7. 데이터 입력 / 수정 / 삭제 – INSERT, UPDATE, DELETE문

SQL의 DML문은 INSERT(입력) , UPDATE(수정), DELETE (삭제), MERGE(조건에 따라 INSERT와 UPDATE 동시에 수행) 의 기능으로 분류됩니다.

### INSERT 문

## 특징

* 기본적으로 하나의 INSERT 문장은 한 개의 ROW 입력
* INSERT 구문 종류에 따라 한 INSERT 문장으로 여러 개의 ROW를 동시에 입력 가능
* 구문형식

INSERT INTO 테이블명 (컬럼1, 컬럼2, …) VALUES (값1, 값2, ….)

* 컬럼1, 컬럼2와 값1, 값2의 개수, 순서, 데이터 형태 일치
* 테이블 명 다음 (컬럼1, 컬럼2, …) 부분은 생략 가능, 생략 시 모든 컬럼 값 입력
* NOT NULL 속성인 컬럼은 반드시 입력해야 함

### INSERT 구문 연습 1.1

**INSERT INTO** EMP ( emp\_no, emp\_name, salary, hire\_date)

**VALUES** (1, '홍길동', 1000, '2020-06-01');

※ hire\_date는 date 형이지만 묵시적 형변환이 적용되어 문자형 값인 '2019-01-01'이 날짜로 자동 변환

### INSERT 구문 연습 1.2

**INSERT INTO** EMP ( emp\_no, emp\_name)

**VALUES** (2, '김유신');

※ 테이블의 일부 컬럼만 선정해 입력 가능 (나머지 부분은null 값으로 출력)

### INSERT 구문 연습 1.3

**INSERT INTO EMP** ( emp\_name, emp\_no )

**VALUES** ('강감찬', 3);

※ 테이블 생성 시 컬럼 순서대로 입력할 필요없고, 입력하려는 컬럼과 입력된 값의 순서만 맞추면 OK

### INSERT 구문 연습 1.4

**INSERT INTO** EMP

**VALUES** (4, '세종대왕', 1000, SYSDATE);

※ 컬럼명 생략 시, VALUES 절에는 테이블의 모든 컬럼에 입력될 값을 명시해야 함. 입력 순서는 테이블 생성 시 기술한 컬럼 순서

### INSERT 구문 연습 1.5

**INSERT INTO** EMP ( emp\_no, salary, hire\_date)

**VALUES** (5, 1000, SYSDATE);

※ emp\_name 컬럼은 Not Null 컬럼, 따라서 반드시 입력해야 하는데, 누락해서 오류 발생

### INSERT 구문 연습 1.6

**INSERT INTO** EMP

**VALUES** (4, '신사임당', 1000, SYSDATE);

※ emp\_no에 4를 입력했으나, 이미 입력되어 있음. 기본 키 컬럼은 중복 값을 허용하지 않음

### INSERT 구문 연습 1.7

**INSERT INTO** EMP

**VALUES** (5, '신사임당', 1000, **TO\_DATE**('2020-06-29 19:54:30', 'YYYY-MM-DD HH24:MI:SS'));

※ hire\_date 입력 시, TO\_DATE 함수를 사용해 정확한 날짜 형식을 주고 입력 (문자형 > 숫자형)

### INSERT 구문 2

* 구문형식

**INSERT INTO** 테이블명 (컬럼1, 컬럼2, …)

**SELECT** exp1, exp2, ..

**FROM** …

* 한 번 실행 시 여러 개의 ROW 입력 가능 -> SELECT 문이 반환하는 데이터에 따라 좌우
* 컬럼1, 컬럼2, …와 exp1, exp2, … 는 개수, 순서, 데이터 형이 일치
* 테이블명 다음 (컬럼1, 컬럼2, …) 부분은 생략 가능, 생략 시 모든 컬럼 값 입력
* NOT NULL 속성인 컬럼은 반드시 입력

### INSERT 구문 연습 2.1

**INSERT INTO** EMP

**SELECT** emp\_no + 10, emp\_name, salary, hire\_date

**FROM** EMP;

※ 기존에 입력된 5건을 select 해 다시 입력. 단, emp\_no 기본 키 컬럼 중복 값 입력 방지를 위해 기존 값에 + 10 해서 입력 (1~5, 11~15)

