

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999 (language name became Y2K compliant!)
 - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

Structured Query Language

- Data-definition Language (DDL)
- Data-manipulation Language (DML)
- Integrity
 - SQL DDL (vs. DCL) include commands for specifying integrity constraints
 - Updates that violates integrity constraints are disallowed.
- Transaction control (TCL)
- View definition
- Authorization
- Embedded SQL and dynamic SQL

Data Definition Language

- Allows the specification of not only a set of relations but also information about each relation, including:
 - The schema for each relation.
 - The domain of values associated with each attribute.
 - Integrity constraints
 - The set of indices to be maintained for each relations.
 - Security and authorization information for each relation.
 - The physical storage structure of each relation on disk.

DDL

Data Definition Language

- Create Table, Drop Table , Truncate Table, Alter Table
- Data Type
- Constraint (NOT NULL, DEFAULT, CHECK, REFERENCE)

DDL 요약

- CREATE TABLE: 테이블 생성
- ALTER TABLE: 테이블 관련 변경
- DROP TABLE: 테이블 삭제
- RENAME: 이름 변경
- TRUNCATE: 테이블의 모든 데이터 삭제
- COMMENT: 테이블에 설명 추가

테이블 생성

- CREATE TABLE문 이용
- 테이블이름, 컬럼 이름, 데이터 타입 등 정의

```
CREATE TABLE book (  
    bookno NUMBER(5),  
    title VARCHAR2(50),  
    author VARCHAR2(10),  
    pubdate DATE  
);
```



bookno	title	author	pubdate
1	토지	박경리	2005-03-12
2	슬램덩크	다케이코	2006-04-05
...

Subquery를 이용한 테이블 생성

- Create a table with the same schema as an existing table:

create table *temp_account* **like** *account*

- Subquery의 결과와 동일한 테이블 생성됨
- 질의 결과 레코드들이 포함됨
- NOT NULL 제약 조건 만 상속됨

```
CREATE TABLE empSALES
AS
    SELECT * FROM emp
    WHERE job = 'SALES';
```

TABLE 종류

- User Tables:
 - Are a collection of tables created and maintained by the user
 - Contain user information
- Data Dictionary:
 - Is a collection of tables created and maintained by the Oracle Server
 - Contain database information

기본 데이터 타입

Data type	Description
VARCHAR2(size)	가변길이 문자열 (최대 4000byte)
CHAR(size)	고정길이 문자열 (최대 2000byte)
NUMBER(p,s)	가변길이 숫자. 전체 p자리 중 소수점 이하 s자리 (p:38, s:-84~127, 21Byte) 자리수 지정 없으면 NUMBER(38)
DATE	고정길이 날짜+시간, 7Byte

- 참고
 - VARCHAR2와 CHAR의 차이점
 - INT, FLOAT 등의 ANSI Type도 내부적으로 NUMBER(38)로 변환됨

Data type	Description
NCHAR(size)	national character set에 따라 결정되는 size 만큼의 고정길이 character data로 최대 2000byte까지 가능. 디폴트는 1 character.
NVARCHAR2 (size)	national character set에 따라 결정되는 size 만큼의 가변길이 character data로 최대 4000 byte까지 가능하며 반드시 길이를 정해 주어야 함.
LONG	가변 길이character data로 최대 2 gigabyte까지 가능.
RAW (size)	가변 길이raw binary data로 최대 2000 까지 가능하며 반드시 길이를 주어야 함.
LONG RAW	가변길이 raw binary data로 최대2 gigabyte까지 가능.
BLOB	Binary data로 4 gigabyte까지 가능.
CLOB	Single-byte character data로 4 gigabyte까지 가능.
NCLOB	national character set까지 포함한 모든 character data로 4 gigabyte까지 가능.
BFILE	외부 화일로 저장된 binary data로 4 gigabyte까지 가능.
ROWID	Row의 물리적 주소를 나타내는 binary data로 extended rowid 는 10 byte, restricted rowid는 6 byte 길이.
TIMESTAMP	Date값을 미세한 초 단위까지 저장. NLS_TIMESTAMP_FORMAT 형식으로 처리.
INTERVAL YEAR TO MONTH	두 datetime 값의 차이에서 YEAR와 MONTH값 만을 저장.
INTERVAL DAY TO SECOND	두 datetime 값의 ₁ 차이를 DAY, HOUR, MINUTE, SECOND 까지 저

ALTER TABLE

- 컬럼 추가
 - ALTER TABLE book **ADD** (pubs VARCHAR2(50));
- 컬럼 수정
 - ALTER TABLE book **MODIFY** (title VARCHAR2(100));
- 컬럼 삭제
 - ALTER TABLE book **DROP** author;
- UNUSED 컬럼
 - ALTER TABLE book **SET UNUSED** (author);
ALTER TABLE book **DROP UNUSED COLUMNS**;

기타 테이블 관련 명령

- 테이블 삭제
 - **DROP TABLE** book;
- 데이터 삭제
 - **TRUNCATE TABLE** book;
- Comment
 - **COMMENT ON TABLE** book **IS** 'this is comment';
- RENAME
 - **RENAME** book **TO** article;
- 주의:
 - ROLLBACK의 대상이 아님!

제약조건

- Constraint
 - Database 테이블 레벨에서 특정한 규칙을 설정해둠
 - 예상치 못한 데이터의 손실이나 일관성을 어기는 데이터의 추가, 변경 등을 예방함
- 종류
 - NOT NULL
 - UNIQUE
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK

제약조건 정의

■ Syntax

```
CREATE TABLE 테이블이름 (  
    컬럼이름 datatype [DEFAULT 기본값] [컬럼제약조건],  
    컬럼이름 datatype [DEFAULT 기본값] [컬럼제약조건],  
    ...  
    [테이블 제약조건] ...);
```

- 컬럼 제약조건: [CONSTRAINT 이름] constraint_type
- 테이블제약조건: [CONSTRAINT 이름] constraint_type(column,..)

■ 주의

- 제약조건에 이름을 부여하지 않으면 Oracle이 Sys-Cn의 형태로 자동 부여

제약조건 (1/3)

■ NOT NULL

- NULL 값이 들어올 수 없음
- 컬럼형태로만 제약조건 정의할 수 있음 (테이블 제약조건 불가)

```
CREATE TABLE book (  
    bookno NUMBER(5) NOT NULL  
);
```

■ UNIQUE

- 중복된 값을 허용하지 않음 (NULL은 들어올 수 있음)
- 복합 컬럼에 대해서도 정의 가능
- 자동적으로 인덱스 생성

```
CREATE TABLE book (  
    bookno NUMBER(5) CONSTRAINT c_emp_u UNIQUE  
);
```

제약조건 (2/3)

■ PRIMARY KEY

- NOT NULL + UNIQUE
(인덱스 자동 생성)
- 테이블 당 하나만 나올 수 있음
- 복합 컬럼에 대해서 정의 가능
(순서 중요)

```
CREATE TABLE book (  
    ssn1 NUMBER(6),  
    ssn2 NUMBER(7),  
    PRIMARY KEY (ssn1,ssn2)  
);
```

■ CHECK

- 임의의 조건 검사. 조건식이 참이어야 변경 가능
- 동일 테이블의 컬럼만 이용 가능

```
CREATE TABLE book (  
    rate NUMBER CHECK (rate IN (1,2,3,4,5))  
);
```


제약조건 (3/3)

■ FOREIGN KEY

- 참조 무결성 제약
- 일반적으로 REFERENCE 테이블의 PK를 참조
- REFERENCE 테이블에 없는 값은 삽입 불가
- REFERENCE 테이블의 레코드 삭제 시 동작
 - **ON DELETE CASCADE:** 해당하는 FK를 가진 참조행도 삭제
 - **ON DELETE SET NULL:** 해당하는 FK를 NULL로 바꿈

```
CREATE TABLE book (  
    ...  
    author_id NUMBER(10),  
    CONSTRAINT c_book_fk FOREIGN KEY (author_id)  
    REFERENCES author(id)  
    ON DELETE SET NULL  
);
```

