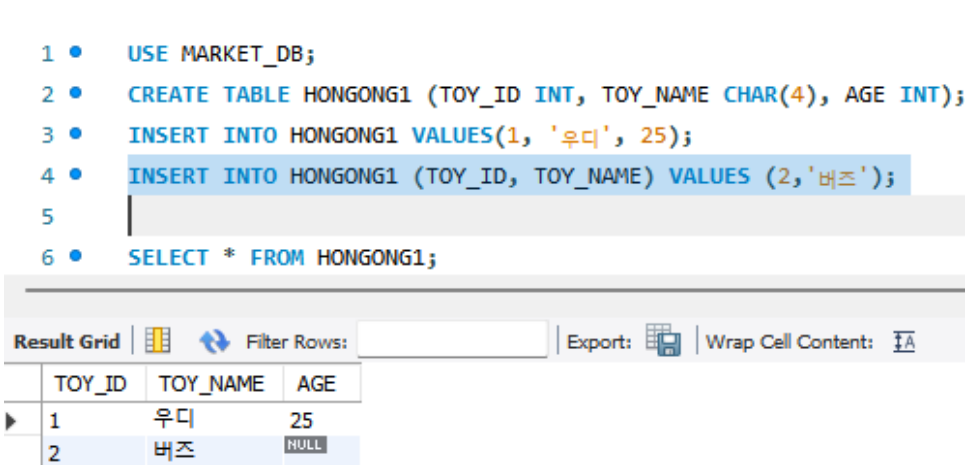
INSERT INTO 테이블 [(열1, 열2, ...)] VALUES (값1, 값2,...)

INSERT 문은 어렵지 않으니 주의할 점 몇 가지만 살펴본다.

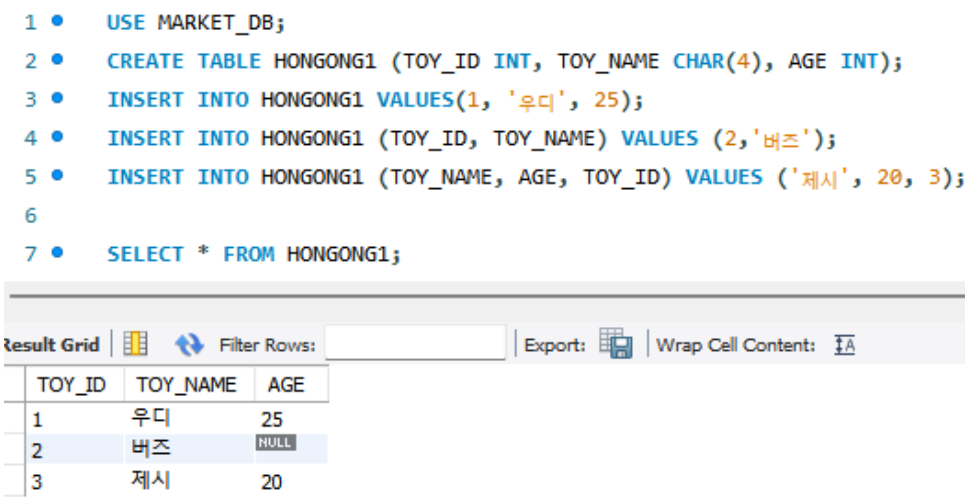
- 테이블 이름 다음에 나오는 [(열1, 열2,...)]은 생략이 가능하다. 열 이름을 생략할 경우엔 VALUES 다음에 나오는 값들의 순서 및 개수를 테이블을 정의할 때의 열 순서 및 개수와 동일해야 한다.



- 해당 예제에서 아이디와 이름만 입력하고 나이는 입력하고 싶지 않다면 다음과 같이 테이블 이름 뒤에 입력할 열의 이름을 명시해야 한다. 이 경우엔 생략한 나이 열에는 NULL 값이 들어간다.



- 열의 순서를 바꿔서 입력하고 싶을 땐 열 이름과 값을 원하는 순서에 맞춰 써주면 된다.