### INSERT 구문 연습 2.2

TRUNCATE TABLE EMP; ## EMP TABLE 전체 삭제

**INSERT INTO** EMP

**SELECT** employee\_id, first\_name || ' ' || last\_name, salary, hire\_date

**FROM** EMPLOYEES

WHERE department\_id = 90;

※ employees 테이블에서 부서번호가 90번인 사원의 데이터를 조회해 emp 테이블에 입력

### INSERT 구문 연습 2.3

**INSERT** INTO EMP

**SELECT** employee\_id, first\_name || ' ' || last\_name, salary, hire\_date

**FROM** employees;

※ employees 테이블에서 90번 부서 사원 이미 입력. 다시 전체 사원 입력을 시도하니 기본키인 emp\_no 중복 값 오류 발생 (무결성 제약조건 (HR.EMP\_PK)

### INSERT 구문 연습 2.4

실습용 EMP\_INFO1 테이블 생성

**INSERT INTO** EMP\_INFO1

**SELECT** a.employee\_id,

a.first\_name || ' ' || a.last\_name,

a.salary, a.hire\_date, b.department\_name,

d.country\_name

**FROM** employees a, departments b, locations c, countries d

**WHERE** a.department\_id = b.department\_id

AND b.location\_id = c.location\_id

AND c.country\_id = d.country\_id;

※ 조인을 사용한 쿼리를 사용해 그 결과를 emp\_info1 테이블에 입력

### INSERT 구문 3

* 한 번 실행 시 여러 테이블에 동시 INSERT
* 컬럼과 값의 쌍 개수, 순서 데이터 형이 맞아야 함
* 입력하고자 하는 컬럼은 조정 가능
* 실제 사용하는 경우는 별로 없음
* 구문형식

- **INSERT ALL**

**INTO** 테이블명1 (컬럼1, 컬럼2, …)

VALUES ( 값1, 값2, …)

**INTO** 테이블명2 (컬럼1, 컬럼2, …)

VALUES ( 값1, 값2, …)

….

**SELECT** exp1, exp2, ..

**FROM** …

SELECT 절을 INTO에 넣어서 각각의 테이블명1,2의 결과값을 도출한다.

### INSERT 구문 3.1

EMP1, 2, 3 테이블을 미리 생성한다.

**INSERT ALL**

**INTO** **EMP1** (emp\_no, emp\_name, salary, hire\_date)

VALUES (emp\_no, emp\_name, salary, hire\_date)

**INTO** **EMP2** (emp\_no, emp\_name, salary, hire\_date)

VALUES (emp\_no, emp\_name, salary, hire\_date)

**SELECT** employee\_id emp\_no,

first\_name || ' ' || last\_name emp\_name, salary, hire\_date

**FROM** employees;

※ SELECT 절을 INTO에 넣어서 각각의 EMP1, 2의 결과값을 도출한다.

### INSERT 구문 3.2

**INSERT ALL**

**INTO** **EMP1** (emp\_no, emp\_name, salary, hire\_date)

VALUES (emp\_no, emp\_name, salary, hire\_date)

**INTO** **EMP2** (emp\_no, emp\_name, salary, hire\_date)

VALUES (emp\_no, emp\_name, salary, hire\_date)

**INTO** **EMP3** (emp\_no, emp\_name)

VALUES (emp\_no, emp\_name, salary, hire\_date)

**SELECT** employee\_id emp\_no,

first\_name || ' ' || last\_name emp\_name, salary, hire\_date

**FROM** employees;

※ SELECT 절을 INTO에 넣어서 각각의 EMP1, 2, 3의 결과값을 도출한다.

EMP3에는 salary, hire\_date가 없음으로 SELECT 문이 적용이 되면 salary, hire\_date는 null값으로

뜬다.

### INSERT 구문 4.1 (구문3 + WHEN 조건)

* WHEN 조건을 체크해 조건이 맞으면 INSERT
* WHEN 조건과 INTO 절을 여러 개 명시할 수 있음
* 한 번 실행 시 여러 테이블에 동시 INSERT
* ELSE 절 추가 가능
* INSERT4 구문형식

**INSERT ALL**

**WHEN** 조건1 **THEN**

**INTO** 테이블명1 (컬럼1, 컬럼2, …)

**VALUES** ( 값1, 값2, …)

**WHEN** 조건2 **THEN**

**INTO** 테이블명2 (컬럼1, 컬럼2, …)

**VALUES** ( 값1, 값2, …)

**ELSE INTO** …

….