ADD / DROP CONSTRAINTS

- 제약조건 추가
 - **ALTER TABLE 테이블이름 ADD CONSTRAINT ...**
 - NOT NULL은 추가 못함

```
ALTER TABLE emp ADD CONSTRAINT emp_mgr_fk  
FOREIGN KEY(mgr)REFERENCES emp(empno);
```

- 제약조건 삭제
 - **ALTER TABLE 테이블이름 DROP CONSTRAINT 제약조건이름**
 - PRIMARY KEY의 경우 FK 조건이 걸린 경우에는 CASCADE로 삭제해야 함

```
ALTER TABLE book DROP CONSTRAINT c_emp_u;  
ALTER TABLE dept DROP PRIMARY KEY CASCADE;
```

Data Dictionary

- Oracle이 관리하는 모든 정보를 저장하는 카탈로그
- 내용
 - 모든 스키마 객체 정보, 스키마 객체의 공간 정보, 컬럼의 기본값, 제약조건 정보, 오라클 사용자 정보, 권한 및 롤 정보, 기타 데이터베이스 정보 ...
- Base-Table과 View로 구성됨
 - VIEW의 Prefix
 - USER: 로그인한 사용자 레벨
 - ALL: 모든 사용자 정보
 - DBA: 관리자
- SYS scheme에 속함
- 주의
 - DICTIONARY의 테이블이나 컬럼 이름은 모두 대문자 사용!

Dictionary 예

- 모든 Dictionary 정보

```
SELECT * FROM DICTIONARY
```

- 사용자 스키마 객체 확인 (테이블)

```
SELECT object_name  
FROM user_objects  
WHERE object_type = 'TABLE';
```

- 제약조건 확인 (EMP 테이블의)

```
SELECT constraint_name,  
constraint_type,search_condition  
FROM user_constraints  
WHERE table_name = 'EMP';
```

- 제약조건 컬럼 확인

```
SELECT constraint_name, column_name  
FROM user_cons_columns  
WHERE table_name = 'EMP';
```

Dictionary 예(2/2)

- 모든 사용자 확인

```
SELECT username,  
       default_tablespace,  
       temporary_tablespace  
FROM DBA_USERS;
```

DML

INSERT, DELETE, UPDATE

Data Manipulation Language

- 종류
 - Add new row(s)
 - **INSERT INTO** 테이블이름 [(컬럼리스트)] **VALUES** (값리스트);
 - Modify existing rows
 - **UPDATE** 테이블이름 **SET** 변경내용 [**WHERE** 조건];
 - Remove existing rows
 - **DELETE FROM** 테이블이름 [**WHERE** 조건];
- 트랜잭션의 대상
 - 트랜잭션은 DML의 집합으로 이루어짐.

INSERT

- 묵시적 방법: 컬럼 이름. 순서 지정하지 않음.
테이블 생성시 정의한 순서에 따라 값 지정

```
INSERT INTO dept VALUES (777, 'MARKETING', NULL);
```

- 명시적 방법: 컬럼 이름 명시적 사용. 지정되지 않은 컬럼 NULL 자동 입력

```
INSERT INTO dept(dname, deptno) VALUES ('MARKETING',  
777);
```

- Subquery 이용: 타 테이블로부터 데이터 복사 (테이블은 이미 존재하여야 함)

```
INSERT INTO deptusa  
SELECT deptno, dname FROM dept WHERE country =  
'USA';
```

– 참고: CREATE TABLE AS SELECT는 없는 테이블을 생성 & 데이터 복사

UPDATE

- 조건을 만족하는 레코드를 변경

- 10번 부서원의 월급 100인상 & 수수료 0으로 변경

```
UPDATE emp SET sal = sal + 100, comm = 0  
WHERE deptno = 10;
```

- WHERE 절이 생략되면 모든 레코드에 적용

- 모든 직원의 월급 10%인상

```
UPDATE emp SET sal = sal * 1.1;
```

- Subquery를 이용한 변경

- 담당업무가 'SCOTT'과 같은 사람들의 월급을 부서 최고액으로 변경

```
UPDATE emp SET sal = (SELECT MAX(sal) FROM emp)  
WHERE job = (SELECT job FROM emp WHERE  
ename='SCOTT');
```

DELETE

- 조건을 만족하는 레코드 삭제

```
DELETE FROM emp WHERE ename = 'SCOTT';
```

- 조건이 없으면 모든 레코드 삭제 (주의!)
 - 모든 직원 정보 삭제

```
DELETE FROM emp;
```

- Subquery를 이용한 DELETE
 - 'SALES'부서의 직원 모두 삭제

```
DELETE FROM emp WHERE deptno =  
    (SELECT deptno FROM dept WHERE dname =  
    'SALES');
```

참고

- 데이터 입력, 수정시 자주 사용되는 Pseudo 컬럼
 - USER : Current user name.
 - SYSDATE : Current date and time.
 - ROWID : Location information of rows

```
INSERT INTO emp(eno, hiredate) VALUES (200, SYSDATE);
```

- DEFAULT: default값이 정의된 컬럼에 기본값을 입력할 경우 사용할 수 있음

```
INSERT INTO book VALUES (200, 'Gems', DEFAULT);
```

- DELETE 와 TRUNCATE의 차이점
 - Delete는 Rollback 가능 but 대량의 log 등을 유발하므로 Truncate보다 느림
- 모든 DML문은 Integrity Constraint를 어길 경우 에러 발생

Summary

- DDL: 데이터 정의
 - CREATE TABLE: 테이블 생성
 - ALTER TABLE: 테이블 관련 변경
 - DROP TABLE: 테이블 삭제
 - RENAME: 이름 변경
 - TRUNCATE: 테이블의 모든 데이터 삭제
- DML
 - INSERT INTO ... VALUES ...
 - UPDATE ... SET ...
 - DELETE ... FROM

기본 SELECT

Basic Query Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

select A_1, A_2, \dots, A_n
from r_1, r_2, \dots, r_m
where P

- A_i represents an attribute
 - r_i represents a relation
 - P is a predicate.
- This query is equivalent to the **relational algebra** expression.

$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation.

SELECT

- 데이터베이스에서 원하는 데이터를 검색, 추출
- Syntax
 - **SELECT** [ALL | DISTINCT] 열_리스트
[**FROM** 테이블_리스트]
[**WHERE** 조건]
[**GROUP BY** 열_리스트 [HAVING 조건]]
[**ORDER BY** 열_리스트 [ASC | DESC]];
- 기능
 - Projection: 원하는 컬럼 선택
 - Selection: 원하는 튜플 선택
 - Join: 두개의 테이블 결합
 - 기타: 각종 계산, 정렬, 집계(Aggregation)

SELECT의 기능

Projection

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	80/12/17	880		20
7499	ALLEN	SALESMAN	7698	81/02/20	1760	300	30
7520	WARD	SALESMAN	7698	81/02/22	1375	500	30
7566	JONES	MANAGER	7839	81/04/02	2975		20
7654	MARTIN	SALESMAN	7698	81/09/28	1375	1400	30
7698	BLAKE	MANAGER	7839	81/05/01	2850		10
7782	CLARK	MANAGER	7839	81/06/09	2450		10
7789	SCOTT	ANALYST	7566	87/04/19	3000		20
7839	KING	PRESIDENT		81/11/17	5000		10
7844	TURNER	SALESMAN	7698	81/09/08	1650	0	30
7876	ADAMS	CLERK	7788	87/05/23	1210		20
7900	JAMES	CLERK	7698	81/12/03	1045		30
7902	FORD	ANALYST	7566	81/12/03	3000		20
7934	MILLER	CLERK	7782	82/01/23	1430		10

Selection

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Join

기본 SELECT

- 형식

- SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table;

- 내용

- * : 모든 컬럼 반환
- DISTINCT: 중복된 결과 제거
- SELECT 컬럼명: Projection
- FROM: 대상 테이블
- ALIAS: 컬럼 이름 변경 (as)
- Expression: 기본적인 연산 및 함수 사용 가능

The select Clause

- The **select** clause list the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra

- Example: find the names of all instructors:

```
select name  
from instructor
```

- Find the names of all branches in the *loan* relation

```
select branch-name  
from loan
```

- In the “pure” relational algebra syntax, the query would be:

$\Pi_{\text{branch-name}}(\text{loan})$

- An asterisk in the select clause denotes “all attributes”

```
select *  
from loan
```