AUTO_INCREMENT : 자동으로 증가

AUTO_INCREMENT는 열을 정의할 때 1부터 증가하는 값을 입력해준다. INSERT에서는 해당 열이 없다고 생각하고 입력하면 된다. 단, **AUTO_INCREMENT로 지정하는 열은 꼭 PRIMARY KEY로 지정해준다.**

CREATE TABLE HONGONG2 (TOY_ID INT AUTO_INCREMENT PRIMARY KEY, TOY_NAME CHAR(4), AGE INT);

이렇게 하면 테이블 형식이 갖춰진다. 이제 테이블에 데이터를 입력한다. 자동 증가하는 부분은 NULL을 입력하면 된다.

```
9  CREATE TABLE HONGONG2 (TOY_ID INT AUTO_INCREMENT PRIMARY KEY, TOY_NAME CHAR(4), AGE INT);
10 INSERT INTO HONGONG2 VALUES (NULL, '보핍', 25);
11 INSERT INTO HONGONG2 VALUES (NULL, '슬링키', 22);
12 INSERT INTO HONGONG2 VALUES (NULL, '렉스', 21);
13 SELECT * FROM HONGONG2;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

TOY_ID	TOY_NAME	AGE
1	보핍	25
2	슬링키	22
3	렉스	21
NULL	NULL	NULL

계속 입력하다 보면 현재 어느 숫자까지 증가되었는지 확인할 필요가 있다. 그런 경우 다음과 같은 SQL을 사용한다.

SELECT LAST_INSERT_ID();

```
14 SELECT last_insert_id();
```

Result Grid | Filter Rows:

last_insert_id()
3

만약 AUTO_INCREMENT로 입력되는 **다음 값**을 100부터 시작하도록 변경하고 싶다면 다음과 같이 실행한다. ALTER TABLE 뒤에는 테이블 이름을 입력하고 자동 증가를 100부터 시작하기 위해 AUTO_INCREMENT를 100으로 지정한다.

ALTER TABLE 테이블명 AUTO_INCREMENT=시작하고 싶은 인덱스;

```
14 ALTER TABLE HONGONG2 AUTO_INCREMENT=100;
15 INSERT INTO HONGONG2 VALUES (NULL, '재남', 35);
16 SELECT * FROM HONGONG2;
```

Result Grid | Filter Rows: | Edit:

TOY_ID	TOY_NAME	AGE
1	보핍	25
2	슬링키	22
3	렉스	21
100	재남	35
NULL	NULL	NULL

💬 ALTER TABLE은 테이블을 변경하라는 의미이다. 테이블의 열 이름 변경,새로운 열 정의, 열 삭제 등의 작업을 한다. 5장에서 더 자세히 다룬다.

처음부터 입력되는 값을 1000으로 지정하고, 다음 값은 1003, 1006,... 으로 3씩 증가하도록 설정하려면 시스템 변수인 @@AUTO_INCREMENT_INCREMENT를 변경해야 한다. 테이블을 새로 만들고 자동 증가의 시작은 1000으로 설정한다. 그리고 증가값은 3으로 하기 위해 @@AUTO_INCREMENT_INCREMENT를 3으로 지정한다.

SET @@AUTO_INCREMENT_INCREMENT='증가값'

```sql
CREATE TABLE HONGONG3(TOY_ID INT AUTO_INCREMENT PRIMARY KEY, TOY_NAME CHAR(4), AGE INT);
ALTER TABLE HONGONG3 auto_increment=1000;
SET @@auto_increment_increment=3;

INSERT INTO HONGONG3 VALUES (NULL, '토마스', 20);
INSERT INTO HONGONG3 VALUES (NULL, '제임스', 23);
INSERT INTO HONGONG3 VALUES (NULL, '고든', 25);
SELECT * FROM HONGONG3;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