**SELECT** exp1, exp2, ..

**FROM** …

### INSERT 구문 4.2 (구문4.2 – ALL + FIRST)

**INSERT FIRST**

**WHEN** 조건1 **THEN**

**INTO** 테이블명1 (컬럼1, 컬럼2, …)

**VALUES** ( 값1, 값2, …)

**WHEN** 조건2 **THEN**

**INTO** 테이블명2 (컬럼1, 컬럼2, …)

**VALUES** ( 값1, 값2, …)

**ELSE INTO** …

….

**SELECT** exp1, exp2, ..

**FROM** …

* · ALL 대신 FIRST 사용
* 첫 번째 WHEN 조건을 만족하면 이후 INTO 절 수행
* 첫 번째 조건을 만족한 데이터(ROW)가 두 번째 조건을 만족하더라도 두 번째 테이블에는 INSERT 되지 않음
* CASE 표현식과 동작 방식 흡사

TRUNCATE TABLE emp1; TRUNCATE TABLE emp2; TRUNCATE TABLE emp3;

**INSERT ALL**

**WHEN** **dept\_id = 20** **THEN**

**INTO** EMP1 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**WHEN** **dept\_id BETWEEN 30 AND 50** **THEN**

**INTO** EMP2 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**WHEN** **dept\_id > 50** THEN

**INTO** EMP3 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**SELECT** employee\_id emp\_no, first\_name || ' ' || last\_name emp\_name, salary, hire\_date, department\_id dept\_id

**FROM** employees;

※ EMP1은 DEPT\_ID가 20인 데이터만 추출, EMP2은 DEPT\_ID가 30 ~ 50 사이인 데이터만 추출, EMP3은 DEPT\_ID가 50 이상인 데이터만 추출

TRUNCATE TABLE emp1; TRUNCATE TABLE emp2; TRUNCATE TABLE emp3;

**INSERT ALL**

**WHEN** **salary >= 2500** **THEN**

**INTO** EMP1 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**WHEN** **salary >= 5000** **THEN**

**INTO** EMP2 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**WHEN** **salary >= 7000** THEN

**INTO** EMP3 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**SELECT** employee\_id emp\_no, first\_name || ' ' || last\_name emp\_name, salary, hire\_date, department\_id dept\_id

**FROM** employees;

SELECT MIN(salary), MAX(salary) FROM EMP1; (2500, 24000)

SELECT MIN(salary), MAX(salary) FROM EMP2; (5800, 24000)

SELECT MIN(salary), MAX(salary) FROM EMP3; (7000, 24000)

## FIRST 기재된 구문

**INSERT FIRST**

**WHEN** **salary >= 2500** **THEN**

**INTO** EMP1 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**WHEN** **salary >= 5000** **THEN**

**INTO** EMP2 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**WHEN** **salary >= 7000** THEN

**INTO** EMP3 (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**VALUES** (emp\_no, emp\_name, salary, hire\_date, dept\_id)

**SELECT** employee\_id emp\_no, first\_name || ' ' || last\_name emp\_name, salary, hire\_date, department\_id dept\_id

**FROM** employees;

SELECT MIN(salary), MAX(salary) FROM EMP1; (2500, 24000)

SELECT MIN(salary), MAX(salary) FROM EMP2; (null, null)

SELECT MIN(salary), MAX(salary) FROM EMP3; (null, null)

※ employees 테이블의 데이터가 첫 번째 조건을 만족하므로 EMP1 테이블에만 데이터 INSERT

**### UPDATE 구문**

* 테이블에 저장된 데이터를 수정하는 문장
* 컬럼 값을 수정, 조건에 따라 여러 개의 ROW 처리 가능
* 한 번 실행으로 여러 개의 컬럼 값, 여러 개의 ROW 처리 가능
* 어떤 ROW를 수정할 것인지는 WHERE 절에서 처리
* 변경 하려는 컬럼과 변경값은 데이터 형이 맞아야 함
* 변경값 항목에는 표현식, 서브쿼리도 사용 가능
* WHERE 조건을 만족하는 ROW 만 처리됨, WHERE 절 생략 시 전체 ROW에 대해 컬럼

