

## 4.2 SQL 개요

### □ SQL 개요

- ✓ SQL은 현재 DBMS 시장에서 관계 DBMS가 압도적인 우위를 차지하는데 중요한 요인의 하나
- ✓ SQL은 IBM 연구소에서 1974년에 **System R**이라는 관계 DBMS 시제품을 연구할 때 관계 대수와 관계 해석을 기반으로, 집단 함수, 그룹화, 갱신 연산 등을 추가하여 개발된 언어
- ✓ 1986년에 ANSI(미국 표준 기구)에서 SQL 표준을 채택함으로써 SQL이 널리 사용되는데 기여
- ✓ 다양한 상용 관계 DBMS마다 지원하는 SQL 기능에 다소 차이가 있음
- ✓ 본 책에서는 SQL2를 따름
- ✓ 관계 데이터 모델은 집합을 기반을 두고 있어 테이블 내에 동일한 튜플을 허용하지 않지만 SQL은 이를 허용함

## 4.2 SQL 개요(계속)

〈표 4.2〉 SQL의 발전 역사

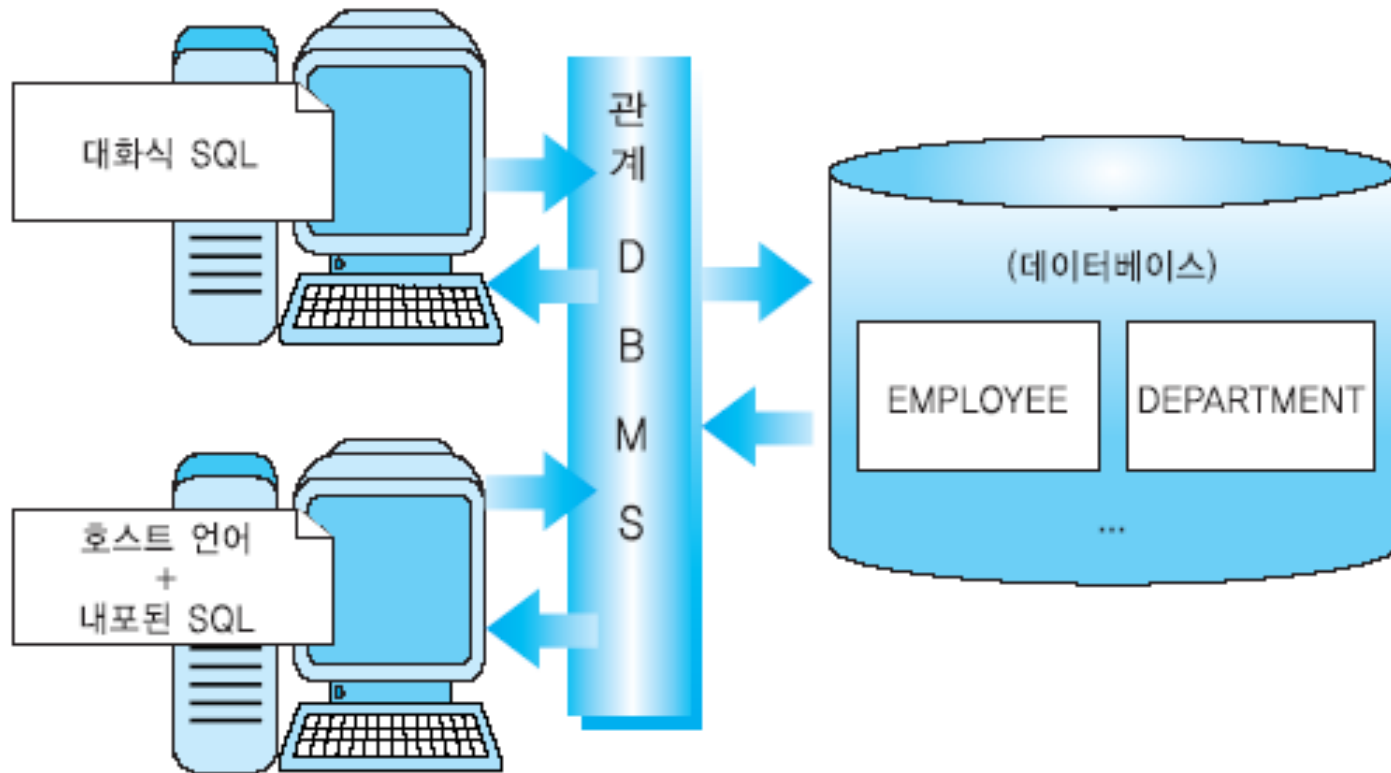
버전	특징
SEQUEL	Structured English Query Language의 약어. Sysetm R 프로젝트에서 처음으로 제안됨
SQL	Structured Query Language의 약어. 1983년에 IBM의 DB2, 1991년에 IBM SQL/DS에 사용됨
SQL-86	1986년에 미국 ANSI에서 표준으로 채택됨. 1987년에 ISO에서 표준으로 채택됨
SQL-89	무결성 제약조건 기능이 강화됨
SQL2(SQL-92)	새로운 데이터 정의어와 데이터 조작어 기능이 추가됨. 약 500페이지 분량
SQL3(SQL-99)	객체 지향과 순환, 멀티미디어 기능 등이 추가됨. 약 2000페이지 분량

## 4.2 SQL 개요(계속)

### □ SQL 개요(계속)

- ✓ SQL은 비절차적 언어(선언적 언어)이므로 사용자는 자신이 원하는 바(what)만 명시하며, 원하는 것을 처리하는 방법(how)은 명시할 수 없음
- ✓ 관계 DBMS는 사용자가 입력한 SQL문을 번역하여 사용자가 요구한 데이터를 찾는데 필요한 모든 과정을 담당
- ✓ 자연어에 가까운 구문을 사용하여 질의를 표현할 수 있음
- ✓ 두 가지 인터페이스
  - 대화식 SQL(interactive SQL)
  - 내포된 SQL(embedded SQL)

## 4.2 SQL 개요(계속)



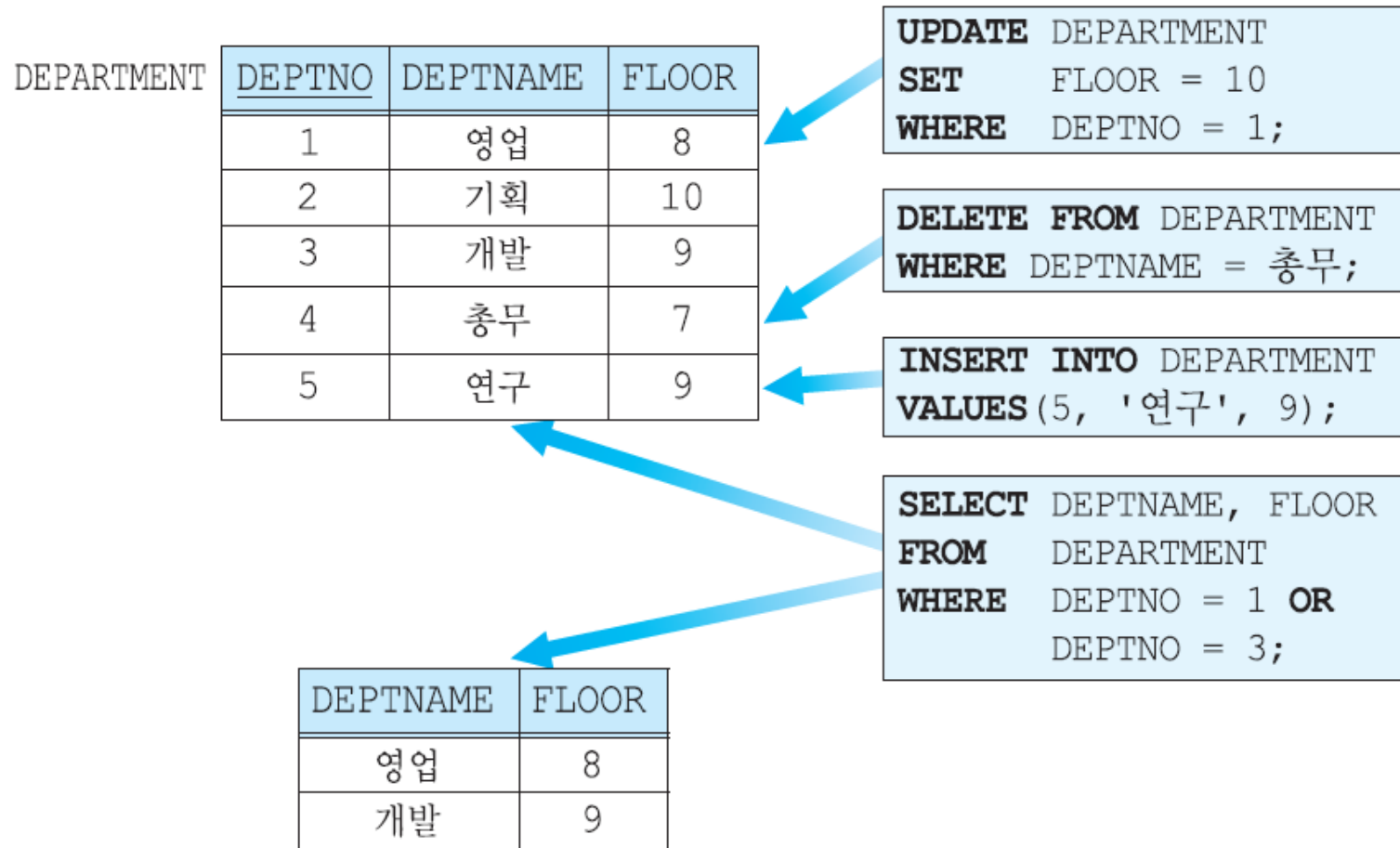
[그림 4.3] 관계 데이터베이스에 대한 두 가지 인터페이스

## 4.2 SQL 개요(계속)

### □ 오라클 SQL의 구성요소

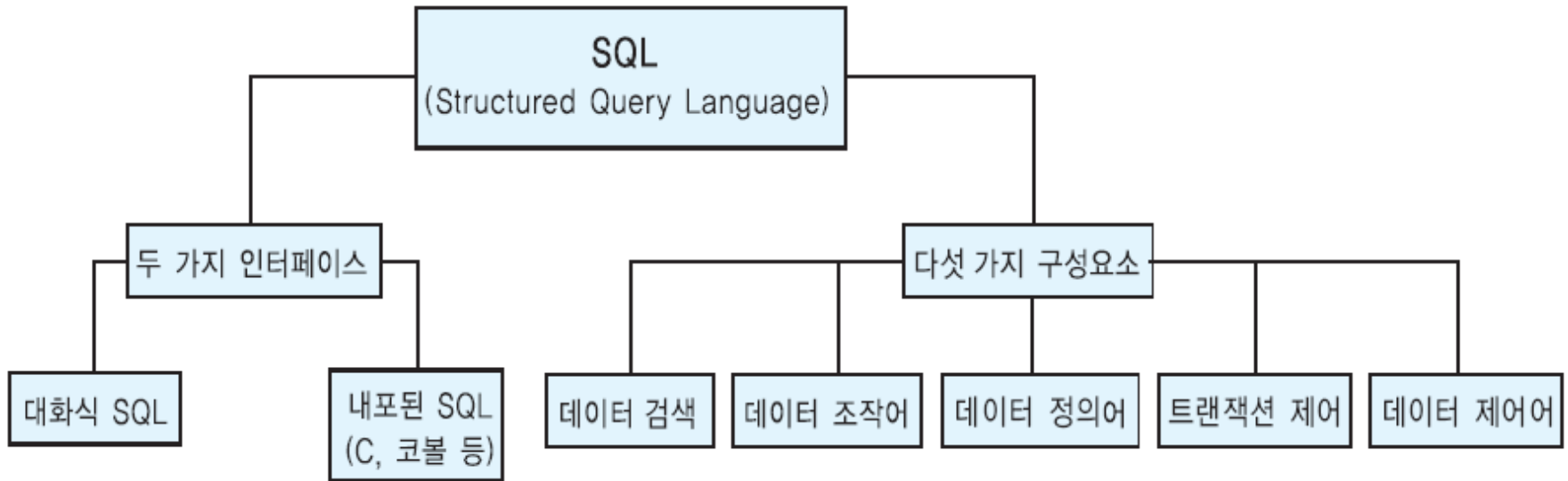
- ✓ 데이터 검색
- ✓ 데이터 조작어
- ✓ 데이터 정의어
- ✓ 트랜잭션 제어
- ✓ 데이터 제어어