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)

- E.g., *Name* \equiv *NAME* \equiv *name*

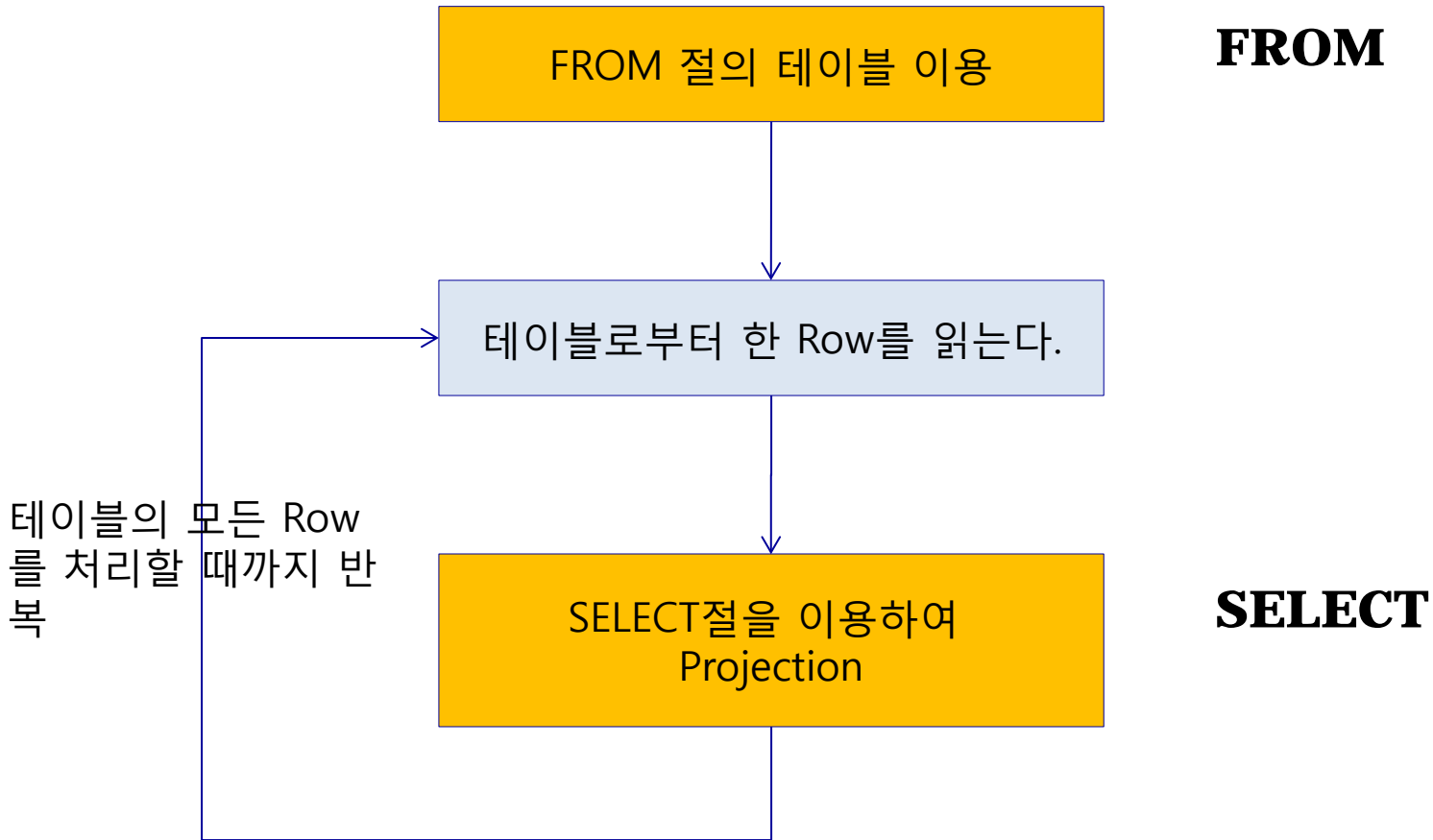
SELECT 예제

- SELECT * FROM emp;
- SELECT ename FROM emp;
- SELECT ename, job FROM emp;
- SELECT ename 이름 FROM emp;

```
SQL> select * from emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	80/12/17	800		20
7499	ALLEN	SALESMAN	7698	81/02/20	1600	300	30
7521	WARD	SALESMAN	7698	81/02/22	1250	500	30
7566	JONES	MANAGER	7839	81/04/02	2975		20
7654	MARTIN	SALESMAN	7698	81/09/28	1250	1400	30
7698	BLAKE	MANAGER	7839	81/05/01	2850		30
7782	CLARK	MANAGER	7839	81/06/09	2450		10
7788	SCOTT	ANALYST	7566	87/04/19	3000		20
7839	KING	PRESIDENT		81/11/17	5000		10
7844	TURNER	SALESMAN	7698	81/09/08	1500	0	30
7876	ADAMS	CLERK	7788	87/05/23	1100		20
7900	JAMES	CLERK	7698	81/12/03	950		30
7902	FORD	ANALYST	7566	81/12/03	3000		20
7934	MILLER	CLERK	7782	82/01/23	1300		10

SELECT, FROM 절 처리방법



실제 모든 SQL이 이렇게 처리되는 것은 아닙니다. SQL의 처리 순서는 DBMS가 질의 최적화 과정을 통하여 결정합니다. 질의의 종류, 데이터의 분포 등에 따라 질의의 실제 순서는 달라질 수도 있습니다.

산술연산

- 기본적인 산술연산 사용 가능
 - +, -, *, /, 부호, 괄호 등
 - 우선순위: 부호, * / , + -
 - 컬럼 이름, 숫자
 - 예
 - SELECT ename, (sal+200) * 12 FROM emp;
 - SELECT ename, -sal * 10 FROM emp;

```
SQL> SELECT ename, (sal+200) * 12 FROM emp;
```

ENAME	(SAL+200)*12
SMITH	12000
ALLEN	21600
WARD	17400
JONES	38100
MARTIN	17400
BLAKE	36600

NULL

- 아무런 값도 정해지지 않았음을 의미
- 어떠한 데이터타입에도 사용가능
- NOT NULL이나 Primary Key 컬럼에는 사용할 수 없음
- NULL을 포함한 산술식은 일반적으로 NULL
 - `SELECT sal, comm, (sal+comm)*12 FROM emp;`
- `NVL(expr1, expr2)`
 - `expr1`이 NULL이면 `expr2`를 출력한다.
 - 데이터타입이 호환 가능 하여야 함.
 - `SELECT sal, comm, (sal+NVL(comm,0))*12 FROM emp;`

Column Alias

- 컬럼의 제목을 변경
- 큰따옴표(" ")을 사용하여 alias내에 공백이나 특수문자를 포함할 수 있다.
- 형태
 - SELECT ename name FROM emp;
 - SELECT ename as name FROM emp;
 - SELECT ename "as" FROM emp;
 - SELECT (sal + comm) "Annual Salary" FROM emp;

