

## Head First SQL

노트북: DataBase

만든 날짜: 2019-12-11 오후 10:31

수정한 날짜: 2020-01-27 오후 11:21

작성자: yoonjeong\_choi@tmax.co.kr

태그: Head First, SQL

URL: <https://untitledtblog.tistory.com/131>

---

## [ 주요 SQL 쿼리 ]

- CREATE
- USE
- DESC(or DESCRIBE) table\_name
- INSERT INTO table\_name (field1,field2,...)
- VALUE (value1, value2,...)
- SELECT (field1, field2,...) FROM
- WHERE
- IN
  - field IN (value1, value2,...)
  - field NOT IN (value1, value2,...)
- BETWEEN
  - field\_name BETWEEN a AND b
- DELETE FROM
- UPDATE
- SHOW
  - CREATE DATABASE db\_name
  - CREATE TABLE table\_name
  - COLUMNS FROM table\_name : 해당 테이블 모든 열
  - INDEX FROM table\_name : 인덱싱 되어 있는 열과 무슨 테이블의 인덱스를 가지는지 표시 (ex. PRIMARY KEY)
  - WARNINGS : SQL 쿼리문이 경고 메시지를 표시한다면 경고 메시지가 출력
- PRIMARY KEY(field\_name)
- ALTER TABLE table\_name
  - CHANGE COLUMN field\_name
  - MODIFY COLUMN field\_name
  - ADD COLUMN field\_name data\_type
  - DROP COLUMN field\_name
  - RENAME TO hope\_table\_name
- 문자열 관련 함수
  - 쿼리의 결과로 변경된 문자열을 반환할 뿐 레코드를 변경하지 않음
  - RDMS 소프트웨어 따라 함수가 다를 수 있음
  - SUBSTRING\_INDEX(field\_name, 구분자, 구분자의 위치)
  - SUBSTRING(str, start\_pos, length) : start\_pos에 있는 문자에서부터 시작해서 길이가 length인 str의 일부 반환

- UPPER(str), LOWER(str) : 문자열 str에 대해서 모두 대문자/소문자로 각각 반환
- REVERSE(str) : 문자열 str의 순서를 역순으로 반환
- LTRIM(str), RTRIM(str) : 문자열 str의 앞(왼쪽)/뒤(오른쪽)에 있는 공백 문자들을 제거한 문자열 반환
- LENGTH(str) : 문자열 str의 문자수 반환
- SUBSTR(field\_name, start) : field\_name에 있는 문자열에서 start번째부터 시작하는 부분 문자열 반환
- CASE ~ END
  - CASE WHEN ~ THEN ~ ELSE~ END;
- ORDER BY
  - ASC, DECS
- GROUP BY
  - COUNT, AVG, MAX, MIN
- DISTINT
- LIMIT
- FOREIGN KEY(field\_name) REFERENCES table\_name (FK\_field\_name)
- JOIN
  - AS
  - CROSS JOIN another table
  - INNER JOIN another table ON conditions
  - NATURAL JOIN another table
- OUTER JOIN
  - LEFT OUTER JOIN
  - RIGHT OUTER JOIN
- UNION
- UNION ALL
- CREATE TABLE AS
- INTERSECTION
- EXCEPT
- CHECK (coditions)
- CREATE VIEW view\_name AS
  - SELECT \* FROM view\_name
- START TRANSACTION;
  - COMMIT, ROLLBACK
- CREATE USER
- GRANT
  - GRANT ~ ON ~ TO ~ WITH GRANT OPTION
- REVOKE
  - REVOKE GRANT OPTION ON ~ ON ~ FROM ~
  - CASCADE, RESTRICT
- ROLE
  - WITH ADMIN OPTION

## [ 데이터베이스 정규화 법칙 ]

- 제 1 정규형(1NF)

- 규칙 1 : 열의 데이터들은 원자적 값을 가져야 한다
  - **atomic(원자적)** : 공간 절약을 위해 데이터가 충분히 쪼개 졌다
  - 원자적 데이터 규칙 1 : 원자적 데이터로 구성된 열은 그 열에 같은 타입의 데이터를 여러개 가질 수 없다
  - 원자적 데이터 규칙 2 : 원자적 데이터로 구성된 테이블은 같은 타입의 데이터를 여러 열에 가질 수 없다
  - 열의 데이터들은 서로 독립적이어야 한다
- 규칙 2 : 같은 데이터가 여러 열에 중복되지 않는다
- 각 행은 유일무이한 식별자인 **기본키(primary key)**를 가지고 있어야 한다
- 즉, 완전히 정규화하려면 각 레코드에 기본키를 부여 해야 한다
- 제 2 정규형(2NF)
  - 기본키와 기본키가 아닌 열들과의 관계
  - 규칙 1 : 1NF여야 한다
    - 기본키를 제외한 열들은 독립적이어야 한다
  - 규칙 2 : 부분적 함수 의존이 없어야 한다
    - 기본키가 아닌 열들은 기본키에 종속적이어야 한다(by 1NF)
    - 즉, 에러 발생 가능성을 줄이기 위해서는 부분적 함수 종속 관계가 없어야 한다
- 제 3 정규형(3NF)
  - 키가 아닌 열들의 관계
  - 규칙 1 : 2NF여야 한다
  - 규칙 2 : 이행적 종속이 없어야 한다
    - 키가 아닌 열들끼리는 독립적이어야 한다

## [ 테이블 생성 및 삭제 ]

- Contents(SQL 도구 상자)
  - CREATE DATABASE db\_name : 데이터들을 보관할 데이터 베이스 생성
  - USE DATABASE db\_name : 테이블을 만들고 조작하기 위한 데이터베이스로 들어가기 위한 명령어
  - CREATE TABLE table\_name : 해당 DB에서 테이블 셋팅을 시작
    - 테이블에 넣은 테이블의 종류를 분석하여 필드의 이름 및 데이터 타입을 설정
  - NULL vs NOT NULL
  - DEFAULT : 열에 대한 디폴트 값 설정
  - DROP TABLE table\_name : 해당 테이블 삭제
- CREATE DATABASE db\_name : 데이터 베이스 생성

- `USE db_name` : 해당 데이터베이스 선택
- `CREATE TABLE table_name` : 해당 데이터베이스에 테이블 생성
- 저장 관련 변수들
  - `CHAR`, `CHARACTER` : 길이가 고정된 문자열
  - `VARCHAR` : 길이 255까지의 문자 데이터
  - `INT`, `INTEGER` : 정수
  - `DEC(자연수 지수, 소수 자릿수)` : 10진수 소수
  - `DATE` : 시간 없는 날짜
  - `DATETIME` : 시간 및 날짜
  - `BLOB` : 큰 덩어리의 문자 데이터
- 문자열은 반드시 작은 따옴표(')를 이용한다!!
- `DESC table_name` : 해당 테이블의 각 field의 이름과 저장 변수 타입을 보여준다
- `INSERT INTO table_name (field1,field2,...)VALUES (value1, value2,...)` : values들을 해당 테이블에 저장
  - field와 value의 순서가 일치해야 한다
  - `INSERT INTO table_name VALUES (value1, value2,...)`
    - 해당 테이블의 field 순서를 아는 경우 전체 values를 저장

## [ SELECT : 원하는 데이터를 선택하는 쿼리 ]

- Contents(SQL 도구 상자)
  - `SELECT *` : 테이블의 모든 열을 선택하는 키워드
- `SELECT (field1,field2,...) FROM table_name` : 해당 테이블에서 (field1,field2,...) 가져 온다
- `*` : 모든 fields 선택
- `/'` : 문자열에 '가 들어간 경우 이스케이프 문자를 사용해야 한다

## [ WHERE : 조건문 ]

- Contents(SQL 도구 상자)
  - `=`, `<>`, `<`, `>`, `<=`, `>=` : 부등호 연산자
  - `IS NULL` : NULL 값인지 확인
  - `AND` and `OR` : 조건문들을 결합하기 위해 사용
  - `NOT` : 조건문 결과의 반대값 반환
  - `BETWEEN` : 일정 범위의 값을 찾을 수 있는 키워드
  - `LIKE` : 와일드 카드를 이용하기 위한 키워드
    - `%` : 다수 불특정 문자열과 일치
    - `_` : 하나의 불특정 문자열과 일치

- AND, OR
- NOT : !조건문
- 비교 연산자 : =, <>, <, >, <=, >=
- BETWEEN field < a AND field
- IS NULL : NULL 데이터 인지 체크
- LIKE : 와일드카드 패턴을 이용
  - 열의 정보가 복잡하면 원하는 특정 데이터를 나타내는 것이 쉽지 않음.
  - % : 다수 불특정 문자열과 일치
  - \_ : 하나의 불특정 문자와 일치
- field BETWEEN a AND b : a < field < b 인지 체크
- IN : 여러번 OR 키워드(field=value)를 사용해야 할 때, 간단하게 해주는 키워드
  - field IN (value1, value2, ...) : field 가 values 중 하나와 일치
  - field NOT IN (value1, value2, ..) : field가 values과 모두 일치하지 않음

## [ DELETE와 UPDATE : 데이터 변경 및 삭제 ]

- Contents(SQL 도구 상자)
  - DELETE : 특정 데이터를 삭제하기 위한 키워드
  - UPDATE : 특정 데이터들의 열 값을 새로운 값으로 변경
  - SET : UPDATE와 같이 기존 열의 값을 변경(산술적 함수를 이용하여)
- DELETE FROM table\_name WHERE ... : 해당 테이블에서 조건에 맞는 행들을 삭제
- DELETE는 SELECT와 다르게 반환값이 없다
- UPDATE table\_name SET field1=value1, field2=value2,... WHERE ... : 해당 테이블에서 조건에 맞는 행들의 fields를 values로 변경
- 단, UPDATE는 INSERT&DELETE을 통해 예전 레코드를 재사용하여 새로운 레코드를 만든다
- SET field = function(field) : function에 대한 field의 값으로 업데이트
  - 숫자 열에 대해서는 기본적인 수학 연산(사칙연산+\alpha) 수행 가능
  - 문자 연산 또한 가능 ex) UPPER(), LOWER()
  - ex) SET cost = cost + 1 : 가격을 1씩 올려준다
- 단, 테이블의 설계가 잘 되어 있으면 전반적으로 변경할 일이 적다. 즉, 잘 설계된 테이블은 테이블의 데이터에 집중 할 수 있다.