값 변경

* 구문형식

**UPDATE** 테이블명

**S**E**T** 컬럼1 = 변경값1,

컬럼2 = 변경값2,

…

**WHERE** 조건

## UPDATE문 1.1

SELECT \*

FROM EMP;

**UPDATE** emp

**SET** salary = 0

**WHERE** salary < 20000;

※ Emp 테이블의 salary 컬럼을 0으로 바꾸겠다. (salary가 20000 미만인 데이터만 적용)

## UPDATE문 1.2

ALTER TABLE emp

ADD retire\_date DATE ;

**UPDATE** emp

**SET** retire\_date = SYSDATE

**WHERE** emp\_no = 102;

SELECT \*

FROM EMP;

※ emp에 retire\_date 컬럼을 추가하고, retire\_date는 SYSDATE로 표기한다. (단, emp\_no=102)

## UPDATE문 1.2

**UPDATE** EMP\_INFO1

**SET** emp\_name = emp\_name || '(middle)'

**WHERE** salary BETWEEN 10000 AND 20000;

SELECT \*

FROM EMP\_INFO1

WHERE INSTR(emp\_name, 'middle') > 0 ;

EMP\_INFO1 테이블을 수정하는데, emp\_name 컬럼명을 emp\_name || '(middle)로 수정하고 salary 컬럼에서 10000~20000사이의 값을 가진 값을 출력한다.

EMP\_NAME에 middle이라는 글자를 찾아라

-- update 확인

SELECT \*

FROM EMP\_INFO1

WHERE INSTR(emp\_name, 'middle') > 0

AND salary NOT BETWEEN 10000 AND 20000;

역으로 구문을 작성해서 위에 작성한 구문이 제대로 업데이트 되었는지 확인필요

## UPDATE문 1.3

**UPDATE** EMP\_INFO1

**SET** emp\_name = emp\_name || ' (1)'

,department\_name = department\_name || ' (1)'

WHERE hire\_date < TO\_DATE('2005-01-01', 'YYYY-MM-DD');

SELECT \*

FROM EMP\_INFO1

WHERE INSTR(department\_name, '(1)') > 0 ;

EMP\_INFO1 를 수정하는데 emp\_name, department\_name컬럼을 각각 emp\_name || ' (1)', department\_name || ' (1)'로 수정하는데 '2005-01-01'보다 작은 것을 출력한다.

department\_name, '(1)' 값이 기재되어 있는 데이터 전부 출력

## UPDATE문 1.4

SELECT \*

FROM EMP1

WHERE dept\_id IS NULL; ## kimberely Grant

**UPDATE** EMP1

**SET** dept\_id = **( SELECT MAX(department\_id)**

**FROM DEPARTMENTS**

**WHERE manager\_id IS NULL**

)

WHERE dept\_id IS NULL; ##1

SELECT \*

FROM EMP1

WHERE emp\_no = 178;

Dept\_id에서 NULL인 값인 것 중에서, Departments 테이블의 manager\_id 컬럼이 NULL인데 department\_id가 MAX인 값을 출력한다. (단일값 출력)

### DELETE 문 (삭제)

* ROW 단위로 삭제됨
* WHERE 절 조건에 맞는 ROW가 삭제
* WHERE 절 생략 시 테이블에 있는 모든 ROW 삭제
* **DELETE [FROM]** 테이블명

**WHERE** 조건

* FROM 은 생략 가능

## DELETE문 1.1

SELECT \*

FROM emp;

**DELETE** emp

**WHERE** emp\_no in (101, 102);

※ emp\_no가 101, 102인 데이터 삭제

## DELETE문 1.2

**DELETE** emp1

**WHERE** emp\_name LIKE 'J%';

SELECT \*

FROM emp1

ORDER BY emp\_name;

COMMIT;

Emp\_name에서 J로 시작하는 데이터를 삭제해라

* INSERT가 다른 구문에 비해 부하가 그나마 덜 걸린다. (데이터를 넣기만 하면 되니까)