TOY_ID	TOY_NAME	AGE
1000	토마스	20
1003	제임스	23
1006	고든	25
NULL	NULL	NULL

💬 **시스템 변수**

시스템 변수란 MySQL에서 자체적으로 가지고 있는 설정값이 저장된 변수를 말한다. 주로 MySQL의 환경과 관련된 내용이 저장되어 있다. 시스템 변수는 앞에 @@가 붙는 것이 특징이며, 시스템 변수의 값을 확인하려면 **SELECT @@시스템변수**를 실행하면 된다. 만약 전체 시스템 변수의 종류를 알고 싶다면 **SHOW GLOBAL VARIABLES**를 실행하면 된다.

💬 **데이터 입력 줄**

위에서 데이터 3건을 입력하기 위해 3줄로 입력했다.

```sql
INSERT INTO HONGONG3 VALUES (NULL, '토마스', 20);
INSERT INTO HONGONG3 VALUES (NULL, '제임스', 23);
INSERT INTO HONGONG3 VALUES (NULL, '고든', 25);
```

이는 다음과 같이 1줄로 입력할 수 있다.

INSERT INTO 테이블_이름 VALUES (값1, 값2, ...),(값3, 값4, ...) ...;

INSERT INTO ~ SELECT : 다른 테이블의 데이터를 한번에 입력

많은 양의 데이터를 지금까지 했던 방식으로 직접 타이핑해서 입력하려면 오랜 시간이 걸린다. 다른 테이블에 이미 데이터가 입력되어 있다면 INSERT INTO ~ SELECT 문을 사용해 해당 테이블의 데이터를 가져와서 한번에 입력할 수 있다.

INSERT INTO 테이블_이름 (열_이름1, 열_이름2, ...) SELECT 문;

주의할 점은 SELECT 문의 열 개수는 INSERT할 테이블의 열 개수와 같아야 한다. 즉, SELECT의 열이 3개라면 INSERT될 테이블의 열도 3개여야 한다.

1. MySQL을 설치할 때 기본적으로 내장되어 있는 world 데이터베이스의 city 테이블의 개수를 앞에서 배운 **count( )**로 조회해본다.

```sql
USE WORLD;
SELECT COUNT(*) FROM CITY;
```

Result Grid | Filter Rows:

COUNT(*)
4079

2. city 테이블의 구조를 살펴본다. **DESC** 문으로 테이블 구조를 확인할 수 있다. DESC 문을 통해 CREATE TABLE을 어떻게 했는지 알 수 있다.

```
29 •  DESC CITY;
```

esult Grid | Filter Rows: | Export: | Wrap Cell

Field	Type	Null	Key	Default	Extra
ID	int	NO	PRI	NULL	auto_increment
Name	char(35)	NO			
CountryCode	char(3)	NO	MUL		
District	char(20)	NO			
Population	int	NO		0	

MUL이란 MULTIPLE의 줄임말, 해당 열이 unqiue 하지 않은 인덱스에 포함될 수 있으며 여러 행이 동일한 값을 가질 수 있는 열

3. 데이터도 LIMIT을 사용해서 5건 정도 살펴본다.

```
30 •  SELECT * FROM CITY LIMIT 5;
```

Result Grid | Filter Rows: | Edit:

ID	Name	CountryCode	District	Population
1	Kabul	AFG	Kabol	1780000
2	Qandahar	AFG	Qandahar	237500
3	Herat	AFG	Herat	186800
4	Mazar-e-Sharif	AFG	Balkh	127800
5	Amsterdam	NLD	Noord-Holland	731200

4. 이 중에서 도시이름(NAME)과 인구(POPULATION)을 가져온다. 먼저 테이블을 생성한다. 테이블은 DESC로 확인한 열 이름(Field)와 데이터 형식(Type)을 사용하면 된다.

   CREATE TABLE CITY_POP (CITY_NAME CHAR(35), POPULATION INT);

5. 이제 CITY 테이블의 내용을 CITY_POP 테이블에 입력한다.