## [ 좋은 테이블 설계 : 테이블의 정규화 ]

- Contents(SQL 도구 상자)
  - 원자적 데이터
    - 원자적 데이터 규칙 1 : 원자적 데이터로 구성된 열은 그 열에 같은 타입의 데이터를 여러개 가질 수 없다
    - 원자적 데이터 규칙 2 : 원자적 데이터로 구성된 테이블은 같은 타입의 데이터를 여러 열에 가질 수 없다
  - SHOW CREATE TABLE : 존재하는 테이블을 생성하기 위한 명령어를 알아 보기 위해 사용
  - 기본키 : 테이블에서 행을 유일무이하게 식별하는 열 하나 or 열의 집합
  - AUTO\_INCREMENT : 열 선언시에 사용하면 그 열은 데이터가 삽입될 때마다 자동으로 유일무이(unique)한 정수값이 할당
  - 제 1 정규형(1NF) : 각 열의 데이터가 원자적인 값이고, 기본키를 가지고 있어야 한다
- 기본적인 idea
  - 쿼리는 간단할수록 좋음
  - 처음부터 가능한 쿼리를 간단하게 만드는 것이 추후에 좋다
  - 데이터의 중복은 데이터베이스의 용량을 더 차지하게 만든다.
    - 데이터의 중복이 필요한 경우, DB의 용량을 고려해야 한다
    - ex)전체 주소, 각 주소(시, 동)
- SQL은 관계형 데이터베이스 관리시스템(RDBMS)를 위한 언어
  - 관계형 : 테이블의 설계에서 정보를 나타낼 때, 여러 테이블 사이의 **열들이 서로 어떠한 관계가 있는지** 고려해야 한다는 의미
- 테이블 생성을 할 때 대략적인 과정
  - 1. 테이블로 표현하려는 것을 선택 i.e 테이블에 나타내는 것이 무엇인가?
  - 2. 그 테이블을 사용하여 얻어야 하는 정보들의 리스트 작성 i.e 테이블을 **어떻게 사용** 할 것인가?
  - 3. 해당 리스트를 이용하여 테이블을 만들 **정보들의 조각으로** 나눈다 i.e 어떻게 하면 가장 쉽게 쿼리를 보낼 수 있을까?
- **atomic(원자적)** : 공간 절약을 위해 데이터가 충분히 쪼개 졌다
  - 데이터가 원자적이다 <=> **쪼갤 수 없는 가장 작은 조각으로** 쪼개졌다

- 가능한만큼 작은 조각으로 쪼개는 것이 아님!!
- 효율적인 테이블을 만드는데 필요한 **한도** 내에서 가능하면 작은 조각으로 나눠야 한다
- 즉, 필요 이상으로 데이터를 나누지 말아야 한다
- 원자적 데이터는 테이블 내의 데이터를 정확히 나타내는데 도움이 된다 => 쿼리 수행이 효율적!!
- 예제
  - 배달원 : 전체 주소만 필요! 주소가 나눠져있으면 쿼리가 복잡해진다
  - 부동산 : 자세한 주소들에 대한 필드가 필요. 특정 주소(시,구,동)에 있는 매물을 팔아야 하기 때문

- 테이블 설계를 위한 질문들

- 테이블에 **표현하는 것**이 무엇인가?
- 그것을 **얻기**위해 테이블을 **어떻게** 사용할 것인가?
- 짧고 명료한 쿼리를 위해 각 **열**들이 **원자적 데이터**를 가지고 있는가?

- 정규화 : 테이블들이 표준 규칙을 따르게 한다는 뜻

- 정규화의 이점
  - 중복 데이터가 없기 때문에 DB의 크기를 줄일 수 있다.
  - 찾아야 할 데이터가 적어 쿼리가 효율적이다

- 표준 규칙 : **제 1 정규형 (1NF)**

- 각 행의 데이터들은 원자적 값을 가져야 한다.
- 각 행은 유일무이한 식별자인 **기본키(primary key)**를 가지고 있어야 한다
- 즉, 완전히 정규화하려면 각 레코드에 기본키를 부여 해야 한다
- **기본키 규칙**
  - 기본키는 각 레코드를 식별하는데 사용
    - 기본키의 열의 데이터는 중복될 수 없다는 의미
    - 다른 열들의 데이터는 중복 가능
    - ex) 기본키:회원고유번호 else:이름,취미
  - 기본키는 NULL이 될 수 없음
  - 레코드가 삽입될 때 값이 있어야 함
  - 간결해야 함
  - 기본키의 값은 변경 불가

- SHOW CREATE TABLE table\_name

- CREATE TABLE로 테이블을 만들 당시 어떻게 테이블을 생성했는지 보여준다

- 해당 테이블을 생성했던 쿼리문(코드)를 출력
  - 출력된 정보에서는 테이블 이름과 필드 이름 사이에 백틱(`, 숫자 1키 옆)이 추가되어 출력
    - 백틱은 예약어인 SQL 키워드를 열이름으로 사용할 때 사용 가능(좋은 방법은 아니다)
  - 마지막 라인인 ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_0900\_ai\_ci 은 데이터가 어떻게 저장되고 문자열은 무엇을 사용할지 표시
  - 해당 출력문을 복사/붙여넣기를 통해 해당 테이블과 같은 구조의 테이블을 쉽게 생성 가능
- PRIMARY KEY(field\_name) : 해당 필드를 기본키로 지정해주는 쿼리문
    - field\_name은 NOT NULL로 설정 해야 한다(by 기본키 규칙)
    - 기본키에 해당하는 필드는 테이블의 맨 앞 열에 두는 것이 좋다
    - field\_name INT NOT NULL : 사용자가 테이블에 데이터를 추가할 때 기본키가 중복되지 않게 해야 한다
    - AUTO\_INCREMENT
      - 사용 방법 : field\_name INT NOT NULL AUTO\_INCREMENT
      - SQL 소프트웨어가 해당 열을 1부터 1씩 증가시키며 자동으로 채운다
      - 데이터 저장시(INSERT) field\_name에 값을 NULL로 지정한다면, NULL이 무시되고, AUTO\_INCREMENT가 자동으로 수행
        - BUT!! AUTO\_INCREMENT가 없으면, 에러 발생
      - 데이터 저장시(INSERT) field\_name에 값을 지정한다면, AUTO\_INCREMENT가 해당 값으로 오버라이드 된다
  - ALTER TABLE table\_name ADD COLUMN primaryKey\_id INT NOT NULL AUTO\_INCREMENT FIRST, AND PRIMARY KEY (primaryKey\_id)
    - 정규화 되지 않은 테이블에 기본키를 추가하는 방법
    - 이미 있는 데이터들 또한 AUTO\_INCREMENT에 의해 기본키가 자동으로 생성

## [ ALTER : 과거 다시 쓰기 ]

- Contents(SQL 도구 상자)
  - ALTER : 기존 테이블의 데이터를 이용하면서 완벽하지 않은 기존 테이블을 정규화 하기 위한 명령어



- ADD : 테이블에서 원하는 위치에 열 추가
- DROP : 테이블에서 열을 삭제
- CHANGE : 기존 열의 이름과 데이터 타입을 모두 변경
- MODIFY : 기존 열의 데이터 타입만 변경
- 문자열 함수 : 쿼리에서 반환되는 문자열 타입인 열의 내용 수정 가능. 단, 원래의 열 값이 변경되는 것은 아니다

- ALTER TABLE table\_name

- 해당 테이블을 변경하기 위한 명령어
- 열 추가, 테이블 이름 변경 등의 동작을 위한 명령어
- 명령어 뒤에 여러 키워드 or 명령어를 통해 테이블 변경 가능

- ADD COLUMN field\_name data\_type : 해당 필드 이름을 열에 추가

- ADD COLUMN field\_name data\_type\* 를 통해 추가할 열의 위치를 지정 가능
  - FIRST, SECOND, THIRD : 해당 필드를 첫, 두, 세 번째 열에 추가
  - LAST : 해당 필드를 마지막 열에 추가
  - AFTER existing\_field : 해당 필드를 특정 필드 뒤에 추가
  - BEFORE existing\_field : 해당 필드를 특정 필드 앞에 추가
- ADD COLUMN field\_name FIRST : 해당 필드를 첫 번째 열에 추가
- ADD COLUMN field\_name AFTER field\_name1 : 해당 필드를 AFTER 뒤 필드 뒤에 추가

- 기존 테이블 변경을 위한 명령어

- CHANGE : 기존 열의 이름과 데이터 타입 변경
- MODIFY : 기존 열의 데이터 타입 or 위치 변경
- ADD : 테이블에 열 추가
- DROP : 테이블의 열 제거
- RENAME TO table\_name : 해당 이름으로 테이블 이름 변경
- 바꿀 데이터 타입이 예전 타입과 호환되지 않으면 에러 메시지가 발생
  - eg) VARCHAR, CHAR -> INT
- (주의!) 데이터 타입이 '호환'되면서 데이터가 유실될 가능성이 존재
  - eg) VARCHAR(10) -> VARCHAR(1)

- **CHANGE COLUMN field\_name(바꾸고자 하는 필드)**  
 field\_name(원하는 필드 이름) 기타 설정: ,를 사용하여 여러 CREATE를 사용하면 여러 개 열의 이름과 데이터 타입 변경 가능
  - 열의 이름과 데이터 타입을 바꾸고 싶을 때 사용
  - 변경할 컬럼의 순서는 ADD COLUMN과 같은 방식으로 변경 가능
    - (BUT!!) 특정 SQL 소프트웨어에서 제공하는 변경 방식
    - 이동시키려는 열의 값을 복사하고, 임시 테이블에 저장
    - 테이블에 변경 명령을 수행 후, 임시 테이블의 값을 업데이트 한 뒤, 임시 테이블 삭제
    - 이러한 수행들은 SQL 소프트웨어가 지원하지 않는 경우 SELECT를 사용하여 원하는 열의 위치에 삽입 가능
    - 가장 좋은 것은 테이블의 설계 시점에서 순서에 대해 고민하고 결정 하는 것!!
  - 데이터 손실 위험성
    - 바꿀 데이터 타입과 예전 데이터 타입이 호환되지 않는 경우에는 에러 메시지가 출력
    - (BUT!!) 호환되는 경우에, 데이터가 잘릴 가능성 존재
  - eg) **CHANGE COLUMN field\_name1 field\_name2**  
 data\_type : 필드1의 이름을 필드2로 바꾸고 해당 데이터 타입을 data\_type으로 변경
  - eg) **CHANGE COLUMN field\_name1 field\_name2**  
 data\_type NOT NULL AUTO\_INCREMENT, ADD PRIMARY KEY (field\_name2) : 필드1의 이름을 필드2로 바꾸고 해당 타입 데이터를 data\_type으로 변경한 뒤 primary key(기본 키)로 설정
- **MODIFY COLUMN field\_name**: 열의 데이터 타입만을 바꾸고 이름은 그대로 사용
  - 데이터 타입 '만' 바꾸고 싶을 때 사용
  - 컬럼의 순서는 ADD COLUMN와 같은 방식으로 변경 가능
    - (BUT!!) 특정 SQL 소프트웨어에서 제공하는 변경 방식
    - 이동시키려는 열의 값을 복사하고, 임시 테이블에 저장
    - 테이블에 변경 명령을 수행 후, 임시 테이블의 값을 업데이트 한 뒤, 임시 테이블 삭제
    - 가장 좋은 것은 테이블의 설계 시점에서 순서에 대해 고민하고 결정 하는 것!!
  - 데이터 손실 위험성
    - 바꿀 데이터 타입과 예전 데이터 타입이 호환되지 않는 경우에는 에러 메시지가 출력
    - (BUT!!) 호환되는 경우에, 데이터가 잘릴 가능성 존재
  - eg) **MODIFY COLUMN field\_name1 data\_type** : 필드1의 저장 데이터 타입을 data\_type으로 변경

- DROP COLUMN field\_name : 해당 필드 이름의 열 삭제
  - 필요없는 열을 삭제하기 위한 명령어
  - 필요없는 열이 DB 내에 있는 것은 공간 낭비
  - 테이블에 필요한 열만 있도록 하는 것은 좋은 프로그래밍 습관
  - 나중에 필요하면 ALTER&ADD를 이용하여 쉽게 추가 가능
  - 열을 제거하면 그 안에 저장된 모든 데이터 소실
  - DROP 기준 : 중요한 데이터 손실 < 필요없는 데이터가 조금 포함
  - ALTER TABLE table\_name DROP PRIMARY :
    - PRIMARY KEY를 제거 하고 싶을 때 사용
    - eg) ALTER TABLE table\_name CHANGE primary\_id primary\_id NOT NULL AUTO\_INCREMENT; : 해당 필드를 기본 키로 설정하고 싶은 경우
- ALTER TABLE을 SELECT와 UPDATE와 함께 사용하면 원시적이지 않아 문제가 있는 열을 정교하고 원자적인 열로 정제 가능
  - Exercise : 열을 두개의 열로 나눠 보자
  - Situation :
    - location 필드에는 "도시 이름, 주 이름의 약어" (eg.Las Vegas, NV) 형태로 데이터가 저장된다.
    - location 필드를 City name과 State로 분리하고 싶은 상황
  - Base : 두 개의 필드 추가
    - State 열 : ADD COLUMN state CHAR(2);
    - City 열 : ADD COLUMN city VARCHAR(20);
  - State 열 : 마지막 두 문자(주 이름의 약어) 추출
    - SELECT RIGHT(location, 2) FROM my\_contacts
    - 해당 열(문자열 레코드)로부터 정해진 수의 문자들을 선택
    - RIGHT(field\_name), length : 열의 오른쪽에서 시작 (LEFT도 같은 방식으로 사용)
      - field\_name : 사용할 열
      - length : 열의 오른쪽에서부터 몇 개의 문자들을 선택할지 나타내는 숫자
    - 결론 : UPDATE my\_contacts SET state=RIGHT(location, 2)
  - City 열 : 해당 데이터(문자열) 일부를 선택
    - SELECT SUBSTRING\_INDEX(location, ',', 1) FROM my\_contacts;
    - 해당 열(문자열 레코드)로부터 문자열의 일부를 선택
    - SUBSTRING\_INDEX : 특정 문자나 문자열 앞의 모든 문자열 반환
      - SUBSTRING\_INDEX(field\_name, 구분자, 구분자의 위치)
    - location : 열 이름