## 6. 테이블 생성과 데이터 복사를 동시에 진행

* 구문형식

**CREATE TABLE** 테이블명 **AS**

**SELECT** \*

**FROM** 복사대상테이블;

* 테이블이 생성됨과 동시에 SELECT 문이 반환하는 데이터도 함께 입력됨
* DDL로 COMMIT 이 필요 없음

## 예시 1

**CREATE TABLE** employees\_copy **AS**

**SELECT** \*

**FROM** employees;

SELECT \*

FROM employees\_copy;

Employees 테이블과 동일한 employees\_copy가 복사가 되면서 동시에 테이블이 생성됨

**## 트랜잭션 (Transaction) 처리 (거래)**

* SQL에서는 COMMIT, ROLLBACK 문장으로 트랜잭션 처리

거래 성공 -> COMMIT : 변경된 데이터 최종 저장

거래 실패 -> ROLLBACK : 변경 이전 상태로 회귀

* INSERT, UPDATE, DELETE, MERGE 문 실행 후 오류 없을 경우 반드시 COMMIT 문 실행
* 데이터 가공 작업 실패나 기타 사유 (예, WHERE 절 없이 DELETE 했을 경우)로 인해 작업 전 상태로 가고 싶다면 ROLLBACK 문 실행

INSERT INTO 테이블1 … ;

**COMMIT**; ## 테이블 1에 대한 INSERT 작업 반영

UPDATE 테이블1 SET 컬럼1 = 값

WHERE …;

**ROLLBACK**;

* **ROLLBACK은 COMMIT이 된 이후의 작업부터 적용이 됨. (**INSERT INTO 테이블1 부분이 원복되지 않음.

예시 1.

DELETE emp\_tran

WHERE dept\_id = 90;

**COMMIT;**

SELECT \* FROM emp\_tran;

UPDATE emp\_tran

SET emp\_name = 'HAHA'

WHERE dept\_id = 60;

**ROLLBACK;**

SELECT \* FROM emp\_tran;

Dept\_id가 90인 것을 삭제하고, dept\_id = 60인 emp\_name을 ‘HAHA’로 수정, ROLLBACK을 하면 다시 이전의 기존값으로 원상복귀한다.

### MERGE문

* INSERT와 UPDATE를 한 번에 처리
* 대상 테이블에 대해 조건에 따라 INSERT 나 UPDATE 를 수행
* 일반적으로 테이블의 주요 키 값을 체크, 해당 값이 존재하면 UPDATE, 존재하지 않으면 INSERT 수행
* 구문형식

**MERGE INTO** 대상테이블명

**USING** 참조테이블 or 서브쿼리

**ON** 조인조건

**WHEN MATCHED THEN**

**UPDATE** **SET** 컬럼1 = 값1, 컬럼2 = 값2, …

**WHEN NOT MATCHED THEN**

**INSERT** (컬럼1, 컬럼2, …)

**VALUES** (값1, 값2, …);

INSERT할 대상테이블 명을 넣고 참조테이블 or 서브쿼리에서 나오는 값을 대상 테이블 명에 넣는 구조

### 예제 1

우선 테이블을 복제한다.

**CREATE TABLE** dept\_mgr **AS**

SELECT \*

FROM departments;

ALTER TABLE dept\_mgr

ADD CONSTRAINTS dept\_mgr\_pk PRIMARY KEY (department\_id); SELECT \* FROM dept\_mgr;

**## 예제 2**

**MERGE** **INTO** dept\_mgr a

**USING** ( SELECT 280 dept\_id, '영업부(Merge)' dept\_name

FROM dual

UNION ALL

SELECT 285 dept\_id, '경리부(Merge)' dept\_name

FROM dual

) **b**

**ON** ( a.department\_id = b.dept\_id )

**WHEN MATCHED THEN** ## ON 조건에 만족하는 건이 있으면

**UPDATE** **SET** a.department\_name = b.dept\_name

**WHEN NOT MATCHED THEN** ## 일치하는 건이 없으면

**INSERT** (a.department\_id, a.department\_name)

**VALUES** (b.dept\_id, b.dept\_name);

SELECT \* FROM dept\_mgr;

USING은 가상의 서브쿼리를 작성한 것이고 USING과 MERGE INTO를 조인할 조인이 ON이다.