## 4.2 SQL 개요(계속)



[그림 4.4] 데이터 검색과 데이터 조작어의 기능

## 4.2 SQL 개요(계속)



[그림 4.5] SQL의 인터페이스와 구성요소

## 4.3 데이터 정의어와 무결성 제약조건

〈표 4.4〉 데이터 정의어의 종류

CREATE	DOMAIN	도메인을 생성
	TABLE	테이블을 생성
	VIEW	뷰를 생성
	INDEX	인덱스를 생성. SQL2의 표준이 아님
ALTER	TABLE	테이블의 구조를 변경
DROP	DOMAIN	도메인을 제거
	TABLE	테이블을 제거
	VIEW	뷰를 제거
	INDEX	인덱스를 제거. SQL2의 표준이 아님



## 4.3 데이터 정의어와 무결성 제약조건(계속)

### □ 데이터 정의어

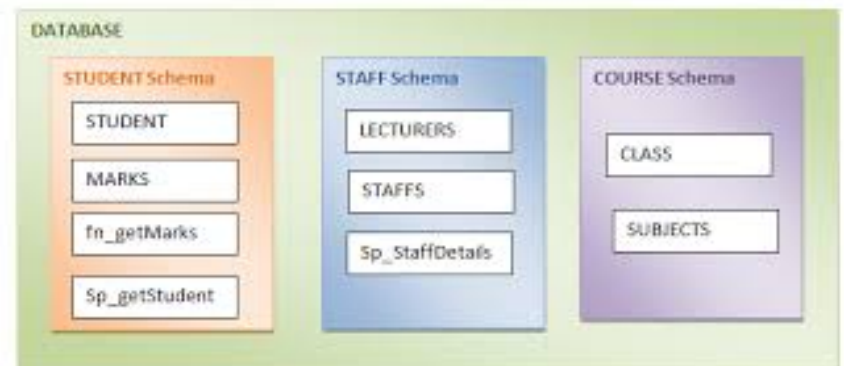
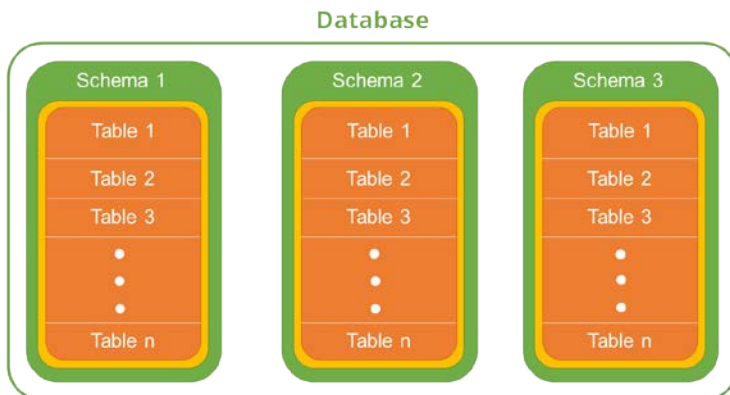
#### ✓ 스키마의 생성과 제거

- SQL2에서는 동일한 데이터베이스 응용에 속하는 릴레이션, 도메인, 제약조건, 뷰, 권한 등을 그룹화하기 위해서 스키마 개념을 지원

```
CREATE SCHEMA MY_DB AUTHORIZATION kim;
```

```
DROP SCHEMA MY_DB RESTRICT;
```

```
DROP SCHEMA MY_DB CASCADE;
```



## 4.3 데이터 정의어와 무결성 제약조건(계속)

### □ 릴레이션 정의

```
CREATE TABLE EMPLOYEE
    (EMPNO      NUMBER    NOT NULL,
     EMPNAME    CHAR(10) ,
     TITLE      CHAR(10) ,
     MANAGER     NUMBER,
     SALARY     NUMBER,
     DNO        NUMBER,
     PRIMARY KEY (EMPNO) ,
     FOREIGN KEY (MANAGER) REFERENCES EMPLOYEE (EMPNO) ,
     FOREIGN KEY (DNO) REFERENCES DEPARTMENT (DEPTNO) ) ;
```

[그림 4.6] DEPARTMENT 릴레이션과 EMPLOYEE 릴레이션의 생성

## 4.3 데이터 정의어와 무결성 제약조건(계속)

〈표 4.5〉 릴레이션의 정의에 사용되는 오라클의 데이터 타입

데이터 타입	의미
INTEGER 또는 INT	정수형
NUMBER(n, s)	소수점을 포함한 n개의 숫자에서 소수 아래 숫자가 s개인 십진수
CHAR (n) 또는 CHARACTER (n)	n바이트 문자열. n을 생략하면 1
VARCHAR (n) , VARCHAR2 (n) 또는 CHARACTER VARYING (N)	최대 n바이트까지의 가변 길이 문자열
BIT (n) 또는 BIT VARYING (n)	n개의 비트열 또는 최대 n개까지의 가변 비트열
DATE	날짜형. 날짜와 시간을 저장
BINARY_FLOAT	오라클 10g부터 도입되었는데, 32비트에 실수를 저장
BINARY_DOUBLE	오라클 10g부터 도입되었는데, 64비트에 실수를 저장
BLOB	Binary Large Object. 멀티미디어 데이터 등을 저장

- Oracle의 VARCHAR가 표준과 다름  
- Empty string을 NULL로 간주
- VARCHAR 시맨틱이 변경 예정
- VARCHAR2 사용을 권장

- 가변길이 문자열은 내부 처리로 성능 저하될 수 있음
- 키나 인덱스 컬럼 등은 고정길이로 공간이 크게 낭비되지 않으면 고정길이 문자열 사용

## 4.3 데이터 정의어와 무결성 제약조건(계속)

### ❑ 릴레이션 제거

```
DROP TABLE DEPARTMENT ;
```

### ❑ ALTER TABLE

```
ALTER TABLE EMPLOYEE ADD PHONE CHAR(13) ;
```

### ❑ 인덱스 생성

```
CREATE INDEX EMPDNO_IDX ON EMPLOYEE(DNO) ;
```

## 4.3 데이터 정의어와 무결성 제약조건(계속)

### □ 제약조건

```
CREATE TABLE EMPLOYEE
(EMPNO    NUMBER    NOT NULL,                (1)
 EMPNAME  CHAR(10)  UNIQUE,                  (2)
 TITLE    CHAR(10)  DEFAULT '사원',          (3)
 MANAGER   NUMBER,
 SALARY    NUMBER    CHECK (SALARY < 6000000), (4)
 DNO       NUMBER    CHECK (DNO IN (1,2,3,4,5,6)) DEFAULT 1, (5)
 PRIMARY KEY (EMPNO),                        (6)
 FOREIGN KEY (MANAGER) REFERENCES EMPLOYEE (EMPNO), (7)
 FOREIGN KEY (DNO) REFERENCES DEPARTMENT (DEPTNO) (8)
 ON DELETE CASCADE);                        (9)
```

[그림 4.7] 릴레이션 정의에서 다양한 제약조건을 명시

Unnamed  
constraint?

## 4.3 데이터 정의어와 무결성 제약조건(계속)

```
CREATE TABLE EMPLOYEE (  
    ID NUMBER ,  
    NAME CHAR(10) ,  
    SALARY NUMBER ,  
    MANAGER_SALARY NUMBER ,  
    CHECK (MANAGER_SALARY > SALARY) ) ;
```

## 4.3 데이터 정의어와 무결성 제약조건(계속)

### ❑ 참조 무결성 제약조건 유지

**ON DELETE NO ACTION**

**ON DELETE CASCADE**

**ON DELETE SET NULL**

**ON DELETE SET DEFAULT**

**ON UPDATE NO ACTION**

## 4.3 데이터 정의어와 무결성 제약조건(계속)

### 예 : ON DELETE CASCADE

4.5절에서 설명할 DELETE문을 사용하여 다음과 같이 DEPARTMENT 릴레이션에서 3번 부서의 튜플을 삭제하면, EMPLOYEE 릴레이션에서 3번 부서에 근무하는 모든 직원들의 튜플도 자동적으로 삭제된다.

```
DELETE DEPARTMENT  
WHERE DEPTNO = 3;
```



## 4.3 데이터 정의어와 무결성 제약조건(계속)

DEPARTMENT

DEPTNO	DEPTNAME	FLOOR
1	영업	8
2	기획	10
3	개발	9
4	총무	7

① 삭제

연쇄

EMPLOYEE

EMPNO	EMPNAME	...	DNO
2106	김창섭	...	2
3426	박영권	...	1
3011	이수민	...	3
1003	조민희	...	2
3427	최종철	...	3
1365	김상원	...	1
4377	이성래	...	2

기본 키의 삭제가  
외래 키에도 파급됨

② 삭제

## 4.3 데이터 정의어와 무결성 제약조건(계속)

### □ 무결성 제약조건의 추가 및 삭제

```
ALTER TABLE STUDENT ADD CONSTRAINT STUDENT_PK  
    PRIMARY KEY (STNO);
```

```
ALTER TABLE STUDENT DROP CONSTRAINT STUDENT_PK;
```

## 4.4 SELECT문

### □ SELECT문

- ✓ 관계 데이터베이스에서 정보를 검색하는 SQL문
- ✓ 관계 대수의 실렉션과 의미가 완전히 다름
- ✓ 관계 대수의 실렉션, 프로젝션, 조인, 카티션 곱 등을 결합한 것
- ✓ 관계 데이터베이스에서 가장 자주 사용됨
- ✓ 여러 가지 질의들의 결과를 보이기 위해서 그림 4.8의 관계 데이터베이스 상태를 사용함

## 4.4 SELECT문(계속)

EMPLOYEE

<u>EMPNO</u>	EMPNAME	TITLE	MANAGER	SALARY	DNO
2106	김창섭	대리	1003	2500000	2
3426	박영권	과장	4377	3000000	1
3011	이수민	부장	4377	4000000	3
1003	조민희	과장	4377	3000000	2
3427	최종철	사원	3011	1500000	3
1365	김상원	사원	3426	1500000	1
4377	이성래	사장	^	5000000	2

DEPARTMENT

<u>DEPTNO</u>	DEPTNAME	FLOOR
1	영업	8
2	기획	10
3	개발	9
4	총무	7

[그림 4.8] 관계 데이터베이스 상태

## 4.4 SELECT문(계속)

### □ 기본적인 SQL 질의

- ✓ SELECT절과 FROM절만 필수적인 절이고, 나머지는 선택 사항

<b>SELECT</b>	<b>[ DISTINCT ]</b> 애트리뷰트(들)	(1)	} 필수
<b>FROM</b>	릴레이션(들)	(2)	
<b>[ WHERE</b>	조건	(3)	} 선택
	<b>[ 중첩 질의 ]</b>	(4)	
<b>[ GROUP BY</b>	애트리뷰트(들)	(5)	
<b>[ HAVING</b>	조건	(6)	
<b>[ ORDER BY</b>	애트리뷰트(들) <b>[ ASC   DESC ]</b> ;	(7)	

[그림 4.9] SELECT문의 형식

## 4.4 SELECT문(계속)

### □ 별칭(alias)

- ✓ 서로 다른 릴레이션에 동일한 이름을 가진 애트리뷰트가 속해 있을 때 애트리뷰트의 이름을 구분하는 방법

EMPLOYEE.DNO

**FROM** EMPLOYEE **AS** E, DEPARTMENT **AS** D

## 4.4 SELECT문(계속)

□ 릴레이션의 모든 애트리뷰트나 일부 애트리뷰트들을 검색

예 : \* 를 사용하여 모든 애트리뷰트들을 검색

질의: 전체 부서의 모든 애트리뷰트들을 검색하라.

```
SELECT      *  
FROM        DEPARTMENT;
```

DEPTNO	DEPTNAME	FLOOR
1	영업	8
2	기획	10
3	개발	9
4	총무	7

## 4.4 SELECT문(계속)

예 : 원하는 애트리뷰트들의 이름을 열거

질의: 모든 부서의 부서번호와 부서이름을 검색하라.

```
SELECT      DEPTNO, DEPTNAME  
FROM        DEPARTMENT;
```