```
SQL> select empno no, ename as name, job "to do" from emp;
```

NO	NAME	to do
7369	SMITH	CLERK
7499	ALLEN	SALESMAN
7521	WARD	SALESMAN
7566	JONES	MANAGER
7654	MARTIN	SALESMAN
7698	BLAKE	MANAGER
7782	CLARK	MANAGER
7788	SCOTT	ANALYST
7839	KING	PRESIDENT
7844	TURNER	SALESMAN
7876	ADAMS	CLERK

Literal

- SELECT 절에 사용되는 문자, 숫자, Date 타입 등의 상수
- Date 타입이나 문자열은 작은따옴표 (' ')로 둘러싸야 함
- 문자열 결합(Concatenation) 연산자: ||
- 예
 - SELECT ename, 1000, SYSDATE FROM emp;
 - SELECT 'Name is ' || ename || ' and no is ' || empno FROM emp;

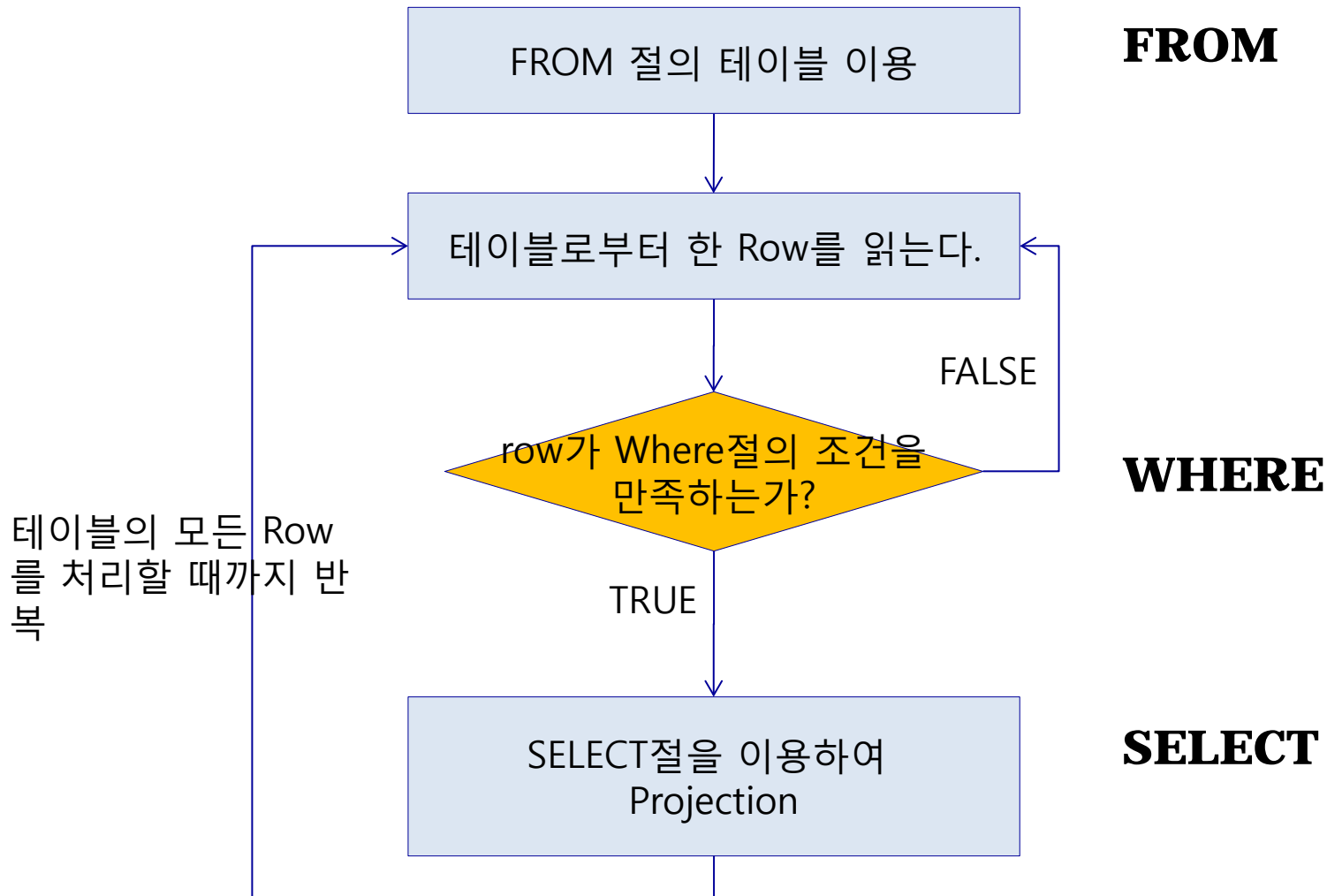
```
SQL> SELECT 'Name is ' || ename || ' and no is ' || empno FROM emp;