   INSERT INTO CITY_POP SELECT NAME, POPULATION FROM CITY;

```
32 •  CREATE TABLE CITY_POP (CITY_NAME CHAR(35), POPULATION INT);
33 •  INSERT INTO CITY_POP SELECT NAME,POPULATION FROM CITY;
34 •  SELECT * FROM CITY_POP LIMIT 5;
```

esult Grid | Filter Rows: | Export: | Wrap Cell Content:

CITY_NAME	POPULATION
Kabul	1780000
Qandahar	237500
Herat	186800
Mazar-e-Sharif	127800
Amsterdam	731200

결과 메시지로 4079행이 잘 처리된 것으로 나온다.

```
4079 row(s) affected Records: 4079  Duplicates: 0  Warnings: 0
```

UPDATE : 데이터 수정

정보가 수정되어 행 데이터를 수정해야 하는 경우도 빈번하게 발생한다. 이럴 때 **UPDATE**문을 사용해서 내용을 수정한다.

UPDATE 테이블_이름 SET 열1=값1, 열2=값2, ... WHERE 조건;

MySQL Workbench 설정 변경

MySQL 워크벤치에서는 기본적으로 **UPDATE** 및 **DELETE**를 허용하지 않기 때문에 UPDATE를 실행하기 전에 설정을 변경해야 한다. 기존에 열린 쿼리 창을 모두 종료하고 **[Edit]→[Preference]**로 이동해서 Workbench Preference 창의 **[SQL Editor]**에서 'Safe Updates' (reject 어쩌구)를 체크 해제한 후 [OK] 버튼을 누른다.

설정을 적용하려면 프로그램을 종료하고 다시 실행해야 한다. 설정이 완료되었으면 다시 실습을 시작한다.

1. 앞에서 생성한 CITY_POP 테이블의 도시 이름(CITY_NAME)중에서 'Seoul'을 '서울'로 변경해본다. 새 쿼리 창을 열고 다음 SQL을 실행한다.

   UPDATE CITY_POP SET CITY_NAME='서울' WHERE CITY_NAME='Seoul';

```
5 •  UPDATE CITY_POP SET CITY_NAME='서울' WHERE CITY_NAME='Seoul';
6 •  SELECT * FROM CITY_POP WHERE CITY_NAME='서울';
```

esult Grid | Filter Rows: | Export: | Wrap Cell Content:

CITY_NAME	POPULATION
서울	9981619
서울	9981619

2. 필요하다면 한번에 여러 열의 값을 변경할 수 있다. 콤마로 분리해서 여러 개의 열을 변경하면 된다. 다음 SQL은 도시 이름인 New York'를 '뉴욕'으로 바꾸면서 동시에 인구는 0으로 설정하는 내용이다.

   UPDATE CITY_POP SET CITY_NAME='뉴욕',POPULATION=0 WHERE CITY_NAME='New York';

```
8 •  UPDATE CITY_POP SET CITY_NAME='뉴욕',POPULATION=0 WHERE CITY_NAME='NEW YORK';
9 •  SELECT * FROM CITY_POP WHERE CITY_NAME='뉴욕';
```

sult Grid | Filter Rows: | Export: | Wrap Cell Content:

CITY_NAME	POPULATION
뉴욕	0
뉴욕	0

WHERE가 없는 UPDATE 문

UPDATE 문은 사용법이 간단하지만 주의할 점이 있다. UPDATE 문에서 WHERE 절은 문법 상 생략이 가능하지만, **생략하면 테이블의 모든 행의 값이 변경된다.** 일반적으로 전체 행의 값을 변경하는 경우는 별로 없으므로 주의해야 한다.

UPDATE CITY_POP SET CITY_NAME='서울' → 해당 코드 사용x

만약 이 SQL을 사용했다면 4000개가 넘는 모든 도시 이름이 '서울'로 바뀐다. WHERE 절이 없기 때문에 도시 이름 열의 모든 값을 '서울'로 바꿔버린다.