DEPTNO	DEPTNAME
1	영업
2	기획
3	개발
4	총무



## 4.4 SELECT문(계속)

### □ 상이한 값들을 검색

예 : DISTINCT절을 사용하지 않을 때

질의: 모든 사원들의 직급을 검색하라.

```
SELECT    TITLE
FROM      EMPLOYEE;
```

TITLE
대리
과장
부장
과장
사원
사원
사장

## 4.4 SELECT문(계속)

예 : DISTINCT절을 사용할 때

질의: 모든 사원들의 상이한 직급을 검색하라.

```
SELECT      DISTINCT TITLE  
FROM        EMPLOYEE;
```

TITLE
대리
과장
부장
사원
사장

## 4.4 SELECT문(계속)

### □ 특정한 튜플들의 검색

예 : WHERE절을 사용하여 검색 조건을 명시

질의: 2번 부서에 근무하는 직원들에 관한 모든 정보를 검색하라.

```
SELECT      *  
FROM        EMPLOYEE  
WHERE       DNO = 2;
```

EMPNO	EMPNAME	TITLE	MANAGER	SALARY	DNO
1003	조민희	과장	4377	3000000	2
2016	김창섭	대리	1003	2500000	2
4377	이성래	사장	^	5000000	2

## 4.4 SELECT문(계속)

### □ 문자열 비교

#### 예 : %를 사용하여 문자열 비교

질의: 이씨 성을 가진 사원들의 이름, 직급, 소속 부서번호를 검색하라.

```
SELECT      EMPNAME, TITLE, DNO
FROM        EMPLOYEE
WHERE        EMPNAME LIKE '이%';
```

EMPNAME	TITLE	DNO
이수민	부장	3
이성래	사장	2

## 4.4 SELECT문(계속)

### □ 다수의 검색 조건

✓ 아래와 같은 질의는 잘못되었음

```
SELECT      FLOOR  
FROM        DEPARTMENT  
WHERE       DEPTNAME= '영업' AND DEPTNAME= '개발' ;
```

〈표 4.6〉 연산자들의 우선 순위

연산자	우선순위
비교 연산자	1
NOT	2
AND	3
OR	4

## 4.4 SELECT문(계속)

### 예 : 부울 연산자를 사용한 프레디키트

질의: 직급이 과장이면서 1번 부서에서 근무하는 직원들의 이름과 급여를 검색하라.

```
SELECT      EMPNAME, SALARY
FROM        EMPLOYEE
WHERE        TITLE = '과장' AND DNO = 1;
```

EMPNAME	SALARY
박영권	3000000

## 4.4 SELECT문(계속)

### □ 부정 검색 조건

#### 예 : 부정 연산자

질의: 직급이 과장이면서 1번 부서에 속하지 않은 직원들의 이름과 급여를 검색하라.

```
SELECT      EMPNAME, SALARY
FROM        EMPLOYEE
WHERE        TITLE = '과장' AND DNO <> 1;
```

EMPNAME	SALARY
조민희	3000000

## 4.4 SELECT문(계속)

### □ 범위를 사용한 검색

#### 예 : 범위 연산자

질의: 급여가 3000000원 이상이고, 4500000원 이하인 직원들의 이름, 직급, 급여를 검색하라.

```
SELECT      EMPNAME, TITLE, SALARY
FROM        EMPLOYEE
WHERE        SALARY BETWEEN 3000000 AND 4500000;
```

BETWEEN은 양쪽의 경계값을 포함하므로 이 질의는 아래의 질의와 동등하다.

```
SELECT      EMPNAME, TITLE, SALARY
FROM        EMPLOYEE
WHERE        SALARY >= 3000000 AND SALARY <= 4500000;
```

EMPNAME	TITLE	SALARY
박영권	과장	3000000
이수민	부장	4000000
조민희	과장	3000000



## 4.4 SELECT문(계속)

### □ 리스트를 사용한 검색

예 : IN

질의: 1번 부서나 3번 부서에 소속된 직원들에 관한 모든 정보를 검색하라.

```
SELECT      *  
FROM        EMPLOYEE  
WHERE       DNO IN (1, 3);
```

EMPNO	EMPNAME	TITLE	MANAGER	SALARY	DNO
1365	김상원	사원	3426	1500000	1
3011	이수민	부장	4377	4000000	3
3426	박영권	과장	4377	3000000	1
3427	최종철	사원	3011	1500000	3

## 4.4 SELECT문(계속)

### □ SELECT절에서 산술 연산자(+, -, \*, /) 사용

#### 예 : 산술 연산자

질의: 직급이 과장인 직원들에 대하여 이름과, 현재의 급여, 급여가 10% 인상됐을 때의 값을 검색하라.

```
SELECT      EMPNAME, SALARY, SALARY * 1.1 AS NEWSALARY
FROM        EMPLOYEE
WHERE       TITLE = '과장' ;
```

EMPNAME	SALARY	NEWSALARY
박영권	3000000	3300000
조민희	3000000	3300000

## 4.4 SELECT문(계속)

### □ 널값

- ✓ 널값을 포함한 다른 값과 널값을 +, - 등을 사용하여 연산하면 결과는 널
- ✓ COUNT(\*)를 제외한 집단 함수들은 널값을 무시함
- ✓ 어떤 애트리뷰트에 들어 있는 값이 널인가 비교하기 위해서  
'DNO=NULL' 처럼 나타내면 안됨

```
SELECT    EMPNO, EMPNAME
FROM      EMPLOYEE
WHERE     DNO = NULL;
```

## 4.4 SELECT문(계속)

### □ 널값(계속)

- ✓ 다음과 같은 비교 결과는 모두 거짓

NULL > 300

NULL = 300

NULL <> 300

NULL = NULL

NULL <> NULL

- ✓ 올바른 표현

```
SELECT  EMPNO, EMPNAME
FROM    EMPLOYEE
WHERE   DNO IS NULL;
```

\* 반대는 IS NOT NULL

## 4.4 SELECT문(계속)

### □ 세 가지 값의 논리 (Three valued logic)

✓ true/false/unknown

〈표 4.7〉 unknown에 대한 OR 연산

	true	false	unknown
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

〈표 4.8〉 unknown에 대한 AND 연산

	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

## 4.4 SELECT문(계속)

〈표 4.9〉 unknown에 대한 NOT 연산

true	false
false	true
unknown	unknown

true = 1, false = 0, unknown = 0.5

$C1 \text{ AND } C2 = \min(C1, C2)$

$C1 \text{ OR } C2 = \max(C1, C2)$

$\text{NOT}(C1) = 1 - C1$

## 4.4 SELECT문(계속)

### □ ORDER BY절

- ✓ 사용자가 SELECT문에서 질의 결과의 순서를 명시하지 않으면 릴레이션에 튜플들이 삽입된 순서대로 사용자에게 제시됨
- ✓ ORDER BY절에서 하나 이상의 애트리뷰트를 사용하여 검색 결과를 정렬할 수 있음
- ✓ SELECT문에서 가장 마지막에 사용되는 절
- ✓ 디폴트 정렬 순서는 오름차순(ASC)
- ✓ DESC를 지정하여 정렬 순서를 내림차순으로 지정할 수 있음
- ✓ 넓은 오름차순에서는 가장 마지막에 나타나고, 내림차순에서는 가장 앞에 나타남
- ✓ SELECT절에 명시한 애트리뷰트들을 사용해서 정렬해야 함

## 4.4 SELECT문(계속)

### 예 : ORDER BY

질의: 2번 부서에 근무하는 직원들의 급여, 직급, 이름을 검색하여 급여의 오름차순으로 정렬하라.

```
SELECT    SALARY, TITLE, EMPNAME
FROM      EMPLOYEE
WHERE     DNO = 2
ORDER BY  SALARY;
```

SALARY	TITLE	EMPNAME
2500000	대리	김창섭
3000000	과장	조민희
5000000	사장	이성래



## 4.4 SELECT문(계속)

### □ 집단 함수

- ✓ 데이터베이스에서 검색된 여러 튜플들의 집단에 적용되는 함수
- ✓ 한 릴레이션의 한 개의 애트리뷰트에 적용되어 단일 값을 반환함
- ✓ SELECT절과 HAVING절에만 나타날 수 있음
- ✓ COUNT(\*)를 제외하고는 널값을 제거한 후 남아 있는 값들에 대해서 집단 함수의 값을 구함
- ✓ COUNT(\*)는 결과 릴레이션의 모든 행들의 총 개수를 구하는 반면에 COUNT(애트리뷰트)는 해당 애트리뷰트에서 널값이 아닌 값들의 개수를 구함
- ✓ 키워드 DISTINCT가 집단 함수 앞에 사용되면 집단 함수가 적용되기 전에 먼저 중복을 제거함

## 4.4 SELECT문(계속)

〈표 4.10〉 집단 함수의 기능

집단 함수	기능
COUNT	튜플이나 값들의 개수
SUM	값들의 합
AVG	값들의 평균값
MAX	값들의 최대값
MIN	값들의 최소값

## 4.4 SELECT문(계속)

### 예 : 집단 함수

질의: 모든 사원들의 평균 급여와 최대 급여를 검색하라.

```
SELECT      AVG (SALARY) AS AVGSAL, MAX (SALARY) AS MAXSAL  
FROM        EMPLOYEE;
```

AVGSAL	MAXSAL
2928571	5000000

## 4.4 SELECT문(계속)

### □ 그룹화

- ✓ GROUP BY절에 사용된 애트리뷰트에 동일한 값을 갖는 튜플들이 각각 하나의 그룹으로 묶임
- ✓ 이 애트리뷰트를 **그룹화 애트리뷰트**(grouping attribute)라고 함
- ✓ 각 그룹에 대하여 결과 릴레이션에 하나의 튜플이 생성됨
- ✓ **SELECT절에는 집단 함수, 그룹화 애트리뷰트들만 나타날 수 있음**
- ✓ 다음 질의는 그룹화를 하지 않은 채 EMPLOYEE 릴레이션의 모든 튜플에 대해서 사원번호와 모든 사원들의 평균 급여를 검색하므로 잘못됨

```
SELECT  EMPNO, AVG(SALARY)
FROM    EMPLOYEE;
```

## 4.4 SELECT문(계속)

### 예 : 그룹화

질의: 모든 사원들에 대해서 사원들이 속한 부서번호별로 그룹화하고, 각 부서마다 부서번호, 평균 급여, 최대 급여를 검색하라.

```
SELECT    DNO, AVG(SALARY)AS AVGSAL, MAX(SALARY)AS MAXSAL
FROM      EMPLOYEE
GROUP BY  DNO;
```

## 4.4 SELECT문(계속)

EMPLOYEE

EMPNO	EMPNAME	TITLE	MANAGER	SALARY	DNO
3426	박영권	과장	4377	3000000	1
1365	김상원	사원	3426	1500000	1
2106	김창섭	대리	1003	2500000	2
1003	조민희	과장	4377	3000000	2
4377	이성래	사장	^	5000000	2
3011	이수민	부장	4377	4000000	3
3427	최종철	사원	3011	1500000	3



→

DNO	AVGSAL	MAXSAL
1	2250000	3000000
2	3500000	5000000
3	2750000	4000000

## 4.4 SELECT문(계속)

### □ HAVING절

- ✓ 어떤 조건을 만족하는 그룹들에 대해서만 집단 함수를 적용할 수 있음
- ✓ 각 그룹마다 하나의 값을 갖는 애트리뷰트를 사용하여 각 그룹이 만족해야 하는 조건을 명시함
- ✓ 그룹화 애트리뷰트에 같은 값을 갖는 튜플들의 그룹에 대한 조건을 나타내고, 이 조건을 만족하는 그룹들만 질의 결과에 나타남
- ✓ HAVING절에 나타나는 애트리뷰트는 반드시 GROUP BY절에 나타나거나 집단 함수에 포함되어야 함