- ', ' : 구분자(SUBSTRING\_INDEX가 찾는 문자열)
- 1 : 1번째 구분자를 찾는다는 의미
  - 2 : 두번째 구분자를 찾고 그 앞의 모든 것을 반환
- 결론 : UPDATE my\_contacts SET city=SUBSTRING\_INDEX(location, ', ', 1);
- 문자열 함수들
  - 쿼리의 결과로 변경된 문자열을 반환할 뿐 레코드를 변경하지 않음
  - SUBSTRING(str, start\_pos, length) : start\_pos에 있는 문자에서부터 시작해서 길이가 length인 str의 일부 반환
  - UPPER(str), LOWER(str) : 문자열 str에 대해서 모두 대문자/소문자로 각각 반환
  - REVERSE(str) : 문자열 str의 순서를 역순으로 반환
  - LTRIM(str), RTRIM(str) : 문자열 str의 앞(왼쪽)/뒤(오른쪽)에 있는 공백 문자들을 제거한 문자열 반환
  - LENGTH(str) : 문자열 str의 문자수 반환

## [ 고급 SELECT ]

- Contents(SQL 도구 상자)
  - ORDER BY : 지정한 열값을 기준으로 결과를 알파벳 순서로 정렬
  - GROUP BY : 공통된 열값으로 행들을 그룹핑
  - DISTINCT : 중복 없이 유일한 값을 한번만 반환
  - LIMIT : 반환할 행의 갯수 제한
  - 함수들
    - SUM
    - COUNT
    - AVG, MAX, MIN
- Background : 고급 SELECT문이 사용되는 상황
  - 비디오 가게에서 영화를 카테고리별로 라벨을 달아 분류하려는 상황
  - 현재는 영화의 타입이 각 장르별로 해당 열에 T/F로 저장
  - 현재 영화의 타입들을 이용하여 category 필드를 추가하려는 상황
  - category 필드를 추가하는데 생기는 조건 및 문제점
    - 영화는 언제나 한 카테고리에 속해야 한다
    - 각 장르 field의 T/F 값에 우선순위가 없음(\*\*\*)
    - 장르 field 열들을 사용해서 카테고리를 정한다

- UPDATE를 통해서 category 열의 레코드를 변경
  - 해당 field의 상태값에 따라 category를 update할 수 있음 i.e UPDATE movie\_table SET category = '해당 카테고리' WHERE 영화 타입 필드 = 'T'
  - 각 field의 T/F 값에 우선순위가 없음(\*\*\*) 때문에 영화의 타입에 대한 field의 순서에 따라 category의 값이 다양
  - WHERE문이 2개 이상이면 같은 열값을 두번 이상 변경
- CASE : 기존 열의 값과 조건을 비교하여 UPDATE 문을 합칠 수 있음
  - CASE ~ END; : 기본 문법
  - Example : UPDATE my\_table SET new\_column CASE WHEN field\_name1 = value1 THEN newValue1 ELSE newValue2 END;
    - field\_name1인 필드에서 value1인 레코드인 열에 대해서 new\_column인 필드에 newValue1을 저장
    - WHEN절을 여러 개 사용하여 다양한 조건 설정 가능
    - WHEN절을 만족하지 않으면 ELSE 절 뒤에 나오는 newValue2로 저장
    - END : CASE 조건문을 끝마치는 쿼리
  - 기존 열의 값을 조건과 비교하여 UPDATE문에 결합
  - CASE WHEN 뒤에 field\_name의 순서에 따라 SET쿼리문이 달라진다. 즉, CASE WHEN 뒤 field\_name의 순서를 통해 각 field\_name의 우선 순위 결정 가능
  - WHEN 뒤 조건이 맞는 경우 다른 WHEN절들은 무시하고 END로 건너뛰어 쿼리 종료
  - 프로그래밍 언어들의 Switch-Case문과 비슷한 기능
    - 즉, switch case(조건) default 원하는 쿼리 <=> CASE (WHEN 조건 THEN 원하는 쿼리).. ELSE ...END
  - ELSE가 없고 모든 WHEN 조건을 맞지 않으면 아무것도 UPDATE되지 않음

```
UPDATE movie_table
SET category =
CASE
```

```
  WHEN drama = 'T' THEN 'drama' ←
  WHEN comedy = 'T' THEN 'comedy'
  WHEN action = 'T' THEN 'action'
  WHEN gore = 'T' THEN 'horror'
  WHEN scifi = 'T' THEN 'scifi'
  WHEN for_kids = 'T' THEN 'family'
  WHEN cartoon = 'T' THEN 'family'
  ELSE 'misc'
```

```
◦ END;
```

- ORDER BY
  - SELECT문 뒤에 사용하여 SELECT문이 반환하는 데이터를 테이블의 열에 따라 알파벳 순서대로 배열
  - 순서는 데이터베이스 시스템에 따라 다름
    - NULL(공백)>특수문자>숫자>대문자>소문자
    - 특수문자 : !"#\$% '()\*+,-./
  - ORDER BY field\_name1, field\_name2; : 필드1로 정렬한 뒤 필드2로 정렬
  - 정렬 순서는 기본값으로 오름차순(ASCENDING)
    - ASC : 디폴트 값으로 열 이름 뒤에 사용 가능
    - DESC : 열 이름 뒤에 사용하여 결과의 순서를 역순으로 만든다
- SUM(field\_name) : 해당 필드의 값을 모두 더한 값 반환
  - ex) SELECT SUM(field\_name) WHERE ....
- GROUP BY field\_name : 해당 필드에서 같은 값을 같은 행끼리 그룹핑
- GROUP BY와 함께 사용할 수 있는 함수
  - AVG(field) : 해당 그룹에 있는 필드의 평균값 반환
  - MIN(field),MAX(field) : 해당 그룹에 있는 필드의 최대/최소 값 반환
  - COUNT(field\_name) : 해당 그룹에 있는 필드에 대한 열의 갯수
  - 이 함수들은 NULL값은 연산에 포함시키지 않는다
- DISTINCT field\_name : 해당 필드에 대해 중복없는 레코드 반환
  - 한 필드에 같은 중복된 레코드가 많은 경우 사용
- LIMIT : 결과에서 몇 개의 행을 반환할지 정하는 키워드
  - LIMIT(number) : 처음부터 number개의 행을 반환
  - LIMIT(start, number) : start에서 시작하여 number개의 행을 반환
  - ex) SELECT field\_name FROM table\_name LIMIT 2;

- Contents(SQL 도구 상자)
  - 스키마 : 관련 객체와 객체들 사이의 연결 & 데이터베이스의 데이터에 대한 설명
  - 스키마 설계에서의 데이터 패턴
    - 일대일 관계 : 부모 테이블의 한 행이 자식 테이블의 한 행과 연결된 관계
    - 일대다 관계 : 첫번째 테이블 A의 한 행이 두번째 테이블 B의 여러 행과 연결되지만, 두번째 테이블 B는 첫번째 테이블 A의 한 행과 연결되는 관계
    - 다대다 관계 : 두 테이블이 연결 테이블로 연결되어 첫번째 테이블 A의 여러 행이 두번째 테이블 B의 여러 행과 연결되고, 두번째 테이블 B의 여러행이 첫번째 테이블 A의 여러 행과 연결된 관계
  - 참조키 : 테이블에서 다른 테이블의 기본키를 참조하는 열
  - 합성키 : 여러 열로 구성된 기본키로 유일무이한 값
  - 부분적 함수 종속 : 키를 구 구성 하는 열 중에 키가 아닌 열과 관련이 없는 경우(cf. 2NF)
  - 이행적 함수 종속 : 키가 아닌 열이 키가 아닌 다른 열과 관련(cf. 3NF)
  - 정규화 법칙
    - 제 1 정규형 : 열이 원자적 값만을 포함하고 열에 데이터가 반복되지 않는 형태
    - 제 2 정규형 : 테이블이 1NF여야 하고 부분적 함수 종속이 없는 형태
    - 제 3 정규형 : 테이블이 2NF여야 하고, 이행적 종속 관계가 없는 형태
- 스키마 : 데이터베이스 내의 데이터( 열들과 테이블들 ) 그리고 데이터들 사이의 연결 방식에 대한 표현
  - 테이블을 그림으로 표현하면, 테이블의 "설계"와 테이블 안의 "데이터"를 분리하여 생각 가능
- 원자적이지 않은 필드를 가진 테이블은 복잡한 쿼리를 사용해야한다
- 테이블 추가 및 연결
  - 테이블에서 원자적이지 않은 필드를 새로운 테이블을 추가하고 옮긴다
  - 테이블을 연결하는 유일한 열 : 기본키
  - 참조키(foreign key) : 테이블의 한 필드로 다른 테이블( 연결한 테이블 )의 기본키

## • 참조키

- 한 테이블의 열들이 다른 테이블의 열과 연결하도록 하는데 사용
  - 연결되는 기본키와 다른 이름일 수도 있음
  - 참조하는 기본키는 **부모키**, 부모키가 있는 테이블을 **부모테이블**이라 한다
  - 참조키의 값은 NULL일 수도 있고 유일할 필요가 없다
- FOREIGN KEY(field\_name) REFERENCES table\_name (FK\_field\_name): field\_name 필드에 해당 테이블의 FK\_field\_name 필드를 참조키로 설정
- 참조키의 제약조건
    - 참조키를 테이블 안의 제약조건으로 생성하면 실수로 테이블 규칙을 위반하는 것을 방지
    - **참조 무결성(referential integrity)** : 참조키의 값으로 는 부모 테이블에 존재하는 키의 값만을 넣을 수 있음
    - 참조키의 값이 부모 테이블의 기본키일 필요는 없지만 유일해야 한다
- CONSTRAINT
    - 테이블에 잘못된 데이터의 입력을 막기 위해 일정한 규칙을 지정
    - 데이터의 정확성과 일관성을 보장하기 위해 각 필드에 정의하는 규칙
    - 사용법 : CONSTRAINT 제약조건명(참조테이블명\_키의필드이름\_키) 제약 조건
    - ex) CONSTRAINT my\_contacts\_contact\_id\_fk  
 FOREIGN KEY(contact\_id)  
 REFERENCES my\_contacts (contact id)
      - 1 line : 참조키가 어느 테이블(my\_contacts)을 참조하고 키의 이름(contact\_id)은 참조키로 제약하는 조건
      - 2,3 lines: 참조키는 my\_contacts 테이블의 contact\_id 필드로 지정하며, 참조키로 현재 테이블의 contact\_id 필드로 지정.
- 참조키 제약 조건 이유
    - 제약조건을 사용함으로써 부모 테이블에 존재하는 값만 저장 가능 => 두 테이블 간의 연결을 강제
    - 즉, 기본키의 값이 다른 테이블의 참조키 제약조건이면 기본키 테이블에서 하나의 행을 지우거나 변경하는 경우 에러 발생