'NAMEIS' || ENAME || 'ANDNOIS' || EMPNO
-----
Name is SMITH and no is 7369
Name is ALLEN and no is 7499
Name is WARD and no is 7521
Name is JONES and no is 7566
Name is MARTIN and no is 7654
Name is BLAKE and no is 7698
Name is CLARK and no is 7782
Name is SCOTT and no is 7788
```


WHERE

- 조건을 부여하여 만족하는 ROW Selection
- 연산자
 - =, !=, >, <, <=, >=
 - IN : 집합에 포함되는가?
 - BETWEEN a AND b : a 와 b 사이?
 - LIKE: 문자열 부분 검색
 - IS NULL, IS NOT NULL: NULL인지 검색
 - AND, OR: 둘다 만족? 둘 중 하나만 만족?
 - NOT: 만족하지 않음?
 - ANY, ALL : 집합 중 어느한열, 집합 중 모든 열 (다른 비교연산자와 함께 사용)
 - EXIST: 결과 Row가 하나라도 있나? (subquery에서)

WHERE 절 처리 방법



실제 모든 SQL이 이렇게 처리되는 것은 아닙니다. SQL의 처리 순서는 DBMS가 질의 최적화 과정을 통하여 결정합니다. 질의의 종류, 데이터의 분포 등에 따라 질의의 실제 순서는 달라질 수도 있습니다.

LIKE연산

- Wildcard를 이용한 문자열 부분 매칭
- Wildcard
 - % : 임의의 길이의 문자열 (공백 문자 가능)
 - _ : 한 글자
- Escape
 - **ESCAPE** 뒤의 문자열로 시작하는 문자는 Wildcard가 아닌 것으로 해석
- 예
 - `ename LIKE 'KOR%'` : 'KOR'로 시작하는 모든 문자열(KOR가능)
 - `ename LIKE 'KOR_'` : 'KOR'다음에 하나의 문자가 오는 모든 문자열
 - `ename LIKE 'KOR/%%' ESCAPE '/'` : 'KOR%'로 시작하는 모든 문자열

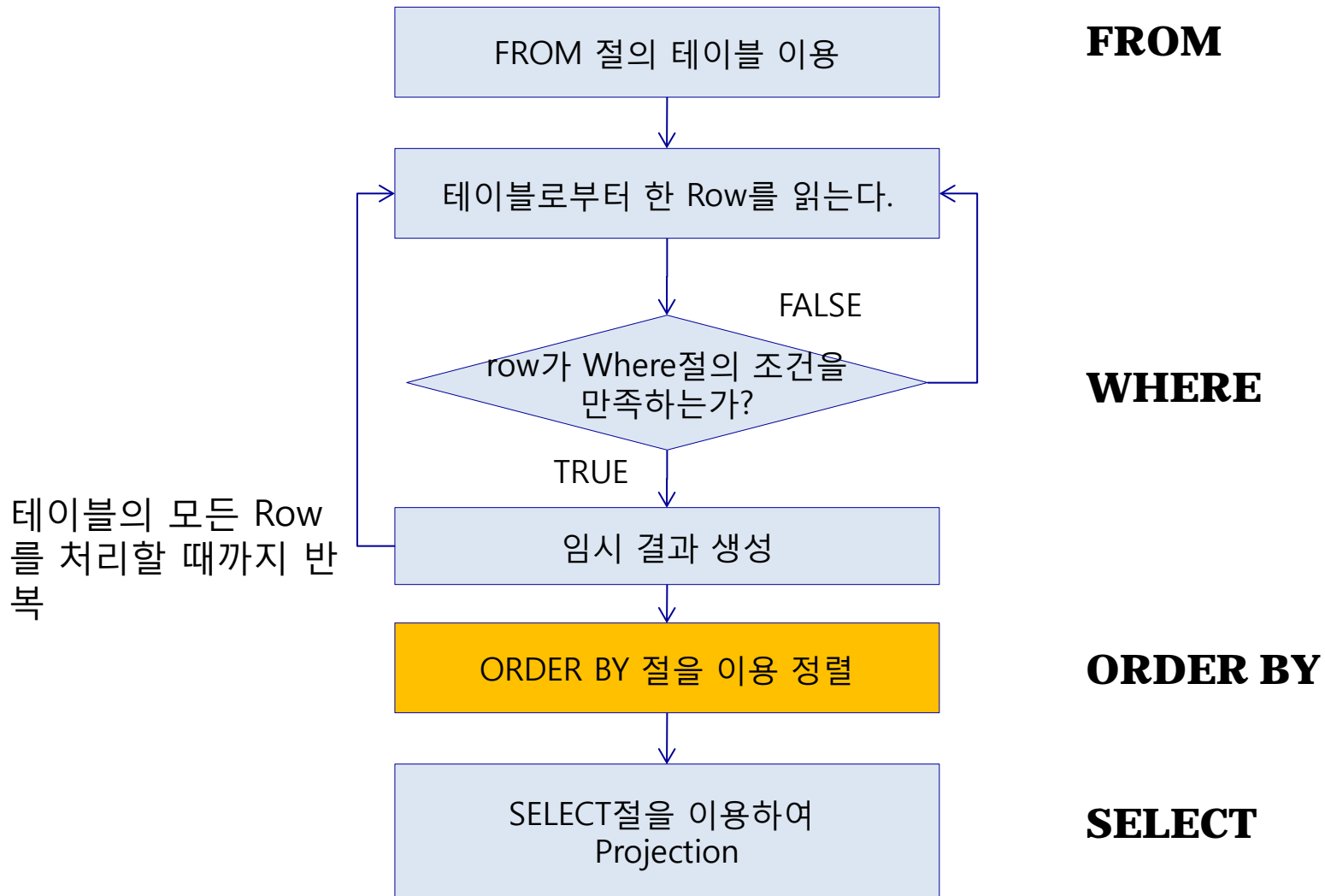
연산자 우선 순위

- ① Arithmetic operators
- ② Concatenation operator
- ③ Comparison conditions
- ④ IS[NOT] NULL, LIKE, [NOT] IN
- ⑤ [NOT] BETWEEN
- ⑥ NOT logical condition
- ⑦ AND logical condition
- ⑧ OR logical condition

ORDER BY

- 주어진 컬럼 리스트의 순서로 결과를 정렬
- 결과 정렬 방법
 - **ASC** : 오름차순 (작은값→큰값) (default)
 - **DESC**: 내림차순(큰값→작은값)
- 여러 컬럼 정의 가능
 - 첫번째 컬럼이 같으면 두번째 컬럼으로, 두번째 컬럼도 같으면...
- 컬럼 이름대신 Alias, expr, SELECT 절상에서의 순서(1, 2, 3...) 도 사용가능
 - 예) `SELECT * FROM emp ORDER BY deptno, sal DESC`
 - 부서번호순으로 정렬하고, sal가 높은 사람부터 출력하시오

ORDER BY절 처리 방법



실제 모든 SQL이 이렇게 처리되는 것은 아닙니다. SQL의 처리 순서는 DBMS가 질의 최적화 과정을 통하여 결정합니다. 질의의 종류, 데이터의 분포 등에 따라 질의의 실제 순서는 달라질 수도 있습니다.

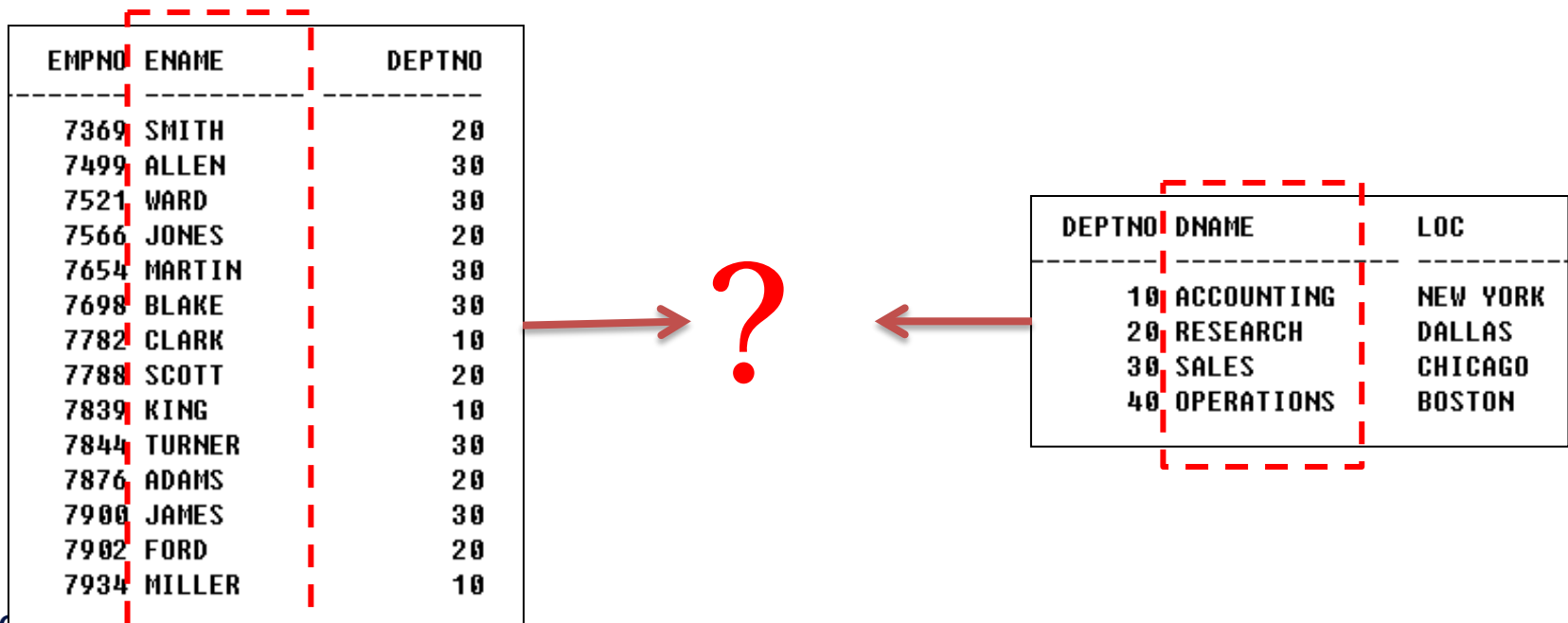
Summary

- SELECT 문 (1)
 - SELECT 절: Projection
 - FROM 절: 대상 테이블
 - WHERE 절: Selection 조건
 - ORDER BY 절: 정렬 조건


JOIN

Join

- 둘 이상의 테이블을 합쳐서 하나의 큰 테이블로 만드는 방법
- 필요성
 - 관계형 모델에서는 데이터의 일관성이나 효율을 위하여 데이터의 중복을 최소화 (정규화)
 - Foreign Key를 이용하여 참조
 - 정규화된 테이블로부터 결합된 형태의 정보를 추출할 필요가 있음
 - 예) 직원의 이름과 직원이 속한 부서명을 함께 보고 싶으면???



Cartesian Product

- 두 테이블에서 그냥 결과를 선택하면? 
 - SELECT ename, dname FROM emp, dept
 - 결과: 두 테이블의 행들의 가능한 모든 쌍이 추출됨
 - 일반적으로 사용자가 원하는 결과가 아님.

- Cartesian Product

$$X \times Y = \{(x, y) | x \in X \text{ and } y \in Y\}$$

- Cartesian Product를 막기 위해서는 올바른 Join조건을 WHERE 절에 부여 해야 함.

ENAME	DNAME
-----	-----
SMITH	ACCOUNTING
ALLEN	ACCOUNTING
WARD	ACCOUNTING
JONES	ACCOUNTING
MARTIN	ACCOUNTING
BLAKE	ACCOUNTING
CLARK	ACCOUNTING
SCOTT	ACCOUNTING
...	
ALLEN	OPERATIONS
WARD	OPERATIONS
JONES	OPERATIONS
MARTIN	OPERATIONS
BLAKE	OPERATIONS
CLARK	OPERATIONS
SCOTT	OPERATIONS
KING	OPERATIONS
TURNER	OPERATIONS
ADAMS	OPERATIONS
JAMES	OPERATIONS
FORD	OPERATIONS
MILLER	OPERATIONS

56 개의 행이 선택되었습니다.

Simple Join

- Syntax