## 4.4 SELECT문(계속)

### 예 : 그룹화

질의: 모든 직원들에 대해서 직원들이 속한 부서번호별로 그룹화하고, 평균 급여가 2500000원 이상인 부서에 대해서 부서번호, 평균 급여, 최대 급여를 검색하라.

```
SELECT      DNO, AVG (SALARY) AS AVGSAL, MAX (SALARY) AS MAXSAL
FROM        EMPLOYEE
GROUP BY    DNO
HAVING      AVG (SALARY) >= 2500000;
```



## 4.4 SELECT문(계속)

EMPLOYEE

EMPNO	EMPNAME	TITLE	MANAGER	SALARY	DNO
3426	박영권	과장	4377	3000000	1
1365	김상원	사원	3426	1500000	1
2106	김창섭	대리	1003	2500000	2
1003	조민희	과장	4377	3000000	2
4377	이성래	사장	^	5000000	2
3011	이수민	부장	4377	4000000	3
3427	최종철	사원	3011	1500000	3

그룹

GROUP BY



DNO	AVGSAL	MAXSAL
1	2250000	3000000
2	3500000	5000000
3	2750000	4000000

HAVING



DNO	AVGSAL	MAXSAL
2	3500000	5000000
3	2750000	4000000

## 4.4 SELECT문(계속)

### □ 집합 연산

- ✓ 집합 연산을 적용하려면 두 릴레이션이 합집합 호환성을 가져야 함
- ✓ UNION(합집합), EXCEPT(차집합), INTERSECT(교집합), UNION ALL(합집합), EXCEPT ALL(차집합), INTERSECT ALL(교집합)

## 4.4 SELECT문(계속)

### 예 : 합집합

질의: 김창섭이 속한 부서이거나 개발 부서의 부서번호를 검색하라.

```
(SELECT      DNO
FROM      EMPLOYEE
WHERE      EMPNAME = '김창섭' )
UNION
(SELECT      DEPTNO
FROM      DEPARTMENT
WHERE      DEPTNAME = '개발' );
```

DNO
2
3

## 4.4 SELECT문(계속)

### □ 조인

- ✓ 두 개 이상의 릴레이션으로부터 연관된 튜플들을 결합
- ✓ 일반적인 형식은 아래의 SELECT문과 같이 FROM절에 두 개 이상의 릴레이션들이 열거되고, 두 릴레이션에 속하는 애트리뷰트들을 비교하는 조인 조건이 WHERE절에 포함됨
- ✓ 조인 조건은 두 릴레이션 사이에 속하는 애트리뷰트 값들을 비교 연산자로 연결한 것
- ✓ 가장 흔히 사용되는 비교 연산자는 =

```
SELECT      ...  
FROM        R, S  
WHERE       R.A <비교 연산자> S.B ;
```

↑  
조인 조건

## 4.4 SELECT문(계속)

### □ 조인(계속)

- ✓ 조인 조건을 생략했을 때와 조인 조건을 틀리게 표현했을 때는 카티션 곱이 생성됨
- ✓ 조인 질의가 수행되는 과정을 개념적으로 살펴보면 먼저 조인 조건을 만족하는 튜플들을 찾고, 이 튜플들로부터 SELECT절에 명시된 애트리뷰트들만 프로젝트하고, 필요하다면 중복을 배제하는 순서로 진행됨
- ✓ 조인 조건이 명확해지도록 애트리뷰트 이름 앞에 릴레이션 이름이나 튜플 변수를 사용하는 것이 바람직
- ✓ 두 릴레이션의 조인 애트리뷰트 이름이 동일하다면 반드시 애트리뷰트 이름 앞에 릴레이션 이름이나 튜플 변수를 사용해야 함

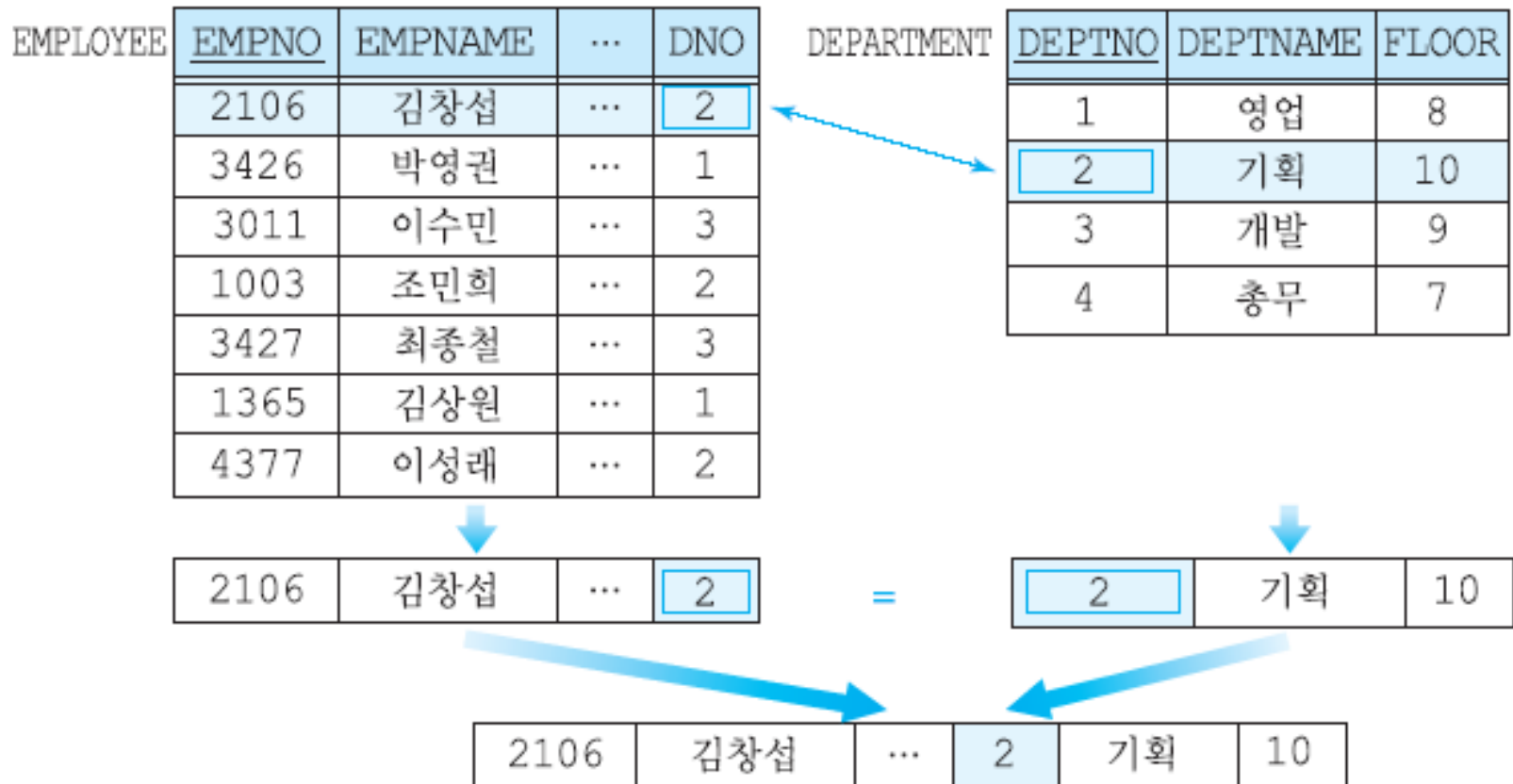
## 4.4 SELECT문(계속)

### 예 : 조인 질의

질의: 모든 사원의 이름과 이 사원이 속한 부서 이름을 검색하라.

```
SELECT      EMPNAME, DEPTNAME  
FROM        EMPLOYEE AS E, DEPARTMENT AS D  
WHERE       E.DNO = D.DEPTNO;
```

## 4.4 SELECT문(계속)



## 4.4 SELECT문(계속)

최종 결과 릴레이션은 아래의 릴레이션에서 **EMPNAME**과 **DEPTNAME**을 프로젝션한 것이다.

EMPNO	EMPNAME	TITLE	MANAGER	SALARY	DNO	DEPTNAME	FLOOR
1003	조민희	과장	4377	3000000	2	기획	10
1365	김상원	사원	3426	1500000	1	영업	8
2106	김창섭	대리	1003	2500000	2	기획	10
3011	이수민	부장	4377	4000000	3	개발	9
3426	박영권	과장	4377	3000000	1	영업	8
3427	최종철	사원	3011	1500000	3	개발	9
4377	이성래	사장	∧	5000000	2	기획	10



## 4.4 SELECT문(계속)

### □ 자체 조인(self join)

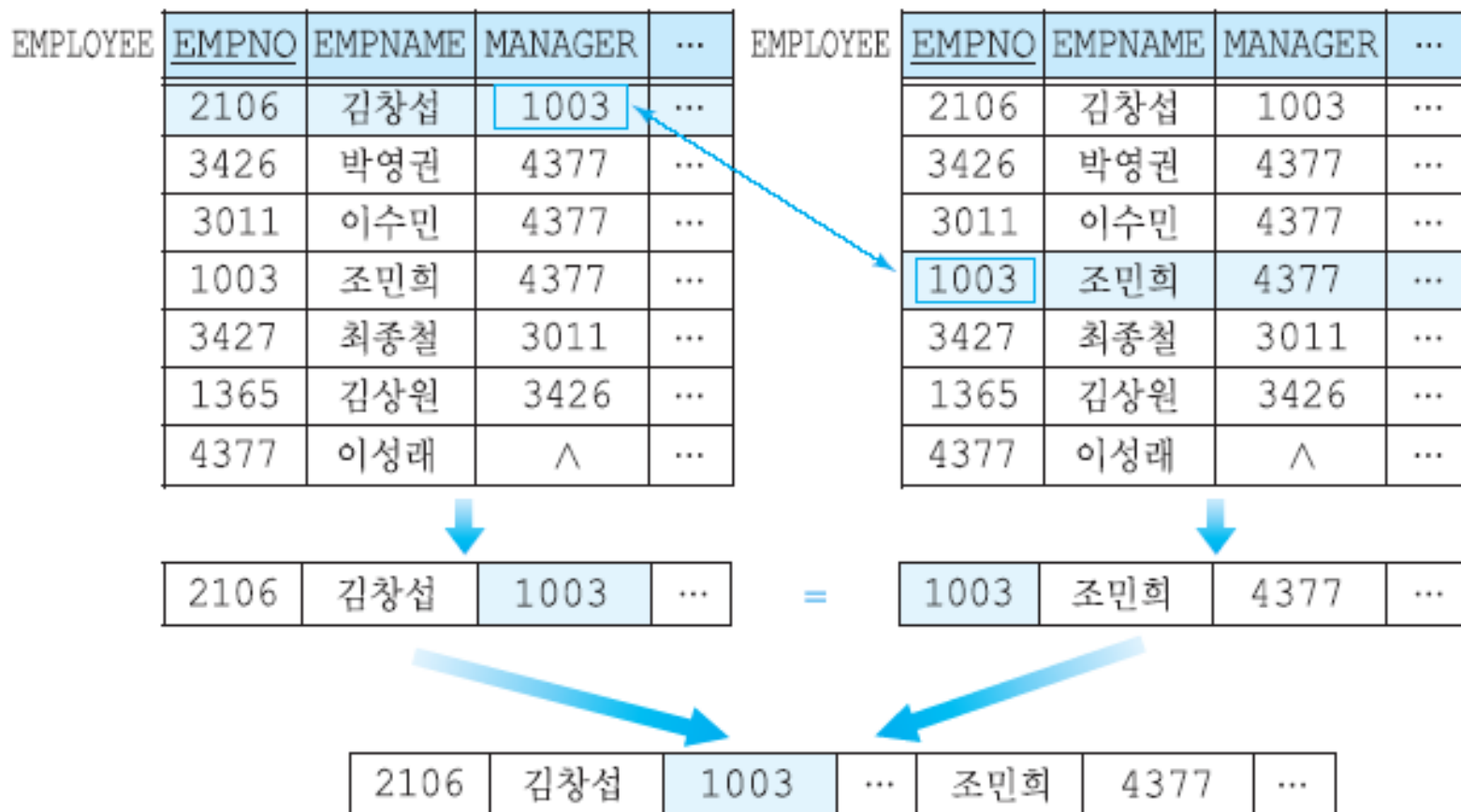
- ✓ 한 릴레이션에 속하는 튜플을 동일한 릴레이션에 속하는 튜플들과 조인하는 것
- ✓ 실제로는 한 릴레이션이 접근되지만 FROM절에 두 릴레이션이 참조되는 것처럼 나타내기 위해서 그 릴레이션에 대한 별칭을 두 개 지정해야 함

#### 예 : 자체 조인

질의: 모든 사원에 대해서 사원의 이름과 직속 상사의 이름을 검색하라.

```
SELECT      E.EMPNAME, M.EMPNAME
FROM        EMPLOYEE E, EMPLOYEE M
WHERE       E.MANAGER = M.EMPNO;
```

## 4.4 SELECT문(계속)



## 4.4 SELECT문(계속)

최종 결과 릴레이션은 아래와 같다.