- 고아(orphan) 행 : 참조하는 키가 있는 부모 테이블의 행을 삭제 한 경우의 자식 테이블의 해당 행
  - 고아 행은 계속해서 축적
  - 고아 행은 관심없는 레코드 => 필요없는 정보를 검색하게 하여 쿼리 속도 저하
- 스키마 설계에서의 데이터 패턴
  - 일대일 패턴
  - 일대다 패턴
  - 다대다 패턴
- 데이터 패턴 1 : 일대일 패턴
  - 테이블 A의 레코드가 테이블 B에 많아야 1개의 레코드와 연결
  - 즉, 부모 테이블 하나의 행은 자식 테이블 하나의 행과 관련
  - 일반적으로 일대일 관계의 데이터는 주 테이블 하나에 함께 사용 하는 것이 효율적이다 i.e) 주 테이블 필드에 추가하는 것
  - 테이블을 추가하여 일대일 패턴을 만들어 효율성을 향상시키는 경우
    - 열을 분리하였을때 데이터 조회 속도가 향상되는 경우
    - 아직 모르는 값을 포함하는 열이 있어도 주 테이블에서는 NULL값이 저장되는 것을 피하고 싶은 경우
    - 데이터의 일부를 일부러 접근하기 어렵게 만들고 싶은 경우
    - BLOB 타입처럼 양이 큰 데이터를 따로 저장하고 싶은 경우
- 데이터 패턴 2 : 일대다 패턴
  - 테이블 A의 한 레코드에 테이블 B의 레코드가 여러개 연결. 테이블 B의 레코드는 테이블 A의 한 레코드에만 연결.
  - 즉, 부모 테이블 하나의 행은 자식 테이블 다수의 행과 관련
  - 해당 레코드(테이블 A or 부모)는 여러 개의 레코드와 연결 => 기본키가 될 수 없음
  - 예시 : 지도교수-대학원생들
- 데이터 패턴 3 : 다대다 패턴
  - 테이블 A의 여러 레코드와 테이블 B의 여러 레코드가 연결
  - 예시 : 소비자-마음에 드는 상품 (여자들-신발들)
  - 테이블 A의 기본키를 테이블 B의 참조키로 설정하는 경우 중복 데이터가 존재
  - 다대다 관계의 테이블에는 중간에 테이블을 하나 추가함으로써 일대다의 관계로 단순화 가능
  - **연결 테이블(junction table)** : 일대다의 관계로 단순화하기 위해서 연결하려는 두 테이블의 기본키들을 열로 가지는 테이블
  - 두 개의 일대다 관계의 이러우지고 사이에 연결 테이블이 존재
  - 다대다 패턴의 경우 "반드시" 연결 테이블이 존재

- (why?) 그렇지 않으면 제 1 정규형(1NF) 위반
  - 즉, 중복된 데이터로 인한 쿼리 부하 감소
  - 테이블 조인(JOIN)을 사요하면 데이터의 무결성(integrity) 유지에 도움
  - 데이터를 업데이트하는 경우에도 이점
- **합성키** : 여러 개의 열들로 구성되어 유일무이한 키를 만드는 기본 키
  - 합성키가 유일무이하려면 구성하는 열들에 같은 레코드를 갖는 행이 없어야 한다
  - 합성키를 만드는 이유
    - 합성키를 통해 해당 테이블의 자연적인 키(자연키, natural key), 즉 비즈니스 모델에서 자연스레 나오는 속성을 통해 기본키를 만들 수 있음
    - 단일 기본키는 사용자에게 의해 만들어진 인위적인 키(인조키, synthetic key)
    - 자연키와 인조키의 사용에 대한 논쟁 존재
    - BUT! 각각의 비즈니스에 따라 다르지만 기본키의 불변성 때문에 인조키를 사용하는 것을 권장
- 한 테이블 내의 열들이 서로 어떻게 연결되는지 이해하는 것이 제 2 정규형과 제 3 정규형을 이해하는 핵심
  - A열의 데이터가 변경될 때 다른 B열의 데이터가 변경되어야 한다면, 변경되어야 하는 열 B는 변경되는 열A에 **함수적으로 종속**하고 있다고 말한다
    - (Notation)  $T.x \rightarrow T.y$  : 테이블 T에서 열 y는 열 x에 함수적으로 종속된다
  - **종속되는 열** : 다른 열이 변하면 변경되는 데이터를 포함하는 열
  - **독립적인 열** : 다른 열에 종속되지 않는 열
  - **부분적 함수 종속 관계** : 합성키를 구성하는 어떤 열 A에 대해서 종속적이지 않은 열 B가 존재할 때 열 B가 열 A에 부분적 함수 종속적이라고 말한다
    - 부분적 함수 종속 관계를 피하는 간단한 방법 : 기본키로써 id필드를 추가하여 테이블의 인덱스 용으로 사용
  - **이행적 함수 종속(transitive functional dependency)**
    - 키가 아닌 열 A가 변경 되었을 때 다른 열 B의 변경을 초래할때 해당 열 B가 열 A에 이행적 함수 종속된다고 말한다
    - 즉, 키가 아닌 열이 키가 아닌 다른 열과 관련되는 경우를 뜻

- 표준 규칙 : 제 2 정규형 (2NF)

- 테이블의 기본키와 데이터와 어떠한 관계에 있는지 초점  
i.e 기본키와 기본키가 아닌 열들과의 관계
  - 배경
    - 어떤 열 A에 대해 종속적인 열들이 존재하면 열 A의 데이터를 변경할 때 종속적인 열들에 대한 데이터 또한 변경을 해야한다
    - 여러 데이터가 변경될수록 에러가 발생할 가능성이 높음
    - 기본키가 아닌 열들은 기본키에 종속적이어야 한다(by 1NF)
    - 즉, 에러 발생 가능성을 줄이기 위해서는 부분적 함수 종속 관계가 없어야 한다
  - 규칙 1 : 1NF여야 한다
  - 규칙 2 : 부분적 함수 의존이 없어야 한다
    - 기본키가 아닌 열들은 기본키에 종속적이어야 한다(by 1NF)
    - 즉, 에러 발생 가능성을 줄이기 위해서는 부분적 함수 종속 관계가 없어야 한다
  - 1NF 테이블이 2NF인 조건들(OR)
    - 하나의 열로 된 기본키를 갖는다
    - 모든 열이 기본키의 일부(종속)
  - "인위적"으로 만든 기본키가 있고, 합성키가 없으면 2NF이다
- 
- 표준 규칙 : 제 3 정규형 (3NF)
    - 규칙 1 : 2NF여야 한다
    - 규칙 2 : 이행적 종속이 없어야 한다
      - 키가 아닌 열들끼리는 독립적이어야 한다

## [ 조인과 다중 테이블 연산 ]

- Contents(SQL 도구 상자)
  - **INNER JOIN(내부 조인)**
    - 조건을 사용해서 두 테이블의 레코드를 결합하는 모든 조인
    - CROSS JOIN의 결과 중 쿼리 조건에 맞는 행들을 반환
  - NATURAL JOIN(자연 조인) : "ON"절이 없는 내부 조인. 같은 열 이름을 가진 두 테이블을 조인할 때만 동작
  - CROSS(CARTESIAN) JOIN(교차 조인) : 한 테이블의 모든 행과 다른 테이블의 모든 행이 연결되는 모든 경우를 반환
  - EQUI JOIN(동등 조인) : 같은 행들을 반환
  - NON-EQUI JOIN(비동등 조인) : 같지 않은 행들을 반환

- COMMA JOIN : 콤마가 CORSS JOIN 키워드 대신 사용된다는 점을 제외하면 크로스 조인이 동일

- SUBSTR : 문자열을 추출하여 일부를 반환하는 문자열 함수
  - SUBSTR(field\_name, start) : field\_name에 있는 문자열에서 start번째부터 시작하는 부분 문자열 반환
  - RDMS 소프트웨어 따라 차이가 있을 수 있음

- 데이터를 증가하면 쿼리별 수행 속도 저하

- 매우 큰 데이터에 대한 쿼리 최적화 방법 : 해당 데이터에 대한 테이블 추가
  - 상황

- 직업(profession) 필드에는 특정 갯수의 직업 레코드가 존재
  - 직업 필드를 새로운 테이블로 분할

profession
prof_id 0
profession

- METHOD 1 : CREATE, SELECT, INSERT 사용
  - 열에 대한 테이블을 생성한 후 각 열들의 데이터를 SELECT를 이용하여 저장
  - SELECT...GROUP BY를 이용하여 중복없는 값들을 얻는다
  - 중복없는 값들을 새로 생성한 테이블에 저장

```
CREATE TABLE profession
(
  id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  profession varchar(20)
);
```

Create the profession table with a primary key column and a VARCHAR column to hold the professions.

```
INSERT INTO profession (profession)
SELECT profession FROM my_contacts
GROUP BY profession
ORDER BY profession;
```

Now fill up the profession column of the profession table with the values from your SELECT.

- METHOD 2 : SELECT, ALTER 사용
  - 열에 대한 테이블을 생성할 때 열 값들을 찾는 SELECT 문을 사용
  - ALTER를 이용하여 기본키 생성

```
CREATE TABLE profession AS
SELECT profession FROM my_contacts
GROUP BY profession
ORDER BY profession;
```

Create the profession table with one column, full of the values from the SELECT...

```
ALTER TABLE profession
ADD COLUMN id INT NOT NULL AUTO_INCREMENT FIRST,
ADD PRIMARY KEY (id);
```

... then ALTER the table to add in the primary key field

- METHOD 3 : 동시에 CREATE, INSERT 사용

- 기본키와 SELECT문의 데이터를 포함한 테이블을 하나의 쿼리로 생성
- 열에 대한 테이블을 기본키와 데이터들(eg. 직업 데이터들)의 필드로 구성하도록 생성
- 동시에, SELECT문을 사용하여 데이터 저장

```
CREATE TABLE profession
(
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    profession varchar(20)
) AS
SELECT profession FROM my_contacts
GROUP BY profession
ORDER BY profession;
```

- **AS** : SQL이 문장 내에서 열과 테이블에 별명(alias)라는 새로운 이름을 부여하여 혼란을 줄이는 쿼리
  - 테이블을 새롭게 만들면서 기존 테이블의 데이터를 옮겨올 때 i.e 쿼리의 결과를 테이블에 넣으려고 할때 사용 (METHOD 3 참고)
  - field\_name AS field\_alias : field\_name의 별명으로 field\_alias로 지정
  - table\_name AS table\_alias : table\_name의 별명으로 table\_alias로 지정.
  - 연결 이름(correlation name) : 테이블의 별명
  - AS 생략 : 테이블과 필드 바로 뒤에 별명을 써도 AS 쿼리 동작

## • JOIN(결합 구문)

- 한 데이터베이스 내의 여러 테이블의 레코드를 조합하여 하나의 열로 표현한 것
- 조인은 2개의 테이블 내에서 각각의 공통값을 이용하여 필드를 조합하는 수단

## • CROSS JOIN

- 한 테이블의 모든 행과 다른 테이블의 모든 행을 짝지워 반환
- 즉, 두 테이블에서 쿼리를 수행하여 두 개의 열에 대한 모든 경우의 조합을 반환
- Example :

```
SELECT t.toy, b.boy
FROM toys AS t
CROSS JOIN
boys AS b;
```

- 결과 : toys 테이블의 toy열과 boys 테이블의 boy열에 대한 모든 경우의 조합이 (t.toy, b.boy) 형태로 반환
- 콤마 조인 CROSS JOIN을 없애고, 콤마를 사용하여 같은 결과 반환 가능

```
SELECT toys.toy, boys.boy
```

```
■ FROM toys, boys;
```

- 내부 조인은 기본적으로 크로스 조인의 결과 중 일부가 쿼리의 조건에 의해 제거

- 내부 조인

- 내부 조인은 조건을 사용하여 두 테이블의 레코드를 결합
- 결합된 행들(CROSS JOIN의 결과값)이 조건을 만족할 경우에만 열들이 반환
- **SELECT some\_fields FROM table 1 INNER JOIN table2 ON 조건들;**