Dept\_mgr에 만족하는 갚이 있으면 UPDATE SET 적용, 아니면 INSERT 적용 (280 dept\_id, '영업부

(Merge)' dept\_name)의 값을 넣어라.

결론적으로는 일치하는 건이 없기 때문에 280 dept\_id, '영업부(Merge)

285 dept\_id, '경리부(Merge) 가 출력됨.

**MERGE** **INTO** dept\_mgr a

**USING** ( SELECT **280** dept\_id, '**영업부(Merge)2**' dept\_name

FROM dual

UNION ALL

SELECT **285** dept\_id, '**경리부(Merge)2**' dept\_name

FROM dual

) **b**

**ON** ( a.department\_id = b.dept\_id )

**WHEN MATCHED THEN** ## ON 조건에 만족하는 건이 있으면

**UPDATE** **SET** a.department\_name = b.dept\_name

**WHEN NOT MATCHED THEN** ## 일치하는 건이 없으면

**INSERT** (a.department\_id, a.department\_name)

**VALUES** (b.dept\_id, b.dept\_name);

SELECT \* FROM dept\_mgr;

### 뷰

* 하나 혹은 그 이상의 다른 테이블이나 뷰로 구성된 논리적 객체 ( 테이블처럼 동작 )
* 뷰 자체에는 데이터가 저장되어 있지 않음
* 하나의 뷰가 또 다른 뷰에서 사용 될 수 있음

뷰의 용도 - 테이블 데이터 보안 강화 -> 컬럼이나 ROW 접근 제한

데이터 복잡성 숨김 -> 복잡하게 얽힌 쿼리를 뷰로 만들어 사용

테이블 구조 변경에 따른 영향도 감소 -> 신규 컬럼 추가 시에도 영향 받지 않음

* 구문 유형

1. 뷰 생성 : CREATE OR REPLACE VIEW 뷰이름 AS SELECT 문; ·
2. 뷰 수정 : CREATE OR REPLACE VIEW 뷰이름 AS SELECT 문; ·
3. 뷰 삭제 : DROP VIEW 뷰이름;

## 예시 1

**CREATE OR REPLACE VIEW** emp\_dept\_v **AS**

SELECT a.employee\_id,

a.first\_name || ' ' || a.last\_name emp\_names,

**a.salary**,

b.department\_id ,b.department\_name

FROM employees a, departments b

WHERE a.department\_id = b.department\_id

ORDER BY 1; SELECT \* FROM emp\_dept\_v;

SELECT \*

FROM emp\_dept\_v;

정상적인 구문인데 salary넣고 뺄 수 있고, 보여주는 기능으로만 가능하다.

회사 중요정보 같은 경우를 외부에 보여줄 때 **a.salary** 같은 부분만없애고 View로 볼 수 있게 한다.

* 사용자는 emp\_dept\_v2 뷰의 조회권한을 hr2에게 부여

GRANT SELECT ON emp\_dept\_v2 TO hr2; ·

* hr2 사용자로 로그인 한 후, emp\_dept\_v2 조회

SELECT \* FROM emp\_dept\_v2; (X)

SELECT \* FROM hr.emp\_dept\_v2; (O)

* hr이 생성한 emp\_dept\_v2 뷰를 hr이 아닌 다른 사용자가 참조하려면 소유자명.객체명으로 접근

### 데이터 딕셔너리

오라클에서 제공하는 데이터베이스 객체(사용자, 테이블, 뷰 등)에 대한 메타정보를 담은 뷰

접두어로 용도 구분

- DBA : 데이터베이스 관리자의 뷰 (모든 사용자 스키마가 포함됨)

- ALL : 현재 로그인한 사용자가 접근할 수 있는 뷰

- USER : 현재 로그인한 사용자가 소유자인 데이터베이스 객체

## 주요 사용자 객체 정보 뷰

- USER\_OBJECTS : 모든 객체 정보

- USER\_TABLES : 테이블 정보

- USER\_INDEXES : 인덱스 정보

- USER\_CONSTRAINTS : 제약조건

- USER\_TAB\_COLS : 테이블과 해당 컬럼 정보

- USER\_VIEWS : 뷰 정보

SELECT \* FROM user\_objects;

※ 데이터 딕셔너리를 통해 데이터베이스 객체에 대한 다양한 정보를 조회할 수 있다.