E.EMPNAME	M.EMPNAME
김창섭	조민희
박영권	이성래
이수민	이성래
조민희	이성래
최종철	이수민
김상원	박영권

## 4.4 SELECT문(계속)

### 예 : 조인과 ORDER BY의 결합

질의: 모든 사원에 대해서 소속 부서이름, 사원의 이름, 직급, 급여를 검색하라. 부서 이름에 대해서 오름차순, 부서이름이 같은 경우에는 SALARY에 대해서 내림차순으로 정렬하라.

```
SELECT      DEPTNAME, EMPNAME, TITLE, SALARY
FROM        EMPLOYEE E, DEPARTMENT D
WHERE        E.DNO = D.DEPTNO
ORDER BY    DEPTNAME, SALARY DESC;
```

DEPTNAME	EMPNAME	TITLE	SALARY	
개발	이수민	부장	4000000	↓ 내림차순
개발	최종철	사원	1500000	
기획	이성래	사장	5000000	↓ 내림차순
기획	조민희	과장	3000000	
기획	김창섭	대리	2500000	↓ 내림차순
영업	박영권	과장	3000000	
영업	김상원	사장	1500000	

오름차순 ↓

## 4.4 SELECT문: 중첩 질의

### □ 중첩 질의(nested query)

- ✓ 질의의 WHERE 또는 FROM절에 다시 '(SELECT ... FROM ... WHERE ...)' 형태로 포함된 SELECT문 (괄호로 묶어서 표시함)
- ✓ **부질의**(subquery)라고 함
- ✓ 중첩 질의를 포함하는 질의를 **외부 질의**라고 부름
- ✓ INSERT, DELETE, UPDATE문에도 사용될 수 있음
- ✓ 중첩 질의의 결과는 세 가지 경우가 있음
  - 한 개의 스칼라값(단일 값)
  - 한 개의 애트리뷰트로 이루어진 릴레이션
  - 여러 애트리뷰트로 이루어진 릴레이션

## 4.4 SELECT문: 중첩 질의 (계속)



[그림 4.10] 중첩 질의의 구조

## 4.4 SELECT문: 중첩 질의 (계속)

### □ 한 개의 스칼라값이 반환되는 경우

- ✓ 스칼라(scala): 컬럼 값으로 사용될 수 있는 원자 값 (atomic value)
- ✓ WHERE 절에서 상수 또는 애트리뷰트가 사용될 위치에 나타날 수 있음

#### 예 : 한개의 스칼라 값이 반환되는 경우

질의: 박영권과 같은 직급을 갖는 모든 사원들의 이름과 직급을 검색하라.

```
SELECT      EMPNAME, TITLE
FROM        EMPLOYEE
WHERE       TITLE = (SELECT      TITLE
                     FROM        EMPLOYEE
                     WHERE       EMPNAME = '박영권' ) ;
```

중첩 질의

EMPNAME	TITLE
박영권	과장
조민희	과장

## 4.4 SELECT문: 중첩 질의 (계속)

- 한 개의 애트리뷰트로 이루어진 릴레이션이 반환되는 경우
  - ✓ 중첩 질의의 결과로 한 개의 애트리뷰트로 이루어진 다수의 튜플들이 반환될 수 있음
  - ✓ 외부 질의의 WHERE절에서 IN, ANY(SOME), ALL, EXISTS와 같은 연산자를 사용해야 함
    - **IN**: 한 애트리뷰트가 값들의 집합에 속하는가를 테스트
    - **ANY**: 한 애트리뷰트가 값들의 집합에 속하는 하나 이상의 값들과 어떤 관계를 갖는가를 테스트
    - **ALL**: 한 애트리뷰트가 값들의 집합에 속하는 모든 값들과 어떤 관계를 갖는가를 테스트
    - **EXISTS**: 중첩 질의의 결과가 빈 릴레이션인지 여부를 검사함
      - 중첩 질의의 결과가 빈 릴레이션이 아니면 참이 되고, 그렇지 않으면 거짓



## 4.4 SELECT문: 중첩 질의 (계속)

예 : IN

(3426 IN 

2106
3426
3011

)은 참이다.

(1365 IN 

2106
3426
3011

)은 거짓이다.

(1365 NOT IN 

2106
3426
3011

)은 참이다.

## 4.4 SELECT문: 중첩 질의 (계속)

예 : ANY

(3000000 < ANY 

2500000
3000000
4000000

)은 참이다.

(4000000 < ANY 

2500000
3000000
4000000

)은 거짓이다.

## 4.4 SELECT문: 중첩 질의 (계속)

예 : ALL

(3000000 < ALL 

2500000
3000000
4000000

)은 거짓이다.

(1500000 < ALL 

2500000
3000000
4000000

)은 참이다.

(3000000 = ALL 

2500000
3000000
4000000

)은 거짓이다.

(1500000 <> ALL 

2500000
3000000
4000000

)은 참이다.

## 4.4 SELECT문: 중첩 질의 (계속)

### 예 : IN을 사용한 질의

질의: 영업부나 개발부에 근무하는 직원들의 이름을 검색하라.

```
SELECT   EMPNAME  
FROM     EMPLOYEE  
WHERE    DNO IN
```

(1, 3)

```
(SELECT   DEPTNO  
  FROM     DEPARTMENT  
  WHERE    DEPTNAME = '영업' OR DEPTNAME = '개발' ) ;
```

## 4.4 SELECT문: 중첩 질의 (계속)

이 질의를 중첩 질의를 사용하지 않은 다음과 같은 조인 질의로 나타낼 수 있다. 실제로, 중첩 질의를 사용하여 표현된 대부분의 질의를 중첩 질의가 없는 조인 질의로 표현할 수 있다.

```
SELECT    EMPNAME
FROM      EMPLOYEE E, DEPARTMENT D
WHERE      E.DNO = D.DEPTNO
            AND (D.DEPTNAME = '영업' OR D.DEPTNAME = '개발');
```

EMPNAME
박영권
이수민
최종철
김상원

## 4.4 SELECT문: 중첩 질의 (계속)

### 예 : EXISTS를 사용한 질의

질의: 영업부나 개발부에 근무하는 직원들의 이름을 검색하라.

```
SELECT    EMPNAME
FROM      EMPLOYEE E
WHERE      EXISTS
              ( SELECT  *
                FROM    DEPARTMENT D
                WHERE    E.DNO = D.DEPTNO
                  AND (DEPTNAME = '영업' OR DEPTNAME = '개발') );
```

EMPNAME
박영권
이수민
최종철
김상원

## 4.4 SELECT문: 중첩 질의 (계속)

- 여러 애트리뷰트들로 이루어진 릴레이션이 반환되는 경우
  - ✓ 단일 애트리뷰트들로 이루어진 릴레이션이 반환되는 경우와 마찬가지로 IN, ANY, ALL, EXISTS 중에 하나를 사용하여 프레디킷을 만들어 사용할 수 있음
  - ✓ IN, ANY, ALL을 사용하는 경우에는 결과 릴레이션과 호환되는 튜플 구조를 갖는 튜플을 사용해서 비교해야 함

```
SELECT EMPNAME
FROM EMPLOYEE E
WHERE SALARY =< 1500000 AND (E.TITLE, E.DNO) IN
  (SELECT TITLE, DNO
   FROM EMPLOYEE
   WHERE SALARY > 1500000
  );
```

## 4.4 SELECT문: 중첩 질의 (계속)

### □ 상관 중첩 질의(correlated nested query)

- ✓ 중첩 질의의 WHERE절에 있는 프레디키트에서 외부 질의에 선언된 릴레이션의 일부 애트리뷰트를 참조하는 질의
- ✓ 중첩 질의의 수행 결과가 단일 값이든, 하나 이상의 애트리뷰트로 이루어진 릴레이션이든 외부 질의로 한 번만 결과를 반환하면 상관 중첩 질의가 아님
- ✓ 상관 중첩 질의에서는 외부 질의를 만족하는 각 튜플이 구해진 후에 중첩 질의가 수행됨
- ✓ 상관 중첩 질의는 외부 질의를 만족하는 튜플 수만큼 여러 번 수행될 수 있음

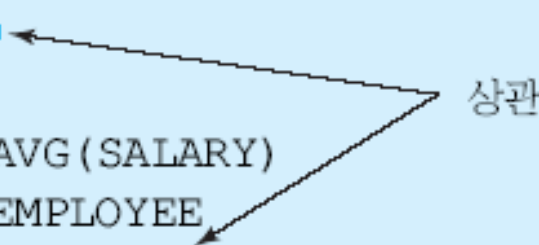


## 4.4 SELECT문: 중첩 질의 (계속)

### 예 : 상관 중첩 질의

질의: 자신이 속한 부서의 직원들의 평균 급여보다 많은 급여를 받는 직원들에 대해서 이름, 부서번호, 급여를 검색하라.

```
SELECT EMPNAME, DNO, SALARY
FROM EMPLOYEE E
WHERE SALARY >
      (SELECT AVG(SALARY)
       FROM EMPLOYEE
       WHERE DNO = E.DNO);
```



EMPNAME	DNO	SALARY
박영권	1	3000000
이수민	3	4000000
이성래	2	5000000

## 4.4 SELECT문: 중첩 질의 (계속)

### □ FROM 절에 사용된 중첩 질의

- ✓ FROM 절에 저장된 일반 테이블과 함께 중첩 질의를 사용할 수 있음
- ✓ 중첩 질의는 테이블 이름이 없으므로 alias를 사용하여 이름 부여

```
SELECT EMPNAME, DEPTNAME  
FROM EMPLOYEE E, (SELECT DEPTNO, DEPTNAME FROM DEPARTMENT) D  
WHERE E.DNO = D.DEPTNO AND TITLE = '과장';
```

## 4.5 INSERT, DELETE, UPDATE문

### □ INSERT문

- ✓ 기존의 릴레이션에 튜플을 삽입
- ✓ 참조되는 릴레이션에 튜플이 삽입되는 경우에는 참조 무결성 제약조건의 위배가 발생하지 않으나 참조하는 릴레이션에 튜플이 삽입되는 경우에는 참조 무결성 제약조건을 위배할 수 있음
- ✓ 릴레이션에 한 번에 한 튜플씩 삽입하는 것과 한 번에 여러 개의 튜플들을 삽입할 수 있는 것으로 구분
- ✓ 릴레이션에 한 번에 한 튜플씩 삽입하는 INSERT문

**INSERT**

**INTO**    릴레이션 ( 애트리뷰트<sub>1</sub>, ..., 애트리뷰트<sub>n</sub>)

**VALUES** ( 값<sub>1</sub>, ..., 값<sub>n</sub> );

## 4.5 INSERT, DELETE, UPDATE문(계속)

예 : 한 개의 튜플을 삽입

질의: DEPARTMENT 릴레이션에 (5, 연구, 0) 튜플을 삽입하는 INSERT문은 아래와 같다.

```
INSERT INTO DEPARTMENT  
VALUES (5, '연구', 0) ;
```

DEPARTMENT

DEPTNO	DEPTNAME	FLOOR
1	영업	8
2	기획	10
3	개발	9
4	총무	7
5	연구	0

## 4.5 INSERT, DELETE, UPDATE문(계속)

### □ INSERT문(계속)

- ✓ 릴레이션에 한 번에 여러 개의 튜플들을 삽입하는 INSERT문

```
INSERT  
INTO    릴레이션 ( 애트리뷰트1, ..., 애트리뷰트n)  
SELECT  ... FROM    ... WHERE    ...;
```