- 동등 조인 : 내부 조인 조건에서 같은 모든 행들 반환하는 내부 조인

- **SELECT some\_fields FROM table1 INNER JOIN table2 ON table1.field1 = table2.field2;**

- 비동등 조인 : 내부 조인 조건에서 같지 않은 모든 행을 반환하는 내부 조인

- **SELECT some\_fields FROM table1 INNER JOIN table2 ON table1.field1 <> table2.field2;**
- 예시 : 아이들의 정보가 있는 boy테이블과 장난감 정보가 있는 toy테이블(toy\_id, toy)에서 아이들이 어느 장난감이 없는지 알 수 있음

```
SELECT boys.boy, toys.toy
FROM boys
INNER JOIN
toys
ON boys.toy_id <> toys.toy_id
■ ORDER BY boys.boy;
```

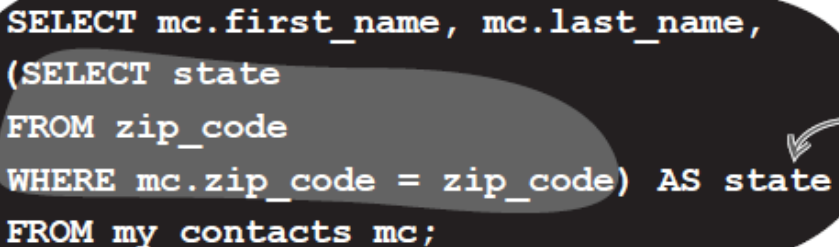
- 자연 조인 : 두 테이블에 같은 이름의 열이 있을 때만 동작하는 내부 조인

- 일치하는 열 이름을 사용하는 내부 조인
- **SELECT some\_fields FROM table1 NATURAL JOIN table2** : 테이블1과 테이블2의 같은 이름의 열의 데이터가 일치하는 행들을 반환
- 동등 조인을 통해 같은 결과 반환 가능

## [ 서브 쿼리 : 쿼리 안의 쿼리 ]

- SQL 도구 상자
  - 내부 쿼리(서브 쿼리) : 쿼리 안의 SELECT 쿼리
  - 외부 쿼리 : 내부 쿼리(서브 쿼리)를 포함한 쿼리
  - 비상관 서브 쿼리 : 외부 쿼리의 내용을 참조하지 않고 독립적으로 수행될 수 있는 서브 쿼리
  - 상관 서브 쿼리 : 외부 쿼리에서 반환된 값에 의존하는 서브 쿼리
- Motivation
  - 데이터베이스에 하나 이상의 쿼리(질문)을 해야 할 경우
  - 쿼리의 결과를 가져다 다른 쿼리의 입력으로 사용해야 할 경우
- 서브 쿼리 : 다른 쿼리에 싸여진 SELECT 쿼리
  - WHERE/FROM/SELECT 절 안에 들어가는 "하나의 SELECT 쿼리"을 통칭
  - 서브 쿼리는 괄호로 감싼 후 사용
  - 외부 쿼리 : 서브 쿼리를 감싸고 있는 쿼리
  - 내부 쿼리 : 서브 쿼리
  - 서브 쿼리를 통해 조인보다 간단한 쿼리문 작성 가능.  
(BUT!!) 조인이 서브 쿼리보다 쿼리 실행속도가 빠름
- 중첩서브쿼리(nested subqueries) : WHERE 절 안에 있는 서브 쿼리
  - WHERE field\_name IN (서브쿼리-SELECT ~~) : 서브 쿼리의 결과들에 포함되는 field\_name의 레코드들
- 스칼라 서브쿼리(scalar subqueries) : SELECT 절 안에 있는 서브 쿼리
  - SELECT 절 안에 있는 서브 쿼리는 반드시 단일 값만 리턴
    - SUM,COUNT,MIN,MAX 등과 같은 함수가 많이 쓰인다
  - Example :

```
SELECT mc.first_name, mc.last_name,
(SELECT state
FROM zip_code
WHERE mc.zip_code = zip_code) AS state
FROM my_contacts mc;
```



- AS state : state이라는 열의 이름으로 서브 쿼리의 값을 열들의 행으로 반환

- 비상관 서브 쿼리 : 외부 쿼리의 어떤 것도 참조하지 않고 단독으로 사용되는 서브 쿼리
  - 비상관 서브쿼리는 외부 쿼리의 값과 상관없이 단독으로 쿼리 실행
  - 즉, 비상관 서브 쿼리에서는 우선 해당 내부 쿼리를 수행하고 그 값을 사용해서 외부 쿼리의 결과를 반환
  - IN, NOT IN을 사용하여 값이 서브 쿼리의 반환된 집합의 원소인지 아닌지 확인 가능 (중첩서브쿼리)

- 상관 서브 쿼리 : 해당 내부 쿼리의 값이 외부 쿼리에 의존
  - 외부 쿼리가 실행되어야만 내부 쿼리가 수행될 수 있는 구조
  - 외부 쿼리에서 만들어진 별명이 내부 쿼리에서 사용
  - EXISTS, NOT EXISTS 키워드를 포함시켜 유용하게 사용 가능
  - Example 1 : 현재 직업이 없는 사람들 즉, job\_current 테이블에 없는 모든 사람의 이름과 이메일 정보 추출

```
SELECT mc.first_name firstname, mc.last_name lastname, mc.email email
FROM my_contacts mc
WHERE NOT EXISTS
  (SELECT * FROM job_current jc
   WHERE mc.contact_id = jc.contact_id );
```

*NOT EXISTS finds the first and last names and email addresses of the people from the my\_contacts table who are not currently listed in the job\_current table.*

- Example 2 : 관심사를 등록한 사람들 즉, contact\_interest 테이블에 있는 모든 사람의 이름과 이메일 정보 추출

```
SELECT mc.first_name firstname, mc.last_name lastname, mc.email email
FROM my_contacts mc
WHERE EXISTS
  (SELECT * FROM contact_interest ci WHERE mc.contact_id = ci.contact_id );
```

*EXISTS finds the first and last names and email addresses of the people from the my\_contacts table whose contact\_id shows up at least once in the contact\_interest table.*

- 서브 쿼리 안에 서브 쿼리를 만드는 가장 좋은 방법 : 쿼리할 질문을 여러 부분으로 나누어 각 부분에 대해 작은 쿼리를 만든다

- 서브 쿼리와 사용하는 키워드들
  - WHERE field\_name 키워드 (서브쿼리-SELECT ~~ ) 형태로 사용
  - ALL : 서브 쿼리의 결과 집합 모두에 해당
    - Example : 서브 쿼리 결과 모두 보다 큰 rating을 받은 식당들 반환

```
SELECT name, rating FROM restaurant_ratings
WHERE rating > ALL
  (SELECT rating FROM restaurant_ratings
   WHERE rating > 3 AND rating < 9);
```



- ANY : 서브 쿼리의 결과 집합 중 어느 하나라도 조건에 맞는지 확인
    - Example : 서브 쿼리 결과 중 rating이 (5, 7) 중 어느 하나보다 큰 열들 반환
 

```
SELECT name, rating FROM restaurant_ratings
WHERE rating > ANY
(SELECT rating FROM restaurant_ratings WHERE
rating > 3 AND rating < 9);
```
  - SOME : SQL 표준 문법과 MySQL에서 ANY와 같은 의미
- (IMPORTANT) : 서브쿼리가 할 수 있는 일은 조인을 사용하여도 할 수 있음

## [ 외부 조인 ]

- SQL 도구상자
  - 왼쪽 외부 조인 : 왼쪽 테이블의 모든 행들을 가져다 오른쪽 테이블의 행에 대응
  - 오른쪽 외부 조인 : 오른쪽 테이블의 모든 행들을 가져다 왼쪽 테이블의 행에 대응
- 외부 조인
  - 조인되는 테이블 중 하나의 모든 행을 다른 테이블에서 일치하는 정보와 함께 반환
  - 내부 조인과 다르게, 외부 조인은 다른 테이블과 일치하는 것이 있는가에 상관없이 행을 반환
    - 일치하는 것이 없는 경우 NULL값으로 표시
  - 내부 조인과 비교해 보면, 외부 조인은 두 테이블 사이의 관계와 관련
  - 두 테이블 사이의 순서가 중요하지 않은 내부 조인과 다르게 외부 조인에서는 테이블들의 순서가 중요
- 왼쪽 외부 조인(left outer join) : left\_table LEFT OUTER JOIN right\_table ON
  - 왼쪽 테이블의 모든 행을 가져다 오른쪽 테이블의 행과 비교
  - 왼쪽 테이블과 오른쪽 테이블이 일대다 관계일 때 유용
  - 왼쪽 테이블 : FROM 뒤에 그리고 조인 전에 나오는 테이블
  - 오른쪽 테이블 : 조인 뒤에 나오는 테이블
  - 즉, 조인 앞에 나오는 테이블이 왼쪽 테이블이고 조인 뒤에 나오는 테이블이 오른쪽 테이블

- 왼쪽 외부 조인의 결과는 왼쪽 테이블의 해당 행들의 갯수만큼 반환
  - 왼쪽 외부 조인의 결과가 NULL값이면 오른쪽 테이블에는 왼쪽 테이블에 상응하는 값이 없다는 것을 의미
  - 즉, NULL값을 갖는 열은 오른쪽 테이블의 열
- Example :
 

```
SELECT g.girl, t.toy
FROM toys t
LEFT OUTER JOIN girls g
ON g.toy_id = t.toy_id;
```

  - 왼쪽 테이블 : toy
  - 오른쪽 테이블 : girl
  - toy 행들에 일치하지 않는 girl들의 행에 대해 g.girl은 NULL

- **오른쪽 외부 조인(right outer join) : right\_table RIGHT OUTER JOIN right\_table ON**

- 오른쪽 테이블을 왼쪽 테이블과 비교한다는 점을 제외하면 왼쪽 외부 조인과 동일

- **완전 외부 조인**

- 왼쪽 외부 조인과 오른쪽 외부 조인을 합친 조인
- 왼쪽과 오른쪽 결과를 모두 가져오는 조인
- (BUT!!) MySQL, SQL Server, Access에서는 지원 X

## [ 셀프 조인 ]

- SQL 도구상자

- 자기 자신을 참조하는 키 : 특정 목적을 위해 같은 테이블 내의 기본키를 가리키는 참조키
- 셀프 조인 : 하나의 테이블을 같은 정보의 테이블이 두 개인 것처럼 쿼리 실행

- Motivation

- 사원의 id와 이름이 있는 employee\_table 테이블이 주어지고, 각 사원의 상사의 사원 id를 추가하려는 상황

- Method 1 :(사원\_id, 상사\_id)로 구성 된 Boss\_table 테이블을 추가

- 사원\_id는 일대일 관계이고, 상사\_id는 일대다 관계
  - 기본키와 참조키가 모두 원래 employee\_table에서 온 열
  - 새 테이블에는 새로운 정보가 없어 비효율적
- Method 2 : boss\_id라는 행을 employee\_table에 추가
  - boss\_id는 "자기 자신을 참조하는 참조키"
- 자기 자신을 참조하는 참조키
  - 같은 테이블에서 또다른 목적으로 사용하는 기본키
  - 같은 테이블의 다른 필드를 참조하는 키라는 뜻
  - 내부 조인을 사용하기 위해서는 같은 테이블이 두 개가 필요
    - 정규화된 데이터베이스에서는 같은 테이블을 두 개 사용할 일이 없음
    - (Result) 새로운 조인이 필요한 상황
- 셀프 조인(Self Join)
  - 동일 테이블 사이의 조인
  - 하나의 테이블로 같은 정보를 가진 테이블이 두 개 있는 것처럼 쿼리를 보낼 수 있음
  - From table1 AS a1 INNER JOIN table2 AS a2 라는 형태로 같은 테이블에 다른 별명을 주어 조인하는 형태
  - Example :
 

```
SELECT c1.name, c2.name AS boss
FROM clown_info c1
INNER JOIN clown_info c2
ON c1.boss_id = c2.id;
```
- 셀프 조인 => 서브 쿼리