```
SELECT t1.col1, t1.col2, t2.col1 ...  
FROM Table1 t1, Table2 t2  
WHERE t1.col3 = t2.col3
```

- 설명

- FROM 절에 필요로 하는 테이블을 모두 적는다.
- 컬럼 이름의 모호성을 피하기 위해(어느 테이블에 속하는지 알 수 없음)이 있을 수 있으므로 Table 이름에 Alias 사용 (테이블 이름으로 직접 지칭 가능)
- 적절한 Join 조건을 Where 절에 부여 (일반적으로 테이블 개수 -1 개의 조인 조건이 필요)
- 일반적으로 PK와 FK간의 = 조건이 붙는 경우가 많음

Join 처리 방법

Where절의 조인 조건을 이용
From절의 테이블들을 Join하여
임시 테이블을 만든다..

FROM

테이블로부터 한 Row를 읽는다.

FALSE

row가 Where절의 조건을
만족하는가?

WHERE

TRUE

임시 결과 생성

ORDER BY 절을 이용 정렬

ORDER BY

SELECT절을 이용하여
Projection

SELECT

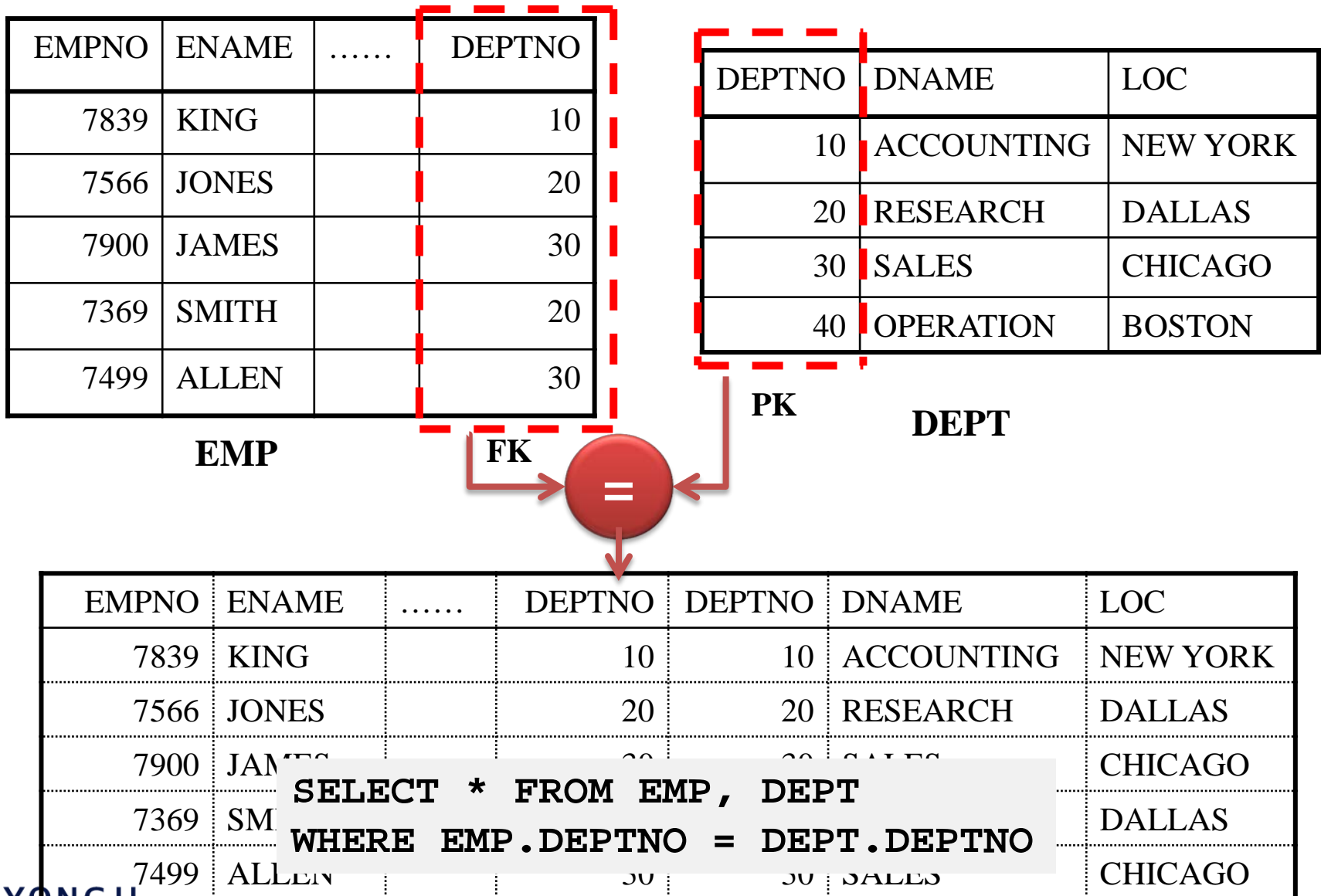
테이블의 모든 Row
를 처리할 때까지 반
복

실제 모든 SQL이 이렇게 처리되는 것은 아닙니다. SQL의 처리 순서는 DBMS가 질의 최적화 과정을 통하여 결정합니다. 질의의 종류, 데이터의 분포 등에 따라 질의의 실제 순서는 달라질 수도 있습니다.

Join 종류

- 용어
 - Cross Join (Cartesian Product): 모든 가능한 쌍이 나타남
 - Inner Join: Join 조건을 만족하는 튜플만 나타남
 - Outer Join: Join 조건을 만족하지 않는 튜플 (짝이 없는 튜플)도 null과 함께 나타남
 - Theta Join: 조건(theta)에 의한 조인
 - Equi-Join: Theta Join & 조건이 Equal (=)
 - Natural Join: Equi-join & 동일한 Column명 합쳐짐.
 - Self Join: 자기 자신과 조인

Equi-Join



Theta Join

- 정의
 - 임의의 조건을 Join 조건으로 사용가능
 - Non-Equi Join이라고도 함

예

```
SELECT e.ename, e.sal, s.grade
FROM emp e, salgrade s
WHERE e.sal BETWEEN s.losal AND s.hisal
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	ENAME	SAL	GRADE
7369	SMITH	CLERK	7902	80/12/17	800	SMITH	800	1
7499	ALLEN	SALESMAN	7698	81/02/20	1600	JAMES	950	1
7521	WARD	SALESMAN	7698	81/02/22	1250	ADAMS	1100	1
7566	JONES	MANAGER	7839	81/04/02	2975	WARD	1250	2
7654	MARTIN	SALESMAN	7698	81/09/28	1250	MARTIN	1250	2
7698	BLAKE	MANAGER	7839	81/05/01	2850	MILLER	1300	2
7782	CLARK	MANAGER	7839	81/06/09	2450	TURNER	1500	3
7788	SCOTT	ANALYST	7566	87/04/19	3000	ALLEN	1600	3
7839	KING	PRESIDENT		81/11/17	5000	CLARK	2450	4
7844	TURNER	SALESMAN	7698	81/09/08	1500	BLAKE	2850	4
7876	ADAMS	CLERK	7788	87/05/23	1100	JONES	2975	4
7900	JAMES	CLERK	7698	81/12/03	950	SCOTT	3000	4
7902	FORD	ANALYST	7566	81/12/03	3000	FORD	3000	4
7934	MILLER	CLERK	7782	82/01/23	1300	KING	5000	5

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

Outer Join

- 정의
 - Join 조건을 만족하지 않는 (짜이 없는) 튜플의 경우 Null을 포함하여 결과를 생성
 - 모든 행이 결과 테이블에 나타남
- 종류
 - Left Outer Join: 왼쪽의 모든 튜플은 결과 테이블에 나타남
 - Right Outer Join: 오른쪽의 모든 튜플은 결과 테이블에 나타남
 - Full Outer Join: 양쪽 모두 결과 테이블에 나타남
- 표현 방법
 - NULL이 올 수 있는 쪽 조건에 (+)를 붙인다. (오라클)

Outer Join

EMP				FK	PK	DEPT		
EMPNO	ENAME	DEPTNO			DEPTNO	DNAME	LOC
7839	KING		10	→		10	ACCOUNTING	NEW YORK
7566	JONES		20	→		20	RESEARCH	DALLAS
7900	JAMES		30	→		30	SALES	CHICAGO
7369	SMITH		20	→		40	OPERATION	BOSTON