#### 예 : 여러 개의 튜플을 삽입

질의: EMPLOYEE 릴레이션에서 급여가 3000000 이상인 직원들의 이름, 직급, 급여를 검색하여 HIGH\_SALARY라는 릴레이션에 삽입하라. HIGH\_SALARY 릴레이션은 이미 생성되어 있다고 가정한다.

```
INSERT    INTO HIGH_SALARY (ENAME, TITLE, SAL)  
SELECT    EMPNAME, TITLE, SALARY  
FROM      EMPLOYEE  
WHERE      SALARY >= 3000000;
```

## 4.5 INSERT, DELETE, UPDATE문(계속)

### □ DELETE문

- ✓ 한 릴레이션으로부터 한 개 이상의 튜플들을 삭제함
- ✓ 참조되는 릴레이션의 삭제 연산의 결과로 참조 무결성 제약조건이 위배될 수 있으나, 참조하는 릴레이션에서 튜플을 삭제하면 참조 무결성 제약조건을 위배하지 않음
- ✓ DELETE문의 구문

**DELETE**

**FROM**      릴레이션

**WHERE**    조건 ;

## 4.5 INSERT, DELETE, UPDATE문(계속)

### 예 : DELETE문

질의: DEPARTMENT 릴레이션에서 4번 부서를 삭제하라.

```
DELETE FROM DEPARTMENT  
WHERE DEPTNO = 4;
```

## 4.5 INSERT, DELETE, UPDATE문(계속)

### □ UPDATE문

- ✓ 한 릴레이션에 들어 있는 튜플들의 애트리뷰트 값들을 수정
- ✓ 기본 키나 외래 키에 속하는 애트리뷰트의 값이 수정되면 참조 무결성 제약조건을 위배할 수 있음
- ✓ UPDATE문의 구문

**UPDATE**    릴레이션  
**SET**        애트리뷰트 = 값 또는 식[ , ...]  
**WHERE**     조건 ;

#### 예 : UPDATE문

질의: 사원번호가 2106인 사원의 소속 부서를 3번 부서로 옮기고, 급여를 5% 올려라.

```
UPDATE EMPLOYEE
SET     DNO = 3, SALARY = SALARY * 1.05
WHERE   EMPNO = 2106;
```

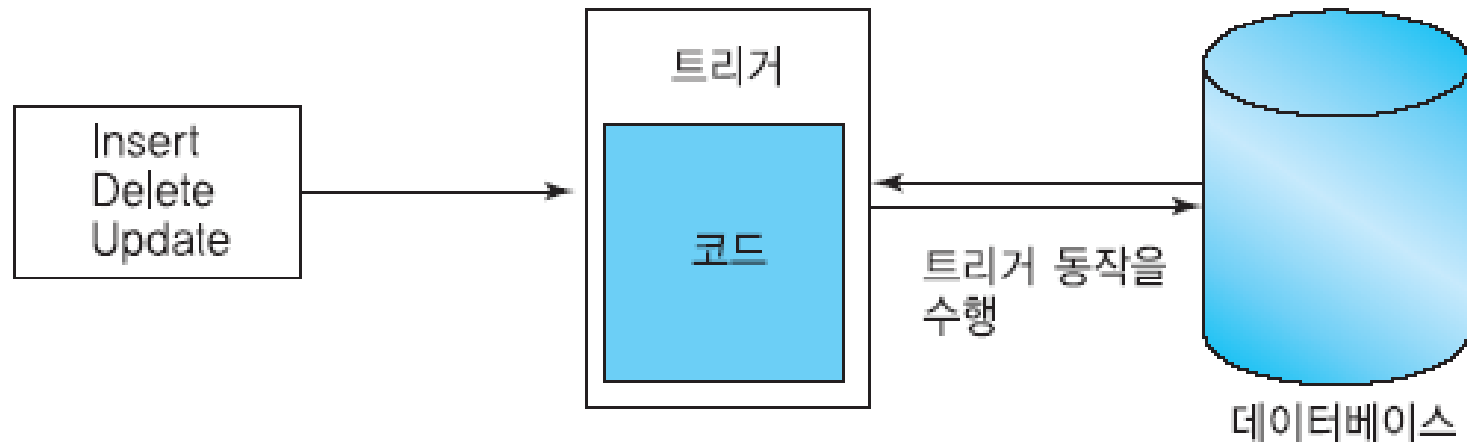


## 4.6 트리거(trigger)와 주장(assertion)

### □ 트리거

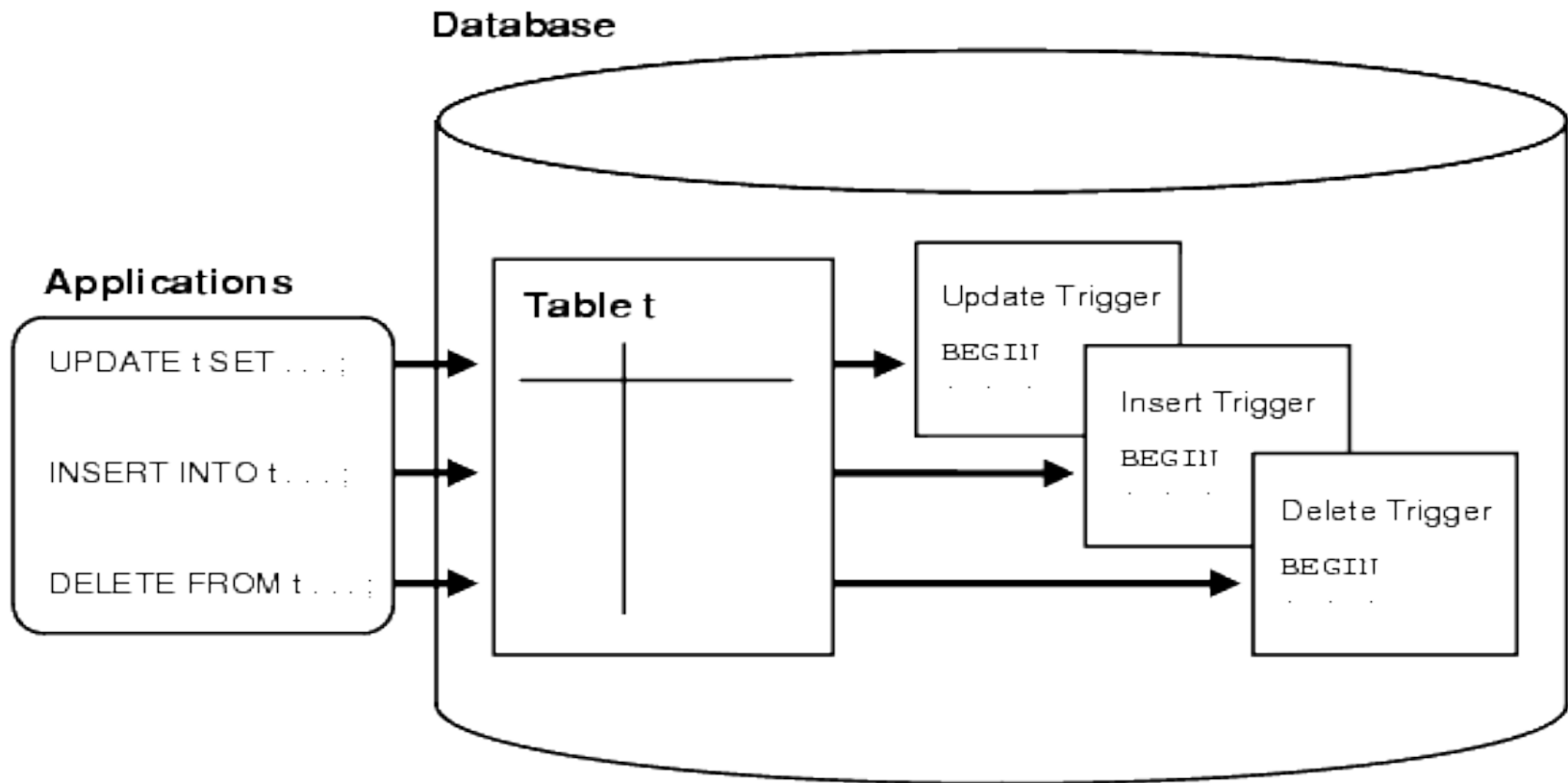
- ✓ 명시된 이벤트(데이터베이스의 갱신)가 발생할 때마다 DBMS가 자동적으로 수행하는, 사용자가 정의하는 문(프로시저)
- ✓ 데이터베이스의 무결성을 유지하기 위한 일반적이고 강력한 도구
- ✓ 테이블 정의시 표현할 수 없는 기업의 비즈니스 규칙들을 시행하는 역할
- ✓ 트리거 표현 요소
  - 트리거를 활성화시키는 사건인 이벤트 (event)
  - 트리거가 활성화되었을 때 수행되는 테스트인 조건 (condition)
  - 트리거가 활성화되고 조건이 참일 때 수행되는 문(프로시저)인 동작 (action)
- ✓ 트리거를 **이벤트-조건-동작(ECA)** 규칙이라고도 부름  
E는 Event, C는 Condition, A는 Action을 의미
- ✓ SQL3 표준에 포함되었으며 대부분의 상용 관계 DBMS에서 제공됨

## 4.6 트리거와 주장(계속)



[그림 4.11] 트리거의 개념

## 4.6 트리거와 주장(계속)



## 4.6 트리거와 주장(계속)

### □ 트리거(계속)

- ✓ SQL3에서 트리거의 형식

**CREATE TRIGGER** <트리거 이름>

**AFTER** <트리거를 유발하는 이벤트들이 OR로 연결된 리스트> **ON** <릴레이션> ← 이벤트

**[WHEN** <조건>

← 조건

**BEGIN** <SQL문(들)> **END**

← 동작

- ✓ 이벤트의 가능한 예로는 테이블에 튜플 삽입, 테이블로부터 튜플 삭제, 테이블의 튜플 수정 등이 있음
- ✓ 조건은 임의의 형태의 프레디케이트
- ✓ 동작은 데이터베이스에 대한 임의의 갱신
- ✓ 어떤 이벤트가 발생했을 때 조건이 참이 되면 트리거와 연관된 동작이 수행되고, 그렇지 않으면 아무 동작도 수행되지 않음
- ✓ 삽입, 삭제, 수정 등이 일어나기 전(before)에 동작하는 트리거와 일어난 후(after)에 동작하는 트리거로 구분