## [ 유니온(UNION), INTERSECTION, EXCEPT ]

- SQL 도구상자
  - UNION : SELECT의 열 리스트를 기반으로 두 개 이상 쿼리의 결과를 합해 하나의 테이블로 표현
  - UNION ALL : UNION은 중복된 값들을 하나만 표시하지만, UNION ALL은 중복값을 모두 표시

- CREATE TABLE AS : SELECT문의 결과로 테이블을 만들때 사용하는 키워드
- INTERSECTION : 첫번째 쿼리의 결과와 두번째 쿼리의 결과 모두에서 나오는(교집합) 값들 반환
- EXCEPT : 첫번째 쿼리의 결과이면서 두번째 쿼리의 결과는 아닌(차집합) 값들 반환

## • 유니온(UNION)

- 두 개 이상의 테이블을 묶는 방법 중 하나
- SELECT의 열 리스트를 바탕으로 두 개 이상 쿼리의 결과를 하나의 테이블로 합친다
- 즉, 유니온의 결과는 각 SELECT의 값들이 합집합의 결과(중복된 결과는 1개만 반환)
- Example 1 :  

```
SELECT field_name1 FROM table1
UNION
SELECT field_name1 FROM table2
UNION
SELECT field_name1 FROM table3
ORDER BY field_name1;
```

  - 테이블 table1, table2, table3에서 field\_name1들의 합집합을 반환
- Example 2 :  

```
SELECT field_name1 FROM table1
UNION
SELECT field_name2 FROM table2
UNION
SELECT field_name3 FROM table3
```

  - SELECT문들의 결과값들  
인 field\_name1, field\_name2, field\_name3들의 합집합 반환

## • 유니온의 한계

- SELECT field\_name1 FROM table\_i ORDER BY field\_name1 처럼 UNION을 사용할 때, **ORDER BY**가 여러 개 나온다면 해석 불가능
- 유니온은 쿼리문의 끝에 단 한 개의 ORDER BY만 해석 가능
  - (why?) 유니온은 여러 SELECT 문의 결과를 합친 후 결과를 만들어 내기 때문

## • 유니온에 대한 SQL 규칙

- 각 SELECT 문에서 반환하는 열의 갯수는 동일해야 한다 (Example1, Example2는 모두 한 개의 열을 반환)
- SELECT 문에 표현식과 집계 함수도 동일해야 한다

- SELECT 문의 순서는 중요하지 않다
- 유니온의 결과에서 중복값은 하나로 나오는 것이 디폴트
  - 특별한 이유로 중복을 보고 싶으면 **UNION ALL**을 사용 가능
- 각 SELECT 문에서 반환하는 열들의 데이터는 타입이 같거나 서로 변환 가능한 값이어야 한다
  - 즉, 반환되는 데이터 타입이 호환가능한 타입으로 변환 될 것을 의미
  - 변환할 수 없으면 쿼리 실패

• Application : 유니온을 사용한 테이블 생성

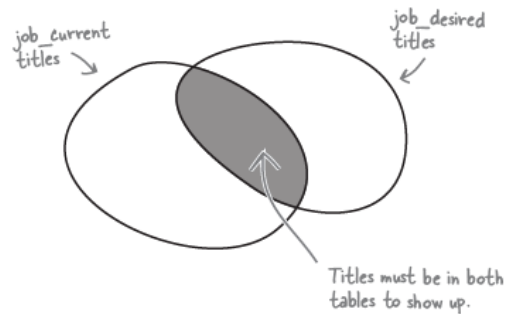
```
CREATE TABLE my_union AS
SELECT title FROM job_current UNION
SELECT title FROM job_desired
UNION SELECT title FROM job_listings;
```

- 새로 생성하는 테이블 my\_union의 행들은 job\_curreunt와 job\_desired 테이블의 의 title의 합집합으로 구성

• INTERSECTION

- 유니온과 거의 같은 방식으로 사용
  - UNION 키워드에 대체하여 사용 가능
- INTERSECTION의 결과는 각 SELECT의 값들이 교집합의 결과

```
SELECT title FROM job_current
INTERSECT
SELECT title FROM job_desired;
```

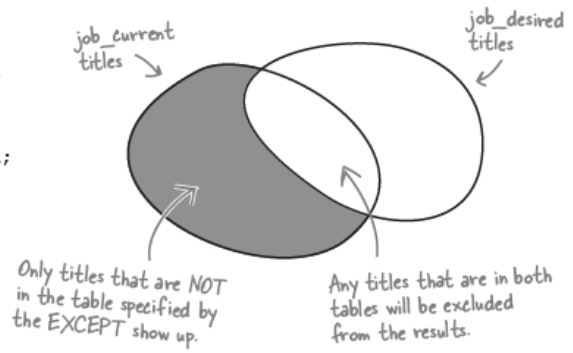


◦

• EXCEPT

- 유니온과 거의 같은 방식으로 사용
  - UNION 키워드에 대체하여 사용 가능
  - (BUT!!) UNION, INTERSECTION과 다르게 SELECT 쿼리문은 두 개만 온다
- 앞의 SQL 문의 결과에서 뒤의 SQL문의 결과에 대한 차집합을 반환

```
SELECT title FROM job_current
EXCEPT
SELECT title FROM job_desired;
```



○

## [ 서브 쿼리 VS 조인 ]

- 서브 쿼리는 내부 조인으로 변경 가능
- 셀프 조인은 서브 쿼리로 변경 가능
  - 이때 서브 쿼리는 외부 쿼리의 결과에 의존 -> 상관 쿼리
- 조인의 특징
  - 결과에 여러 테이블이 필요한 경우 가장 좋은 방법
  - 서브 쿼리보다 수행 속도가 빠름
  - 서브 쿼리보다 쿼리문 파악이 쉬움
- 서브 쿼리의 특징
  - 집계(SUM, AVG, MIN, MAX)를 사용하는데 편함
    - 즉, 여러 열을 반환하지 않는 대신 편리한 기능 제공
  - UPDATE, INSERT, DELETE와 함께 사용 가능

## [ 제약 조건 - 사람들이 잘못된 데이터를 입력하지 못하게 하는 방법 ]

- SQL 도구상자
  - CHECK 제약 조건 : 특정 값들만이 테이블에 추가되고 변경하게 만드는 제약 조건
- 제약 조건
  - 열에 넣을 수 있는 값에 대해 제한하는 것
  - ex) NOT NULL, PRIMARY KEY, FOREIGN KEY, UNIQUE
- CHECK 제약 조건
  - CHECK (conditions) : WHERE절과 같은 조건식을 사용하는 제약 조건 키워드

- 넣으려는 값이 체크 제약 조건에 맞지 않으면 에러 발생
- (주의) MySQL에서는 체크 제약 조건을 사용해도 제약 조건이 추가되지 않음
  - MySQL에서는 특정 조건이 맞으면 쿼리가 실행되도록 하는 **트리거** 사용
- 서브 쿼리를 제외하고 WHERE절에서 사용할 수 있는 대부분의 조건을 CHECK의 조건으로 사용 가능
- Example 1 : 테이블 생성시 체크 제약 조건
 

```
CREATE TABLE piggy_bank
(
    id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    coin CHAR(1) CHECK (coin IN ('P', 'N', 'D', 'Q'))
)
```
- Example 2 : 기존 테이블에서 테이블의 열에 체크 제약 조건 추가
 

```
ALTER TABLE my_contacts
ADD CONSTRAINT CHECK gender IN ('M', 'F');
```

- 여러 사람이 데이터베이스를 이용할 때 데이터베이스를 관리, 통제

## [ 뷰 - 데이터의 일부만을 볼 수 있게 할 방법 ]

- SQL 도구상자
  - 뷰 : 쿼리의 결과를 테이블로 다룰 때 사용
  - 업데이트 가능한 뷰 : 뷰와 관련된 테이블의 데이터를 변경할 수 있는 뷰로 관련 테이블의 NOT NULL인 열들만 포함
  - 업데이트 불가능한 뷰 : 뷰와 관련된 테이블에 데이터를 INSERT/UPDATE하는데 이용할 수 없는 뷰
  - CHECK OPTION : 업데이트 가능한 뷰에 대한 INSERT와 UPDATE가 뷰의 WHERE절을 만족하게 만드는 키워드
- Motivation
  - 자주 사용하는 쿼리를 데이터베이스 안에 저장하여 실수를 줄이고 싶은 상황
- 뷰(VIEW)
  - 쿼리 상에만 존재하는 "테이블"
  - 테이블처럼 행동하고, 테이블에서 할 수 있는 조작을 똑같이 수행할 수 있는 "가상 테이블"
  - 이름이 있는 뷰는 지속 가능

- 뷰를 사용하여 새 열이 데이터베이스에 추가될 때마다 새 정보가 뷰에 반영
- 뷰가 데이터베이스에서 유용한 이유
  - DB의 구조를 변경하여도 데이터에 의존하는 어플리케이션을 변경할 필요가 없음
    - 데이터에 대한 뷰를 만들면 시스템 내부의 테이블 구조를 변경하여도, 기존 테이블의 구조를 흉내내는 뷰를 만들면 어플리케이션은 변경 없이 사용 가능
  - 복잡한 쿼리를 간단한 명령으로 단순하게 만들 수 있음
    - 뷰는 복잡한 쿼리의 내용을 숨김
    - 조인이 많고 복잡한 쿼리를 뷰를 통해 간단하게 만들
    - "간단하다" : 입력 실수의 가능성이 줄고 코드의 가독성이 높아진다는 의미
  - 사용자에게 필요없는 정보를 숨기기 가능
    - 뷰를 만들어 테이블의 세부 정보를 보이게 하지 않고 그 정보 이용 가능
    - 민감한 정보를 숨기면서 필요한 정보만 볼 수 있게 가능
- 뷰 생성/삭제 및 사용
  - **CREATE VIEW** view\_name **AS** 저장할 쿼리문 : 뷰 생성 쿼리
  - **DROP VIEW** view\_name : 해당 뷰 삭제
  - **SELECT \* FROM** view\_name : 뷰를 사용한 **SELECT** 쿼리
  - 뷰 자체에 **INSERT**, **UPDATE**, **DELETE** 또한 사용 가능
    - 즉, 뷰를 하나의 테이블로써 조작 가능
  - 뷰를 통한 데이터 변경
    - 뷰가 집계 값을 사용하는 경우, 데이터 변경에 뷰 사용 불가능
    - 뷰에 **GROUP BY**, **DISTINCT**, **HAVING**이 포함되어 있어도 데이터 변경 불가능
- **WITH CHECK OPTION**
  - **CREATE VIEW** view\_name **AS** ~~~ **WHERE** where\_절 **WITH CHECK OPTION**
  - 해당 뷰에서 **INSERT**, **UPDATE**를 사용한 각 문장을 확인하여 뷰의 where\_절에서 허용하는지 알아보는 쿼리문
  - MySQL에서는 **CHECK OPTION**을 통해 **CHECK** 제약조건 흉내 가능
    - 뷰는 테이블의 내용을 그대로 반영
    - **INSERT** 문장은 **WHERE**절에 허용되는 데이터만 추가되도록 강제