EMPNO	ENAME	SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO (+) = DEPT.DEPTNO					
7839	KING		10	10	ACCOUNTING	NEW YORK	
7566	JONES		20	20	RESEARCH	DALLAS	
7900	JAMES		30	30	SALES	CHICAGO	
7369	SMITH		20	20	RESEARCH	DALLAS	
7499	ALLEN		30	30	SALES	CHICAGO	
				40	OPERATION	BOSTON	

Self Join

- 자기 자신과 Join
- Alias를 사용할 수 밖에 없음

```
SELECT * FROM EMP E1, EMP E2
WHERE E1.EMPNO = E2.MGR
```

PK	EMP	FK	
EMPNO	ENAME	MGR
7839	KING		
7566	JONES	7839	
7900	JAMES	7698	
7369	SMITH	7902	
7499	ALLEN	7698	

EMPNO	ENAME	MGR	EMPNO	ENAME
7566	JONES	7839		7839	KING
7900	JAMES	7698		7698	BLAKE
7369	SMITH	7902		7902	FORD
7499	ALLEN	7698		7698	BLAKE

SQL:1999 Syntax (Oracle 9i)

- From절에서 바로 Join을 명시적으로 정의

```
SELECT table1.column, table2.column
FROM table1
[CROSS JOIN table2] |
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
ON(table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
ON (table1.column_name = table2.column_name)];
```

- 예
 - SELECT * FROM emp **NATURAL JOIN** dept;
 - SELECT * FROM emp **JOIN** dept **USING** (deptno);
 - SELECT * FROM emp **JOIN** dept **ON** emp.deptno = dept.deptno;
 - SELECT * FROM emp **RIGHT OUTER JOIN** dept **ON** (emp.deptno = dept.deptno);

GROUP & AGGREGATION

Aggregate Function (집계함수)

- 여러행으로부터 하나의 결과값을 반환
- 종류
 - AVG
 - COUNT
 - COUNT(*): number of rows in table (NULL도 count된다)
 - COUNT(expr): non-null value (NULL은 빠진다)
 - COUNT(DISTINCT expr): distinct non-null
 - MAX
 - MIN
 - SUM
 - STDDEV
 - VARIANCE

Aggregate Function

```
SELECT sal FROM emp;
```

SAL
800
1600
1250
2975
1250
2850
2450
3000
5000
1500
1100
950
3000
1300

```
SELECT AVG(sal) FROM emp;
```

AVG(SAL)
2073.21429

일반적인 오류

- `SELECT deptno, AVG(sal) FROM emp;`
- 주의
 - 집계함수의 결과는 한 row만 남게 된다.
 - deptno는 하나의 row에 표현될 수 없다.
 - 부서별과 같은 내용이 필요할 때는 **Group by**절 사용

GROUP BY

- Find the average salary of instructors in each department
 - select** *dept_name*, **avg** (*salary*) **as** avg_salary
 - from** *instructor*
 - group by** *dept_name*;

ID	name	dept_name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

GROUP BY

```
SELECT deptno, sal  
FROM emp  
ORDER BY deptno;
```

DEPTNO	SAL
10	2450
10	5000
10	1300
20	2975
20	3000
20	1100
20	800
20	3000
30	1250
30	1500
30	1600
30	950
30	2850
30	1250

```
SELECT deptno, AVG(sal)  
FROM emp  
GROUP BY deptno  
ORDER BY deptno;
```

DEPTNO	AVG(SAL)
10	2916.66667
20	2175
30	1566.66667

일반적인 오류

- 부서별 월급에서 부서명도 출력?

```
SELECT deptno, dname, AVG(sal)
FROM emp
GROUP BY deptno
ORDER BY deptno;
```

- 비록 부서번호에 따라 부서명은 하나로 결정될 수 있지만, dname은 grouping에 참여하지 않았으므로 하나의 row로 aggregate될 수 있다고 볼 수 없음
- 주의
 - SELECT의 컬럼 리스트에는 Group by에 참여한 필드나 aggregate 함수만 올 수 있다.
 - Group by가 수행된 이후에는 Group by에 참여한 필드나 aggregate 함수만 남아있는 셈 (∴ dname을 project 할 수 없음)
 - HAVING, ORDER BY 도 마찬가지

HAVING 절

- Aggregation 결과에 대해 다시 condition을 적용할 때 사용
- 일반적인 오류
 - 평균 월급이 2000 이상인 부서는?

```
SELECT deptno, AVG(sal)
FROM emp
GROUP BY deptno
HAVING AVG(sal) > 2000;
```

- 주의
 - WHERE 절은 Aggregation 이전, HAVING 절은 Aggregation 이후의 filtering
 - Having절에는 Group by에 참여한 컬럼이나 Aggregate 함수만 사용가능

SQL 문 실행 순서

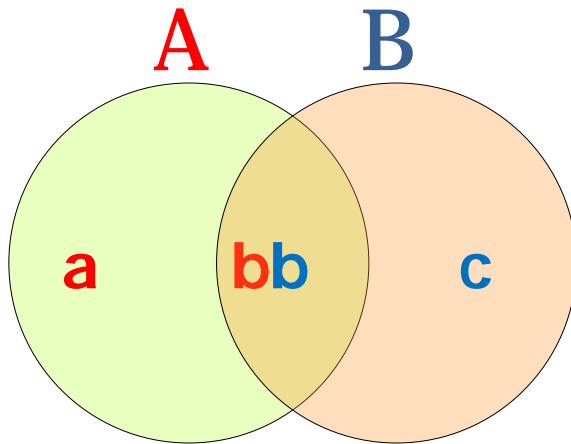


SQL 작성법

- ① 최종 출력될 정보에 따라 원하는 컬럼 SELECT 절에 추가
- ② 원하는 정보를 가진 테이블들을 FROM 절에 추가
- ③ WHERE절에 알맞은 Join 조건 추가
- ④ WHERE절에 알맞은 검색 조건 추가
- ⑤ 필요에 따라 GROUP BY, HAVING 등을 통해 Grouping
- ⑥ 정렬 조건 ORDER BY에 추가

SET Operator

- 두 질의의 결과를 가지고 집합 연산
- UNION, UNION ALL, INTERSECT, MINUS



- $A \cup B = \{a, b, c\}$
- $A \cup ALL B = \{a, b, b, c\}$
- $A \cap B = \{b\}$
- $A - B = \{a\}$

```
SELECT ename FROM emp
UNION
SELECT dname FROM dept;
```

RANK 관련 함수

```
SELECT sal, ename,  
       RANK() OVER (ORDER BY sal DESC) AS rank,  
       DENSE_RANK() OVER (ORDER BY sal DESC) AS dense_rank,  
       ROW_NUMBER() OVER (ORDER BY sal DESC) AS row_number,  
       rownum AS "rownum"  
FROM emp;
```



SAL	ENAME	RANK	DENSE_RANK	ROW_NUMBER	rownum
5000	KING	1	1	1	9
3000	FORD	2	2	2	13
3000	SCOTT	2	2	3	8
2975	JONES	4	3	4	4
2850	BLAKE	5	4	5	6
2450	CLARK	6	5	6	7
1600	ALLEN	7	6	7	2
1500	TURNER	8	7	8	10
1300	MILLER	9	8	9	14
1250	WARD	10	9	10	3
1250	MARTIN	10	9	11	5
1100	ADAMS	12	10	12	11
950	JAMES	13	11	13	12
800	SMITH	14	12	14	1

Summary

- SELECT
 - Join: 둘 이상의 테이블 결합
 - GROUP BY & Aggregation

APPENDIX

More examples & complex subqueries

Schema of the university database

Classroom (building, room number, capacity)

Department(dept name, building, budget)

Course(course id, title, dept_name, credits)

Instructor(ID, name, dept_name, salary)

Section(course id, sec id, semester, year, building, room_number, time_slot_id)

Teaches(ID, course id, sec id, semester, year)

Student(ID, name, dept_name, tot_cred)

Takes(ID, course id, sec id, semester, year, grade)

Advisor(s_ID, i_ID)

Time_slot(time slot id, day, start time, end_time)

Prereq(course id, prereq id)