전체 테이블의 내용은 변경하는 경우는 단위 변환 사용에 사용할 수 있다. 예를 들어 CITY_POP 테이블의 인구 열은 1명 단위로 데이터가 저장되어 있어 단위를 10000명 단위로 변경하면 읽기 쉬워질 것이다.

UPDATE CITY_POP SET POPULATION=POPULATION / 10000 ;

```
11 •  UPDATE CITY_POP SET POPULATION= POPULATION / 10000 ;
12 •  SELECT * FROM CITY_POP LIMIT 5;
```

esult Grid | Filter Rows: | Export: | Wrap Cell Con

CITY_NAME	POPULATION
Kabul	178
Qandahar	24
Herat	19
Mazar-e-Sharif	13
Amsterdam	73

DELETE : 데이터 삭제

테이블의 행 데이터를 삭제하고 싶을 땐 DELETE를 사용해서 행 데이터를 삭제할 수 있다. DELETE는 행 단위로 삭제하며, 형식은 다음과 같다.

DELETE FROM 테이블_명 WHERE 조건;

- CITY_POP 테이블에서 'NEW'로 시작하는 도시를 삭제하기 위해 다음과 같이 실행한다.

  DELETE FROM CITY_POP WHERE CITY_NAME LIKE 'NEW%';

  SQL 문을 실행하면 결과창에 삭제된 행 개수 affected 라고 나온다.

- 만약 NEW 로 시작하는 모든 도시를 지우는 것이 아니라, NEW 글자로 시작하는 도시 중 상위 몇 건만 삭제하려면 LIMIT 구문과 함께 사용하면 된다.

  DELETE FROM CITY_POP WHERE CITY_NAME LIKE 'NEW%' LIMIT 5;

  위에서 모두 지웠기 때문에 해당 SQL 문을 실행해도 변경되는 것은 없다.

UPDATE와 마찬가지로 DELETE 문을 WHERE 절 없이 사용하면 모든 행 데이터가 삭제되므로 주의해야 한다.

💬 대용량 데이터의 삭제

만약 몇억 건의 데이터가 있는 대용량의 테이블이 더 이상 필요 없다면 어떻게 삭제할까?

먼저 다음 SQL문을 통해 대용량 테이블 3개를 준비한다. 데이터는 모두 동일하며 각 테이블 당 44만건의 행 데이터가 존재한다.

CREATE TABLE BIG_TABLE1 (SELECT * FROM WORLD.CITY, SAKILA.COUNTRY);

CREATE TABLE BIG_TABLE2 (SELECT * FROM WORLD.CITY, SAKILA.COUNTRY);

CREATE TABLE BIG_TABLE3 (SELECT * FROM WORLD.CITY, SAKILA.COUNTRY);

SELECT COUNT(*) FROM BIG_TABLE1;

(WORLD, SAKILA 모두 MySQL 설치 시, 기본적으로 내장되어 있는 데이터베이스이다.)

이제 동일한 내용의 대용량 테이블 3개를 **DELETE, DROP, TRUNCATE** 각각 다른 방법으로 삭제한다.

26	22:36:23	DELETE FROM BIG_TABLE1	444611 row(s) affected	5.172 sec
27	22:36:31	DROP TABLE BIG_TABLE2	0 row(s) affected	0.031 sec
28	22:36:33	TRUNCATE TABLE BIG_TABLE3	0 row(s) affected	0.047 sec

실행 시간

SQL 문	설명	속도
DELETE	테이블 내 데이터 삭제, WHERE 문 사용 가능	느림
DROP	테이블 자체를 삭제	빠름
TRUNCATE	테이블 내 데이터 삭제, WHERE 문 사용 x	빠름

결론적으로 대용량 테이블의 전체 내용을 삭제할 때 테이블 자제가 필요 없을 경우에는 **DROP**으로 삭제하고, 테이블의 구조는 남겨 놓고 싶다면 **TRUNCATE**로 삭제하는 것이 효율적이다.