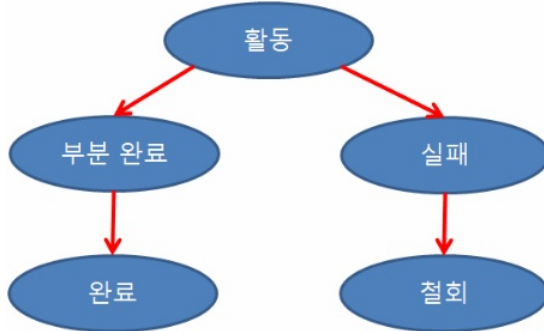
- 변경 가능한 뷰
  - 관련 테이블을 변경할 수 있는 뷰
  - 참조하는 테이블들의 열이 모두 NOT NULL이어야 한다
  - 즉, 테이블에서의 NOT NULL인 열들만을 반환하면, 뷰는 적당한 값을 테이블에 입력 가능
  - INSERT, UPDATE, DELETE을 뷰에 사용 가능하다는 뜻
    - (BUT!!) 일반적으로는 INSERT, UPDATE, DELETE할 때는 테이블을 직접 이용하는 것이 유용
  - 집계 연산자(AVG,MAX,...)나 HAVING,IN,NOT IN과 같은 연산자를 포함하지 않은 뷰만 업데이트 가능
- 기타 뷰에 관련된 정보들
  - SHOW DATABASE 명령어를 통해 모든 뷰와 테이블 출력 가능
  - DESC를 사용하여 뷰의 구조 출력 가능
  - 뷰와 관련된 테이블을 DROP하는 경우 RDBMS 소프트웨어에 따라 결과가 다름
    - MySQL은 관련 테이블을 DROP 가능하지만, 뷰에 관련된 테이블이 없는 경우 해당 뷰 DROP 불가능
- 여러 사람이 데이터베이스를 이용할 때 데이터베이스를 관리, 통제

## [ 트랜잭션 - 데이터를 동시에 입력할 때 서로 엉키지 않게 할 방법 ]

- SQL 도구상자
  - 트랜잭션 : 하나의 단위로서 수행되어야 하는 쿼리의 집합
  - START TRANSACTION : RDBMS에 트랜잭션이 시작됨을 알릴 때 사용하며 COMMIT, ROLLBACK과 함께 사용
- 트랜잭션의 정의
  - 데이터베이스의 상태를 변환시키는 하나의 논리적 기능을 수행하기 위한 작업 단위
    - 즉, 한 단위의 일을 수행하는 일련의 SQL문
  - 트랜잭션동안 "모든" 단계가 방해없이 완료될 수 없다면 어떤 것도 완료되지 말아야 한다
- 트랜잭션의 성질(ACID)
  - ATOMICITY(원자성)

- 트랜잭션을 구성하는 모든 명령이 실행되거나 어떤 명령도 실행되지 않아야 함
    - 트랜잭션의 일부만 수행 불가능
    - 트랜잭션의 구성 요소 중 하나라도 오류가 발생하면 트랜잭션 전부가 취소
  - CONSISTENCY(일관성)
    - 트랜잭션의 실행이 성공적으로 완료되면 언제나 일관성 있는 DB상태로 변환
      - 즉, DB의 일관성 유지
    - 시스템이 가지고 있는 고정요소는 트랜잭션 수행 전과 후의 상태가 같아야 함을 의미
  - ISOLATION(독립성)
    - 모든 트랜잭션은 동시에 일어나는 다른 트랜잭션과 상관 없이 DB에 대해 일관된 뷰를 가지고 있어야 한다
    - 둘 이상의 트랜잭션이 동시에 병행 실행되는 경우 어느 하나의 트랜잭션 실행 중에 다른 트랜잭션의 연산이 끼어들 수 없음
    - 수행중인 트랜잭션이 완전히 완료될 때까지 다른 트랜잭션에서 해당 수행 결과를 참조 불가능
    - Example : 동시에 동일 계좌에 대해서 인출/잔고 확인 등을 하는 경우
  - DURABILITY(지속성)
    - 성공적으로 완료된 트랜잭션의 결과는 시스템이 고장나더라도 영구적으로 반영되어야 함
    - 즉, 이상 상황으로 부터 데이터를 보호
    - 트랜잭션의 기록을 주 DB외에 다른 위치의 DB에 저장하는 방식을 사용
- 트랜잭션을 위한 SQL
    - START TRANSACTION;
      - COMMIT, ROLLBACK이 나올때까지의 실행되는 모든 SQL을 추적
    - COMMIT;
      - 트랜잭션이 행한 연산이 완료된 것을 트랜잭션 관리자에게 알려주는 연산
      - 모든 명령이 문제가 없는 경우 해당 트랜잭션의 내용을 영속적으로 만든다
      - 즉, COMMIT전에는 DB에 어떤 변화도 발생 X
    - ROLLBACK;
      - 트랜잭션 시작(START TRANSACTION;)전 상태로 되돌리는 연산
    - 트랜잭션의 상태
      - 활동(Active) : 트랜잭션이 실행 중인 상태
      - 실패(Failed) : 트랜잭션 실행에 오류가 발생하여 중단된 상태

- 철회(Aborted) : 트랜잭션이 비정상적으로 종료되어 Rollback 연산을 수행한 상태
- 부분 완료(Partially Committed) : 트랜잭션의 마지막 연산까지 실행했지만, Commit 연산이 실행되기 직전 상태
- 완료(Committed) : 트랜잭션이 성공적으로 종료되어 Commit 연산을 실행한 후의 상태



## MySQL에서의 트랜잭션 수행

### ◦ 저장 엔진(storage engine)

- 데이터베이스의 모든 데이터와 구조를 저장하는 하부 구조
- SHOW CREATE TABLE table\_name 에서 맨 마지막 줄에 출력되는 ENGINE=~에 해당하는 부분

### ◦ MySQL에서 트랜잭션을 수행하려면 올바른 저장엔진을 사용해야 함

- InnoDB 나 BDB 엔진을 사용하면 트랜잭션 이용 가능

### ◦ Example :

```

START TRANSACTION;

SELECT * FROM piggy_bank;

UPDATE piggy_bank set coin = 'Q' where coin= 'P';

SELECT * FROM piggy_bank; ← Now you see the changes...

ROLLBACK; ← We changed our minds

SELECT * FROM piggy_bank; ← ...and now you don't
  
```

## • 트랜잭션 로그 : RDBMS에 저장된 트랜잭션 중 수행된 내용들

## [ 보안 - 사용자의 권한 ]

### • SQL 도구 상자

- CREATE USER : RDBMS에서 사용자 계정을 만들고 암호를 부여하는 명령어
- GRANT : 사용자에게 권한을 부여하고, 그 권한을 바탕으로 사용자의 DB 접근을 통제

- REVOKE : 사용자로부터 권한 제거
- WITH GRANT OPTION : 사용자가 자신이 가진 권한과 같은 권한을 다른 사람에게 부여 가능
- ROLE : 권한의 집합으로 사용자에게 특정 권한들을 묶어 할당 가능
- WITH ADMIN OPTION : 역할을 가진 사용자가 자신의 역할을 다른 사람에게 부여 가능

- GOAL

- DB와 DB내의 객체들을 안전하게 보호하는 법
- 데이터를 누가 어떻게 사용할지 통제하는 법

- 배경

- 현재 특정 사용자가 CRUD에 대한 모든 권한이 있는 상황
- 이 사용자가 현재 DB를 조작하면서 지속적으로 에러를 발생하는 상황
- 해결 방법
  - 해당 사용자에게 대해서 SELECT 권한만 부여하고 데이터의 추가(INSERT)나 변경(UPDATE, ALTER) 금지
  - 해당 사용자가 만든 에러를 수정하기 위해 데이터를 삭제할 때, 에러가 발생한 데이터 뿐만 아니라 다른 데이터가 삭제될 위험 요소를 삭제해야 한다

- SQL은 사용자들이 DB에 "할 수 있는 일"과 "할 수 없는 일"을 제어할 방법 제공

- 제어하기 위해서는 해당 DB를 사용하는 모든 사용자들에게 **사용자 계정**을 부여해야 함

- 루트 사용자 계정 보호

- 루트 사용자

- 데이터베이스에 대한 모든 권한을 가지고 있는 사용자
- 데이터베이스의 첫번째 사용자는 루트 사용자
- 다른 모든 사용자의 계정을 만들고 권한을 부여할 수 있는 사용자

- 루트 사용자 계정 보호를 위한 SQL

- MySQL : SET PASSWORD FOR 'root@localhost' = PASSWORD('비밀번호');
- Oracle : ALTER USER ROOT IDENTIFIED BY '비밀번호';
- localhost : 쿼리를 실행하는데 사용하는 컴퓨터가 db에 설치된 컴퓨터와 같은 컴퓨터라는 의미

- 컴퓨터가 어디에 있는지 쿼리에 알려야 하는 경우(원격 접속을 하는 경우)
      - IP 주소 사용
      - localhost 대신 hostname 사용
- 사용자에 대한 정보
  - SQL은 자신에 대한 데이터베이스 보유
  - 해당 DB는 사용자의 ID, 이름, 암호, 각 DB에 대한 권한 등을 포함
- 새로운 사용자 추가
  - 사용자 생성은 RDBMS마다 방법이 다양
  - MySQL
    - CREATE USER user\_name IDENTIFIED BY 'password'
  - 사용자 계정을 생성하는 동시에 권한 부여 또한 가능
  - 사용자 생성과 권한 부여 방법을 따로 알고 있으면 DB에 변경이 생기는 경우에 사용자 권한을 추후에 변경할 수 있다는 장점 존재
- **GRANT** : 사용자에게 권한 부여
  - 데이터베이스의 사용자에게 특정 권한을 부여할 수 있는 키워드
  - GRANT를 통해 가능한 권한 부여
    - 특정 테이블은 특정 사용자들만 변경 가능
    - 특정 테이블의 데이터는 특정 사용자만 접근 가능
    - 특정 테이블의 특정 테이블에 대한 권한
  - **GRANT** 부여할 권한(들) **ON** table\_name **TO** user\_name : 특정 유저에게 특정 테이블에 대한 특정 권한 부여
    - 권한(들)
      - SELECT/UPDATE/DELETE/INSERT
      - SELECT, UPDATE : 해당 두 키워드에 대한 권한 부여
      - ALL : 모든 권한 부여
    - Ex) GRANT SELECT ON clown\_info TO elsie : 유저 elsie에게 clown\_info 테이블에 대해서 SELECT 권한 부여
  - GRANT SELECT ON \*.\* TO user :
    - 해당 유저에게 모든 DB의 모든 테이블에 SELECT 권한을 주는 쿼리문
    - 첫번째 \* : DB 이름
    - 두번째 \* : 테이블 이름
  - GRANT의 다양한 형태
    - 같은 GRANT 문에서 여러 사용자 지목 가능

- **TO** uesr1, user2,...;
- **WITH GRANT OPTION**
  - 해당 사용자에게 자신이 받은 권한을 줄 수 있는 권한 부여
  - Example : 유저 happy,sleepy에게 chores 테이블에 DELETE할 권한을 부여하고, 다른 사람들에게 같은 권한을 줄 권한도 부여
 

```
GRANT DELETE ON chores
TO happy, sleepy
WITH GRANT OPTION;
```
- 전체 테이블 대신 테이블 내의 특정 열 또는 열들에 대한 권한 부여 가능
  - **GRANT SELECT**(field\_name1, field\_name2,...) **ON** table\_name **TO** user1, user2,...;
  - **SELECT**를 제외하고는 열에 대한 권한을 주는 대부분의 **GRANT** 문은 의미가 없는 권한
- 테이블에 대해 하나 이상의 권한 지정
- **GRANT ALL**을 통해 사용자에게 특정 테이블에 대한 **CRUD**(**INSERT**, **SELECT**, **UPDATE**, **DELETE**) 권한 부여
- 특정 데이터베이스(database\_name)에 대한 모든 테이블은 database\_name.\*으로 지정 가능