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2 :
 1. $\sigma_{\theta}(r_1)$: If there are c_1 copies of tuple t_1 in r_1 , and t_1 satisfies selections σ_{θ} , then there are c_1 copies of t_1 in $\sigma_{\theta}(r_1)$.
 2. $\Pi_A(r_1)$: For each copy of tuple t_1 in r_1 , there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1 .
 3. $r_1 \times r_2$: If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2 , there are $c_1 \times c_2$ copies of the tuple $t_1 \cdot t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

- Example: Suppose multiset relations $r_1 (A, B)$ and $r_2 (C)$ are as follows:

$$r_1 = \{(1, a) (2, a)\} \quad r_2 = \{(2), (3), (3)\}$$

- Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, while $\Pi_B(r_1) \times r_2$ would be $\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$

- SQL duplicate semantics:

select A_1, A_2, \dots, A_n
from r_1, r_2, \dots, r_m
where P

is equivalent to the *multiset* version of the expression:

$$\Pi_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

Set Operations

- Find courses that ran in Fall 2010 or in Spring 2011
(select *course_id* from *section* where *sem* = 'Fall' and *year* = 2010)
union
(select *course_id* from *section* where *sem* = 'Spring' and *year* = 2011)
- Find courses that ran in Fall 2010 and in Spring 2011
(select *course_id* from *section* where *sem* = 'Fall' and *year* = 2010)
intersect
(select *course_id* from *section* where *sem* = 'Spring' and *year* = 2011)
- Find courses that ran in Fall 2010 but not in Spring 2011
(select *course_id* from *section* where *sem* = 'Fall' and *year* = 2010)
except
(select *course_id* from *section* where *sem* = 'Spring' and *year* = 2011)

[Schema of the university database](#)

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
 - Example: $5 + \text{null}$ returns null
- The predicate **is null** can be used to check for null values.
 - Example: Find all instructors whose salary is null.
select *name*
from *instructor*
where *salary* is null;
 - **is not null** is also available

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
 - Example: $5 < null$ or $null <> null$ or $null = null$
- Three-valued logic using the truth value *unknown*:
 - OR: $(unknown \text{ or } true) = true$,
 $(unknown \text{ or } false) = unknown$
 $(unknown \text{ or } unknown) = unknown$
 - AND: $(true \text{ and } unknown) = unknown$,
 $(false \text{ and } unknown) = false$,
 $(unknown \text{ and } unknown) = unknown$
 - NOT: $(\text{not } unknown) = unknown$
 - " P is **unknown**" evaluates to true if predicate P evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*
- However, aggregate functions simply ignore nulls
 - more on this shortly!

Null Values and Aggregates

- Total all salaries

```
select sum (salary)  
from instructor;
```

- Above statement ignores null amounts
 - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
 - count returns 0
 - all other aggregates return null

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

- Find courses offered in Fall 2010 and in Spring 2011 :
select distinct *course_id*
from *section*
where *semester* = 'Fall' and *year*= 2010 and
 course_id in (select *course_id*
 from *section*
 where *semester* = 'Spring' and *year*= 2011);
- Find courses offered in Fall 2010 but not in Spring 2011 :
select distinct *course_id*
from *section*
where *semester* = 'Fall' and *year*= 2010 and
 course_id not in (select *course_id*
 from *section*
 where *semester* = 'Spring' and *year*= 2011);

[Schema of the university database](#)

Example Query

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)  
from takes  
where (course_id, sec_id, semester, year) in  
      (select course_id, sec_id, semester, year  
       from teaches  
       where teaches.ID= 10101);
```

- Note: Above query can be written in a different manner.
- The formulation above is simply to illustrate SQL features.

[Schema of the university database](#)

Set Comparison

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > some clause

```
select name
from instructor
where salary > some (select salary
                      from instructor
                      where dept name = 'Biology');
```

Definition of Some Clause

- $F \text{ <comp> some } r \Leftrightarrow \exists t \in r \text{ such that } (F \text{ <comp> } t)$

Where <comp> can be: <, ≤, >, =, ≠

$$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true} \quad (\text{read: } 5 < \text{some tuple in the relation})$$

$$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 = \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$$

$$(5 \neq \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true (since } 0 \neq 5)$$

(= some) ≡ in

However, (≠ some) ≠ not in

Example Query

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                     from instructor
                     where dept_name = 'Biology');
```

Definition of all Clause

- $F \text{ <comp> all } r \Leftrightarrow \forall t \in r (F \text{ <comp> } t)$

$$(5 < \text{all } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \text{all } \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \text{all } \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 \neq \text{all } \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

$(\neq \text{all}) \equiv \text{not in}$

However, $(= \text{all}) \not\equiv \text{in}$

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Correlation Variables

- Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

```
select course_id
from section as S
where semester = 'Fall' and year = 2009 and
      exists (select *
              from section as T
              where semester = 'Spring' and year = 2010
                  and S.course_id = T.course_id);
```

- **Correlated subquery**
- **Correlation name** or **correlation variable**

Not Exists

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                  except
                  (select T.course_id
                   from takes as T
                   where S.ID = T.ID));
```

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.

- Find all courses that were offered **at most** once in 2009

```
select T.course_id
from course as T
where unique (select R.course_id
                  from section as R
                  where T.course_id= R.course_id and R.year = 2009);
```

- If a course is not offered in 2009, the subquery would return an empty result, and the unique predicate becomes true.

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than \$42,000.

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause!
- Another way to write above query

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name) as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

Derived Relations (Cont.)

- And yet another way to write it: **lateral** clause

```
select name, salary, avg_salary  
from instructor I1, lateral (select avg(salary) as avg_salary  
                                from instructor I2  
                                where I2.dept_name= I1.dept_name);
```

- Use lateral clause to access attributes of preceding tables or subqueries in the from clause
 - Cannot use correlation variables from other relations in the from clause
 - Only a few SQL implementations such as IBM DB2 support the lateral clause.

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as  
    (select max(budget)  
     from department)  
select dept_name, budget  
from department, max_budget  
where department.budget = max_budget.value;
```

[Schema of the university database](#)

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
     from instructor
     group by dept_name),
dept_total_avg(value) as
    (select avg(value)
     from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

Scalar Subquery

- **Scalar subquery:** returns only one tuple containing a single attribute
 - E.g., `count(*)` without group by
- **Scalar subqueries can occur in select, where, and having clause.**

```
select dept_name,  
       (select count(*)  
        from instructor  
        where department.dept_name = instructor.dept_name)  
       as num_instructors  
from department;
```


Modification of the Database – Deletion

- Delete all instructors

delete from *instructor*

- Delete all instructors from the Finance department

delete from *instructor*

where *dept_name* = 'Finance';

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

delete from *instructor*

where *dept_name* in (**select** *dept_name*
from *department*
where *building* = 'Watson');

Example Query

- Delete all instructors whose salary is less than the average salary of instructors

delete from *instructor*

where *salary* < (select avg (*salary*) from *instructor*);

- Problem: as we delete tuples from *instructor*, the average salary changes
- Solution used in SQL:
 1. First, compute **avg** salary and find all tuples to delete
 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to *course*

insert into *course*

values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

insert into *course* (*course_id*, *title*, *dept_name*, *credits*)

values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

insert into *student*

values ('3003', 'Green', 'Finance', *null*);

Modification of the Database – Insertion

- Add all instructors to the *student* relation with *tot_creds* set to 0
insert into *student*
 select *ID, name, dept_name, 0*
 from *instructor*
- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like
 insert into table1 select * from table1
would cause problems)

[Schema of the university database](#)

Modification of the Database – Updates

- Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others receive a 5% raise
 - Write two **update** statements:

```
update instructor
  set salary = salary * 1.03
  where salary > 100000;
update instructor
  set salary = salary * 1.05
  where salary <= 100000;
```
 - The order is important
 - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor  
  set salary = case  
    when salary <= 100000 then salary * 1.05  
    else salary * 1.03  
  end
```

Updates with Scalar Subqueries

- Recompute and update *tot_creds* value for all students

```
update student S
  set tot_cred = ( select sum(credits)
                    from takes natural join course
                    where S.ID = takes.ID and
                        takes.grade <> 'F' and
                        takes.grade is not null);
```

 - Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
  when sum(credits) is not null then sum(credits)
  else 0
end
```

 - Sets *tot_creds* to 0 for a student who has not successfully completed any course.

[Schema of the university database](#)

END