## 4.6 트리거와 주장(계속)

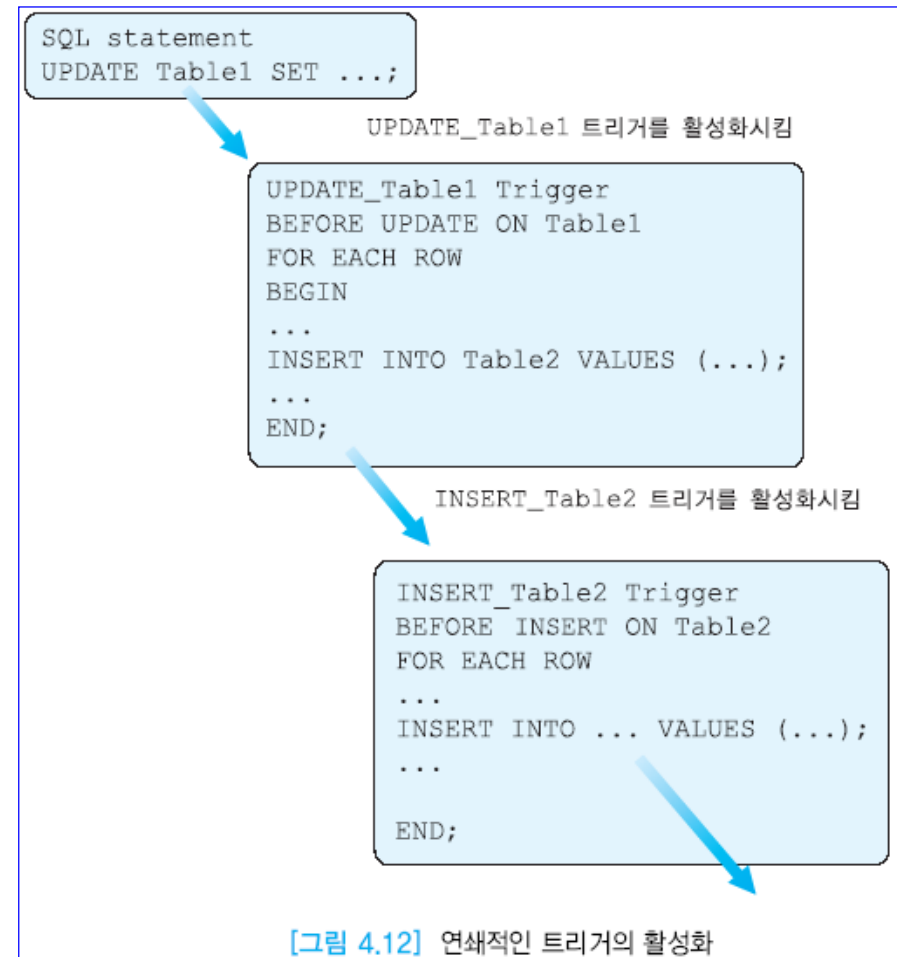
### 예 : 트리거

새로운 사원이 입사할 때마다, 사원의 급여가 1500000 미만인 경우에는 급여를 10% 인상하는 트리거를 작성하라. 여기서 이벤트는 새로운 사원 투플이 삽입될 때, 조건은 급여 < 1500000, 동작은 급여를 10% 인상하는 것이다. 오라클에서 트리거를 정의하는 문장은 SQL3의 트리거 정의문과 동일하지는 않다.

```
CREATE TRIGGER RAISE_SALARY
AFTER INSERT ON EMPLOYEE
REFERENCING NEW AS newEmployee
FOR EACH ROW
WHEN      (newEmployee.SALARY < 1500000)
UPDATE EMPLOYEE
SET       newEmployee.SALARY = SALARY * 1.1
WHERE     EMPNO = newEmployee.EMPNO;
```

## 4.6 트리거와 주장(계속)

- 연쇄적으로 활성화되는 트리거
  - ✓ 하나의 트리거가 활성화되어 이 트리거 내의 한 SQL문이 수행되고, 그 결과로 다른 트리거를 활성화하여 그 트리거 내의 SQL문이 수행될 수 있음



## 4.6 트리거와 주장(계속)

### □ 주장 (ASSERTION)

- ✓ SQL3에 포함되어 있으나 대부분의 상용 관계 DBMS가 아직 지원하고 있지 않음
- ✓ 트리거는 제약조건을 위반했을 때 수행할 동작을 명시하는 것이고, 주장은 제약조건을 위반하는 연산이 수행되지 않도록 함
- ✓ 주장의 구문  
`CREATE ASSERTION 이름`  
`CHECK 조건 ;`
- ✓ 트리거보다 좀더 일반적인 무결성 제약조건
- ✓ DBMS는 주장의 프레디키트를 검사하여 만일 참이면 주장을 위배하지 않는 경우이므로 데이터베이스 수정이 허용됨
- ✓ 일반적으로 두 개 이상의 테이블에 영향을 미치는 제약조건을 명시하기 위해 사용됨

## 4.6 트리거와 주장(계속)

- 대부분의 주장은 **NOT EXISTS**를 포함
- 주장에는 “모든 x가 F를 만족한다”를 이와 동치인 “ $\neg F$ 를 만족하는 x가 존재하지 않는다”로 표시

### 예 : 주장

STUDENT(학생) 릴레이션과 ENROLL(수강) 릴레이션의 스키마가 아래와 같다. STUDENT 릴레이션의 기본 키는 STNO이다. ENROLL 릴레이션의 STNO는 STUDENT 릴레이션의 기본 키를 참조한다.

```
STUDENT(STNO, STNAME, EMAIL, ADDRESS, PHONE)
ENROLL(STNO, COURSENO, GRADE)
```

ENROLL 릴레이션에 들어 있는 STNO는 반드시 STUDENT 릴레이션에 들어 있는 어떤 학생의 STNO를 참조하도록 하는 주장을 정의하려 한다. 다시 말해서 STUDENT 릴레이션에 없는 어떤 학생의 학번이 ENROLL 릴레이션에 나타나는 것을 허용하지 않으려고 한다. 대부분의 주장은 아래의 예처럼 NOT EXISTS를 포함한다.

```
CREATE ASSERTION EnrollStudentIntegrity
CHECK (NOT EXISTS
      (SELECT *
      FROM ENROLL
      WHERE STNO NOT IN
            (SELECT STNO FROM STUDENT))) ;
```

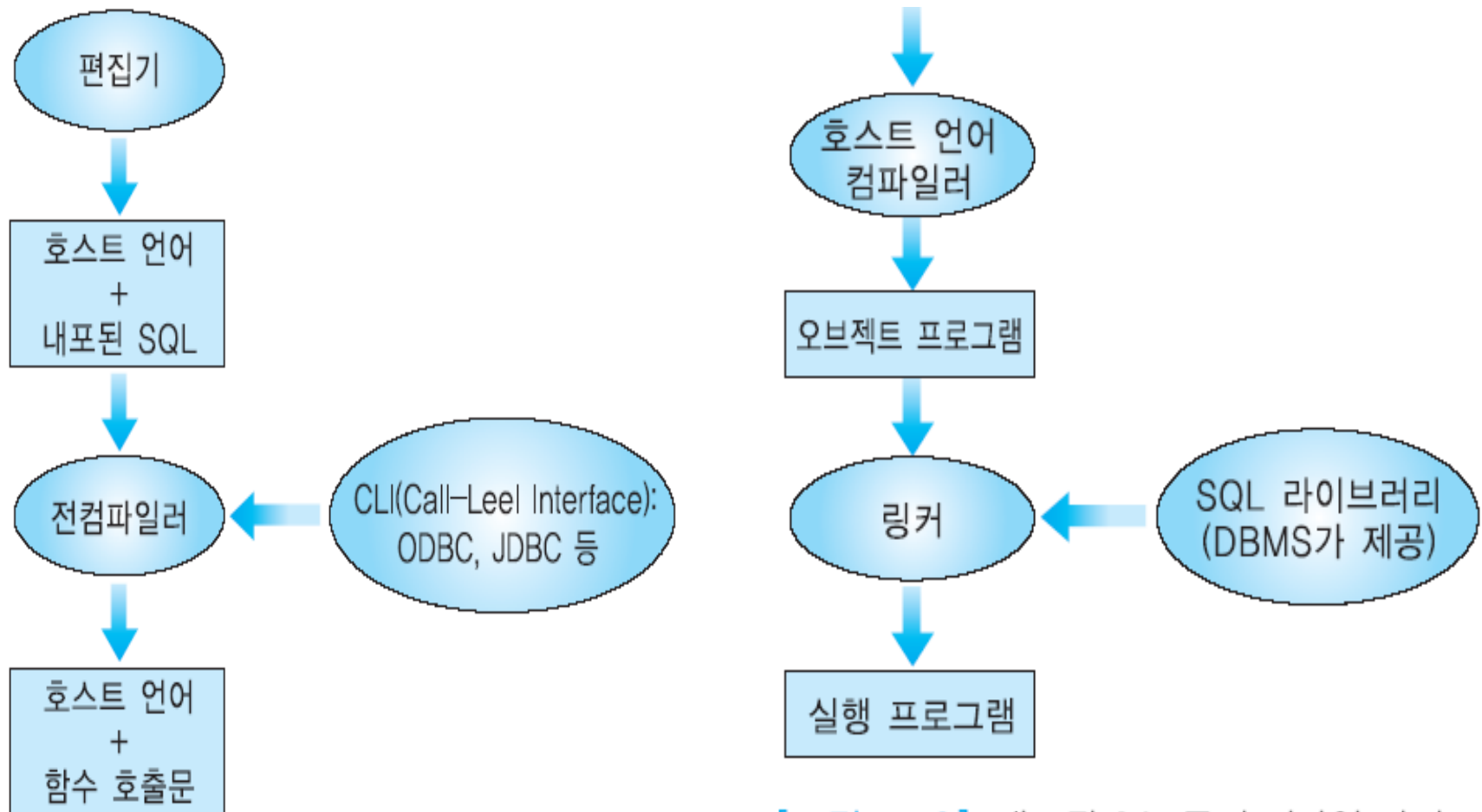


## 4.7 내포된 SQL

### □ 내포된 SQL(embedded SQL)

- ✓ SQL이 호스트 언어의 완전한 표현력을 갖고 있지 않기 때문에 모든 질의를 SQL로 표현할 수는 없음
- ✓ SQL은 호스트 언어가 갖고 있는 조건문(IF문), 반복문(WHILE문), 입출력 등과 같은 동작, 사용자와의 상호 작용, 질의 결과를 GUI로 보내는 등의 기능을 갖고 있지 않음
- ✓ C, C++, 코볼, 자바 등의 언어로 작성하는 프로그램에 SQL문을 삽입하여, 데이터베이스를 접근하는 부분을 SQL이 맡고 SQL에 없는 기능은 호스트 언어로 작성하는 것이 필요
- ✓ 호스트 언어에 포함되는 SQL문을 **내포된 SQL**이라 부름
- ✓ 데이터 구조가 불일치하는 문제(**impedance mismatch** 문제)

## 4.7 내포된 SQL(계속)



[그림 4.13] 내포된 SQL문의 컴파일 과정

## 4.7 내포된 SQL(계속)

### □ Pro\*C


- ✓ 오라클에서 C 프로그램에 SQL 문을 내포시키는 방법
- ✓ 내포된 SQL 문이 포함된 소스파일의 확장자는 .pc
- ✓ .pc 파일을 Pro\*C를 통하여 전컴파일하면 .c인 소스 프로그램이 생성됨
- ✓ 윈도우7 환경에서 Pro\*C를 실행하려면 비주얼 스튜디오 6.0 등의 통합 개발 환경이 필요

## 4.7 내포된 SQL (계속)

- ❑ 호스트 변수 (host variable)
  - ✓ SQL문에 포함된 C 프로그램의 변수
  - ✓ 호스트 언어와 SQL 문 사이에 통신을 위해 사용
    - ✓ SQL 문에 사용될 데이터 값을 입력하거나
    - ✓ SQL 문의 결과를 출력
  - ✓ 호스트 변수를 SQL 문에서 사용할 때 콜론(:)을 붙여서 사용
  - ✓ DECLARE SECTION을 이용해서 선언
    - ✓ Oracle은 DECLARE SECTION을 사용하지 않는 것도 지원 (표준은 아님)

```
EXEC SQL BEGIN DECLARE SECTION;  
  int      no;  
  varchar title[10];  
EXEC SQL END DECLARE SECTION;
```

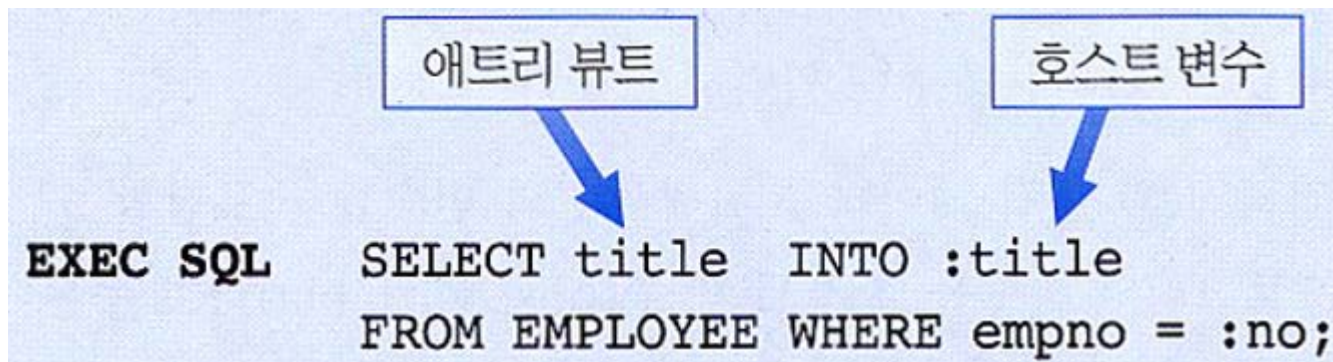
호스트 변수 선언



## 4.7 내포된 SQL(계속)

### □ 정적인 SQL문

- ✓ C 프로그램에 내포된 완전한 SQL문
- ✓ 입력값과 출력 데이터를 위해서 C 프로그램의 변수들을 포함할 수 있음



## 4.7 내포된 SQL(계속)

### □ 동적인 SQL문

- ✓ 불완전한 SQL문으로서 일부 또는 전부를 질의가 수행될 때 입력 가능
- ✓ 응용을 개발할 때 완전한 SQL문의 구조를 미리 알고 있지 않아도 됨
- ✓ 문자열 형 변수에 담아서 표현함으로써 동적으로 변경 가능
- ✓ 컴파일 시점에 SQL 문을 알지 못함

```
/* Update column in table using DYNAMIC SQL*/  
strcpy(hostVarStmtDyn, "UPDATE staff SET salary = salary + 1000 WHERE dept = :v");  
EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;  
EXEC SQL EXECUTE StmtDyn USING :dept;
```