## • **REVOKE** : 권한 취소

- 데이터베이스의 사용자에게 특정 권한을 취소할 수 있는 키워드
- **REVOKE** 취소할 권한(들) **ON** table\_name **FROM** user\_name : 특정 유저에게 특정 테이블에 대한 특정 권한 취소
  - **GRANT** 쿼리에서의 **TO** 대신 **FROM**을 사용한다는 점을 빼고는 **GRANT**와 문법이 동일
  - Ex) **REVOKE SELECT ON** clown\_info **FROM** elsie
- **WITH GRANT OPTION**에 대한 권한 제거 가능
  - **REVOKE GRANT OPTION ON** table\_name 취소할 권한(들) **ON** table\_name **FROM** user1, user2,...;
  - 원래의 권한은 그대로 두고 변경 하지 않음
  - 즉, "권한 부여"에 대한 권한만 취소
  - Ex) **REVOKE GRANT OPTION ON DELETE ON** chores **FROM** happy, sleepy; : 사용자 happy, sleepy는 chores 테이블에 대한 **DELETE** 권한을 여전히 보유하지만, 다른 유저에게 해당 권한 부여 불가능
- 루트 사용자가 특정 사용자에게 **WITH GRANT OPTION**에 대한 권한을 제거하는 경우 특정 사용자가 부여하였던 다른 사용자들에 대한 권한 또한 삭제
  - 즉, 특정 사용자의 권한을 제거하면 해당 사용자에게 종속된 다른 사용자들에 대한 권한들이 같이 제거
- **CASCADE**, **RESTRICT** 키워드를 사용하면 권한이 보존되는 사용자와 보존 되지 않는 사용자를 명확히 파악 가능

- 두 키워드는 권한을 보존하면서 루트 사용자에게 에러 반환
  - CASCADE : 권한을 없앨 사용자가 권한을 준 사용자들의 권한 또한 제거
    - REVOKE에서의 디폴트 값(일반적인 경우)
    - **REVOKE** 취소할 권한(들) **ON** table\_name **FROM** user\_name **CASCADE** : 특정 사용자에게 특정 테이블에 대한 특정 권한을 취소하고 해당 사용자가 권한을 준 다른 사람들의 권한 또한 제거
    - CASCADE는 REVOKE가 지목한 사용자 뿐만 아니라 해당 사용자와 연결된 모든 사람들에게 영향을 준다는 의미
  - RESTRICT : 권한을 없앨 사용자가 이미 다른 사용자에게 권한을 주었으면 에러 반환
    - **REVOKE** 취소할 권한(들) **ON** table\_name **FROM** user\_name **RESTRICT** : 특정 사용자에게 특정 테이블에 대한 권한을 취소하는 과정에서 해당 사용자에게 영향을 받는 사용자가 존재하면 REVOKE문에 대한 에러 반환
- 
- 뷰에 대한 권한 설정
    - 업데이트 가능한 뷰는 테이블에 사용한 GRANT/REVOKE를 같은 방식으로 사용 가능
    - 업데이트 가능하지 않은 뷰는 권한이 있어도 INSERT를 실행 불가
- 
- 역할(ROLE)
    - 특정 권한을 모아 그룹의 사용자들에게 적용하는 방법
    - 역할 생성 및 권한 설정
      - CREATE ROLE role\_name  
GRANT SELECT, INSERT ON table\_name TO role\_name
      - 권한을 할당할 때 사용자 대신 역할의 이름을 사용
    - 사용자에게 역할 적용
      - GRANT role\_name TO user\_name
      - 역할 이름이 테이블의 이름과 권한을 대체
    - 역할 취소
      - REVOKE role\_name FROM user\_name
      - CASCADE, RESTRICT 또한 사용 가능
    - 역할 제거
      - DROP ROLE role\_name
      - 역할을 제거하면, 역할을 적용한 사용자들의 역할에 대한 권한들이 모두 취소
    - 사용자들은 하나 이상의 역할 보유 가능(역할들 사이에 충돌이 없는 경우에만)
    - **WITH ADMIN OPTION**

- GRANT role\_name TO user\_name WITH ADMIN OPTION
- 해당 사용자가 해당 역할을 다른 사용자에게 부여할 수 있는 권한 부여
- 역할을 가지고 있는 모든 사람이 그 역할을 다른 사람에게 주는 것을 허용

## [ 동시성 (Concurrency) ]

- Background
  - DB는 다수의 사용자들이 동시에 접근하는 경우가 빈번하게 발생
  - 사용자들에 대한 적절한 통제가 없으면, DB의 무결성이 깨지고 트랜잭션의 수행에 대해 의도하지 않은 결과를 반환할 가능성 존재
- 직렬성(Serializability) : 각각의 트랜잭션이 일정한 순서를 가지고 순차적으로 실행되는 것
- 동시성(Concurrency) : 트랜잭션들이 순차적으로 실행되는 것이 아니라, 트랜잭션을 구성하는 쿼리문들이 트랜잭션의 순서와 상관없이 실행되는 것
- 동시성 문제 1 : The Lost Update Problem
  - 서로 다른 트랜잭션에서 UPDATE 연산이 연속으로 수행되면서 먼저 실행된 UPDATE 연산이 오버라이팅되는 현상
  - Example : 트랜잭션 A의 UPDATE 연산이 오버라이팅

Transaction A	time	Transaction B
SELECT x	t <sub>1</sub>	-
-	t <sub>2</sub>	SELECT x
UPDATE x	t <sub>3</sub>	-
-	t <sub>4</sub>	UPDATE x

- 동시성 문제 2 : The Uncommitted Dependency Problem
  - 특정 트랜잭션의 실행 중 오류가 발생하여 ROLLBACK이 실행될 때 발생하는 문제
  - 특정 쿼리의 결과와 실제 DB에 저장된 값이 일치하지 않는 문제가 발생
  - Example : 실제 DB에 저장된 x값과 트랜잭션 A가 읽은 x값이 일치하지 않는 예제



Transaction A	time	Transaction B
-	$t_1$	UPDATE x
SELECT x	$t_2$	-
-	$t_3$	ERROR OCCURRED
-	$t_4$	ROLLBACK

• 동시성 문제 3 : The Inconsistent Analysis Problem

- 한 트랜잭션의 실행 중 다른 트랜잭션의 내용이 실행되어 DB의 일관성이 깨지는 현상
- Example : 트랜잭션 A의 결과가 120이 아닌 110이 나오는 문제

var1	var2	var3
40	50	30

Transaction A	time	Transaction B
SELECT var1 sum += var1	$t_1$	-
SELECT var2 sum += var2	$t_2$	-
-	$t_3$	SELECT var3
-	$t_4$	UPDATE var3 = 20
-	$t_5$	COMMIT
SELECT var3 sum += var3	$t_6$	-

• 교착 상태(Deadlock)

- 두 개 이상의 트랜잭션이 서로 lock이 해제되기를 기다리면서 무한정 기다리는 상황
- 교착 상태를 해결하는 방법
  - 교착 상태 예방(prevention) : 교착 상태가 발생하는지 미리 검사하여 교착 상태가 발생하지 않는 요청만 수용하는 방법
  - 교착 상태 회피(avoidance) : 요청을 수행하면서 교착 상태가 발생하는 상황이 오면 특정 방법을 교착 상태를 회피하는 방법
    - 특정 방법 : 큐와 힙 등을 이용한 우선순위 개념을 통한 회피 방법

- 교착 상태 탐지(detection) 및 회복 : 교착 상태가 발생하는 것을 허용하며, 교착 상태 탐지 알고리즘을 통해 교착 상태를 탐지하면 교착 상태 회복 알고리즘을 통해 교착 상태를 제거하는 방법

## • Two-phase locking protocol

- 동시성 제어를 위해 상호 배제 기능을 제공하는 방법
- 구성 단계
  - 확장 단계(growing phase) : 트랜잭션이 락을 설정하는 잠금 연산만 수행 가능하고, 락 해제 연산은 수행이 불가능한 단계
  - 축소 단계(shrinking phase) : 트랜잭션이 해제 연산만 수행 가능하고 잠금 연산은 수행 불가능 단계
- 트랜잭션은 READ, UPDATE 연산을 수행하기 전에 락을 설정해야 한다
- 트랜잭션 종료 전에 락을 해제해야 한다

## • Serializable

- 트랜잭션들이 중첩되어 실행되는 것과 순차적으로 실행되는 것의 결과가 같을 때를 말함
- Serializable은 two-phase locking protocol에 의해 보장

## • 동시성 제한 수준

- RR(Repeatable Read)
  - 트랜잭션들의 실행 순서가 모두 serializable한 제한 수준
  - 하나의 레코드에 대한 락이 트랜잭션이 끝날 때까지 유지
  - two-phase locking protocol와 매우 유사
- CS(Cursor Stability)
  - 트랜잭션이 끝날 때까지 락이 유지되지 않고, 하나의 레코드에 대한 연산이 수행될 때 락이 설정되고, 해당 레코드에 대한 연산이 끝나면 설정된 락이 제거되는 제한 수준
  - two-phase locking protocol을 따르지 않기 때문에 serializability 보장 X

## • 바람직하지 않은 세 가지 연산

- Dirty Read
  - 트랜잭션 A가 x값을 x'으로 변경한 다음 트랜잭션 B가 x'을 읽은 상황에서 트랜잭션 A가 ROLLBACK하는 상황
  - 동시성 문제 2 : The Uncommitted Dependency Problem 참고

- Non-repeatable read
  - 트랜잭션 A가 x를 여러 번 조회하는 중에 트랜잭션 B가 x의 값을 변경하여 트랜잭션 A에서 x의 값들이 서로 다르게 나오는 상황
- Phantom read
  - 트랜잭션 A가 레코드를 여러 번 조회하는 중에 트랜잭션 B가 새로운 레코드를 추가하여 존재하지 않던 레코드가 갑자기 나오는 상황
  - 테이블 단위의 락 설정, 인덱스에 해당하는 필드에 락을 설정하는 방법으로 해결 가능

## [ 트랜잭션 격리 수준 ]

- 격리 수준 : 동시에 여러 트랜잭션이 처리 될때 트랜잭션끼리 얼마나 서로 고립되어 있는지 나타내는 것
- 아래로 내려 갈 수록 고립 정도가 낮아지고, 성능은 높아지는 것이 일반적

Isolation level	Dirty read	Nonrepeatable read	Phantom read
SERIALIZABLE	N	N	N
REPEATABLE READ	N	N	Y
READ COMMITTED	N	Y	Y
READ UNCOMMITTED	Y	Y	Y

- READ UNCOMMITTED
  - 어떤 트랜잭션의 변경 내용이 COMMIT이나 ROLLBACK과 상관 없이 다른 트랜잭션에서 조회 가능한 수준
  - 데이터 정합성에 문제가 많고, RDBMS 표준에서는 격리 수준으로 인정 X
- READ COMMITTED
  - 어떤 트랜잭션의 변경 내용이 COMMIT되어야만 다른 트랜잭션에서 조회 가능한 수준
  - Oracle에서 기본적으로 사용하며, 온라인 서비스에서 가장 많이 선택되는 격리 수준
  - Nonrepeatable read 발생 : 하나의 트랜잭션 내에서 같은 SELECT를 여러 번 수행할 때 같은 결과가 나오지 않을 가능성 존재
  - 일반적인 웹 어플리케이션에서는 크게 문제가 되지 않음
  - 금전적인 처리 관련 업무(입금/출금 처리)에서는 적합하지 않은 격리 수준

- REPETABLE READ

- 트랜잭션이 시작되기 전에 커밋된 내용에 대해서만 조회할 수 있는 격리 수준
- 자신의 트랜잭션 번호보다 낮은 트랜잭션(먼저 실행되는) 번호에서 변경(+커밋)된 레코드들만 볼 수 있는 것
- MySQL에서 사용하는 격리 수준
- Nonrepeatable read가 발생하지 않는 격리 수준
- 트랜잭션이 시작되는 시점의 데이터에 대한 일관성을 보장해야 하기 때문에 한 트랜잭션의 실행시간이 길어질 수록 해당 시간만큼 멀티 버전을 관리해야 하는 단점 존재

- Serializable

- 가장 단순하고 가장 엄격한 격리 수준
- 읽기 작업에서도 공유 잠금을 설정하여 동시에 다른 트랜잭션에서 해당 레코드 변경이 불가능