비교: EXEC SQL EXECUTE IMMEDIATE :hostVarStmtDyn USING :dept;



## 4.7 내포된 SQL(계속)

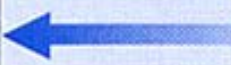
### 예 : 호스트 변수

아래의 부분 프로그램은 호스트 변수를 사용한 C 프로그램의 예를 보여준다. 이 프로그램은 사용자에게 사원의 번호를 입력하도록 하고, 사용자가 입력한 값을 호스트 변수 no에 저장한다. 그 다음에 프로그램은 DBSERVER 데이터베이스의 EMPLOYEE 릴레이션에서 그 사원의 직급을 검색하여 호스트 변수 title에 저장한다.

```
#include <stdio.h>
```

```
EXEC SQL BEGIN DECLARE SECTION;  
    int      no;  
    varchar title[10];  
EXEC SQL END DECLARE SECTION;
```

호스트 변수 선언



```
EXEC SQL INCLUDE SQLCA.H;    /* SQL 통신 영역 */
```

## 4.7 내포된 SQL(계속)

```
void main()
{
    char *uid = "KIM/kim@DBSERVER";
    EXEC SQL WHENEVER SQLERROR GOTO errexit;
    EXEC SQL CONNECT :uid;      /* DBSERVER 데이터베이스에 사용자
                                KIM으로 접속 */

    printf("Enter employee number : ");
    scanf("%d",&no);
```



## 4.7 내포된 SQL(계속)

애틀리 뷰트

호스트 변수

```
EXEC SQL    SELECT title INTO :title
              FROM EMPLOYEE WHERE empno = :no;
printf("\nAuthor's title is %s.\n", title);
EXEC SQL COMMIT WORK;
exit(0);
errexit:
EXEC SQL ROLLBACK WORK;
exit(1);
}
```

## 4.7 내포된 SQL(계속)

### □ 불일치 문제와 커서

- ✓ 호스트 언어는 단일 변수/레코드 위주의 처리(튜플 위주의 방식)를 지원하는 반면에 SQL은 데이터 레코드들의 처리(집합 위주의 방식)를 지원하기 때문에 불일치 문제가 발생함
- ✓ 불일치 문제를 해결하기 위해서 커서(cursor)가 사용됨
- ✓ 두 개 이상의 튜플들을 검색하는 SQL문에 대해서는 반드시 커서를 선언하고 사용해야 함
- ✓ 커서는 한 번에 한 튜플씩 가져오는 수단

## 4.7 내포된 SQL(계속)

### □ 커서

- ✓ DECLARE CURSOR문을 사용하여 커서를 정의함
- ✓ OPEN cursor문은 질의를 수행하고, 질의 수행 결과의 첫 번째 튜플 이전을 커서가 가리키도록 한다. 이 것이 커서의 현재 튜플
- ✓ 그 다음에 FETCH문은 커서를 다음 튜플로 이동하고, 그 튜플의 애트리뷰트 값들을 FETCH문에 명시된 호스트 변수들에 복사함
- ✓ CLOSE cursor는 커서를 닫음

## 4.7 내포된 SQL(계속)

### 예 : 정적인 커서

아래의 부분 프로그램은 정적인 커서의 예를 보여준다.

```
EXEC SQL BEGIN DECLARE SECTION;
char name[ ] = "박영권";
char title[10] ;
EXEC SQL END DECLARE SECTION;

EXEC SQL
    DECLARE title_cursor CURSOR FOR
    SELECT title FROM employee WHERE empname = :name;

EXEC SQL OPEN title_cursor;
EXEC SQL FETCH title_cursor INTO :title;
```

## 4.7 내포된 SQL(계속)

- ❑ 여러 튜플을 읽어 오기 위해서는 루프 내의 FETCH문 사용

```
EXEC SQL WHENEVER NOT FOUND GOTO NotFoundLabel;
for (;;) {
    ...
    EXEC SQL FETCH ...;
    ...
}
...
NotFoundLabel:
...
```

- ✓ 루프 종료 조건: 결과 집합이 비었거나 더 이상의 가져올 튜플이 없으면  
FETCH문은 'no data found' 에러를 발생시키고, WHENEVER NOT  
FOUND가 이를 인지함
  - ✓ SQLCODE를 검사하는 방법을 사용할 수도 있음 (다음에 설명)



## 4.7 내포된 SQL(계속)

### □ WHENEVER

- ✓ 자동적인 에러 검사와 에러 처리를 위한 구문
- ✓ 구문: **WHENEVER** <조건> <동작>
  - <조건>
    - **NOT FOUND**: WHERE절을 만족하는 튜플이 없거나, SELECT INTO 혹은 FETCH가 row를 리턴하지 않음
    - **SQLERROR**: 에러가 발생한 경우
    - **SQLWARNING**: 경고가 발생한 경우
  - <동작>: **CONTINUE, GOTO, STOP, DO**
    - **CONTINUE**: WHENEVER 문을 사용하지 않는 것과 같은 효과
    - **DO {function call | CONTINUE | BREAK}**: 프로그램 제어를 이동
    - **STOP**: 프로그램 종료. COMMIT되지 않은 WORK은 ROLLBACK

## 4.7 내포된 SQL(계속)

### □ CURSOR를 이용한 UPDATE

- ✓ 커서의 현재 투플을 업데이트하려면 **CURRENT OF** 절을 사용하면 됨
- ✓ CURSOR를 선언할 때 **FOR UPDATE OF** 키워드를 선택적으로 추가
  - 개발자가 추가하지 않아도, UPDATE 또는 DELETE 문에 CURRENT OF 절이 나오면 전처리가 필요 시 추가함
- ✓ 제약: Index-organized 테이블에는 사용금지. 한 테이블의 애트리뷰트들만 수정 가능

```
EXEC SQL DECLARE title_cursor CURSOR FOR  
SELECT title FROM EMPLOYEE WHERE empname = :name  
FOR UPDATE OF title;
```

...

```
EXEC SQL FETCH title_cursor INTO :title;  
EXEC SQL UPDATE EMPLOYEE SET title = '상무'  
WHERE CURRENT OF title_cursor;
```

## 4.7 내포된 SQL(계속)

### □ SQL 통신 영역(SQLCA: SQL Communications Area)

- ✓ C 프로그램에 내포된 SQL문에 발생하는 에러들을 사용자에게 알려줌
- ✓ 사용자는 SQLCA 데이터 구조(SQLCA.H)의 에러 필드와 상태 표시자를 검사하여 내포된 SQL문이 성공적으로 수행되었는가 또는 비정상적으로 수행되었는가를 파악할 수 있음
- ✓ SQLCA 데이터 구조 중에서 가장 중요하고 널리 사용되는 필드는 `sqlcode` 멤버 변수
- ✓ `sqlcode`의 값이 0이면 마지막에 내포된 SQL문이 성공적으로 끝났음을 의미
- ✓ SQLCA를 사용하기 위해서는 아래와 같은 문장을 포함해야 함  
`EXEC SQL INCLUDE SQLCA.H;`    또는  
`#include <sqlca.h>`



## 4.7 내포된 SQL(계속)

- 오라클 통신 영역(**ORACA**: Oracle Communications Area)
  - ✓ SQLCA라는 SQL 표준을 오라클에서 확장한 구조체
  - ✓ sqlca에서 얻을 수 있는 정보 외에 추가로 필요한 정보를 호스트 프로그램에게 제공하기 위한 구조체

## 4.7 내포된 SQL(계속)

### 예 : SQLCODE

아래의 부분 프로그램은 SQLCODE를 사용하여, 내포된 SQL문이 성공적으로 끝났는가를 검사한다.

```
EXEC SQL DECLARE c1 CURSOR FOR
  SELECT empno, empname, title, manager, salary, dno
  FROM employee;
EXEC SQL OPEN c1;
while (SQLCODE == 0)
{
  /* 데이터를 성공적으로 가져올 수 있으면 SQLCODE의 값이 0이다. */
  EXEC SQL
    FETCH c1 INTO :eno, :name, :title, :manager, :salary,
    :dno;
  if (SQLCODE == 0)
    printf("%4d %12s %12s %4d %8d %2d",
      eno, name, title, manager, salary, dno);
}
EXEC SQL CLOSE c1;
```

- SQLCODE는 잘못됨
- sqlca.sqlcode로 써야 함
- SQLCODE는 별도의 SQLCA와 다른 변수임

## 4.7 내포된 SQL(계속)

### ❑ 에러 메시지: sqlglm() 함수 사용

- ✓ 에러가 난 것을 먼저 확인하고 사용해야 함
  - ✓ WHENEVER 사용 또는 SQLCODE (or sqlca.sqlcode) 값이 0이 아닐 때

```
EXEC SQL WHENEVER SQLERROR DO sql_error();
```

```
...  
/* other statements */  
...  
Sql_error()  
{  
    char msg[200];  
    size_t buf_len, msg_len;  
  
    buf_len = sizeof (msg);  
    sqlglm(msg, &buf_len, &msg_len); /* note use of pointers */  
    printf("%.*s\n\n", msg_len, msg);  
    exit(1);  
}
```

## 4.7 내포된 SQL(계속)

### □ SQLSTATE 상태 변수

- ✓ SQL92 표준에서는 Embedded SQL의 에러 처리를 위해 SQLSTATE 변수를 도입
  - ✓ 이전에는 SQLCODE를 사용함 (SQL92에서 deprecated 됨)
  - ✓ Oracle의 전처리기 옵션에서 mode=ANSI 로 하여 사용 가능
    - mode=ORACLE이면 SQLCA가 사용됨
- ✓ 예외의 유형을 구분하는 CLASS CODE (2자리 문자)와 구체적인 예외를 나타내는 SUBCLASS CODE (3자리 문자)의 총 5자리 문자의 값을 가짐
  - ✓ 각 자리는 0..9 또는 A..Z
- ✓ 선언(declaration) 필요: `char SQLSTATE[6];`
- ✓ 참조: [https://docs.oracle.com/cd/B10501\\_01/appdev.920/a97269/pc\\_09err.htm](https://docs.oracle.com/cd/B10501_01/appdev.920/a97269/pc_09err.htm)

## 4.7 내포된 SQL(계속)

### □ 지시 변수 (indicator variables)

- ✓ NULL 값 여부 등 호스트 변수에 대한 추가적인 정보 제공
- ✓ 2바이트 정수로 표현
- ✓ 호스트 변수 바로 다음에 지시 변수를 선택적으로 추가
  - 지시 변수도 앞에 콜론(:) 붙임
  - 호스트 변수와 지시 변수 사이에 선택적으로 INDICATOR 키워드 사용 가능

short indicator\_var;

EXEC SQL SELECT xyz INTO :host\_var:indicator\_var FROM ...;

EXEC SQL INSERT INTO R VALUES(:host\_var INDICATOR :indicator\_var, ...);

#### 지시 변수 해석: SELECT INTO 문

-1	(호스트 변수 값이) NULL
0	intact (온전한) value
>0	truncated value, 원래 길이 저장
-2	truncated value, 원래 길이 모름

#### INSERT/UPDATE 문

-1	(호스트 변수 값이) NULL
>=0	normal value