

# 8장. SQL의 시스템 관점

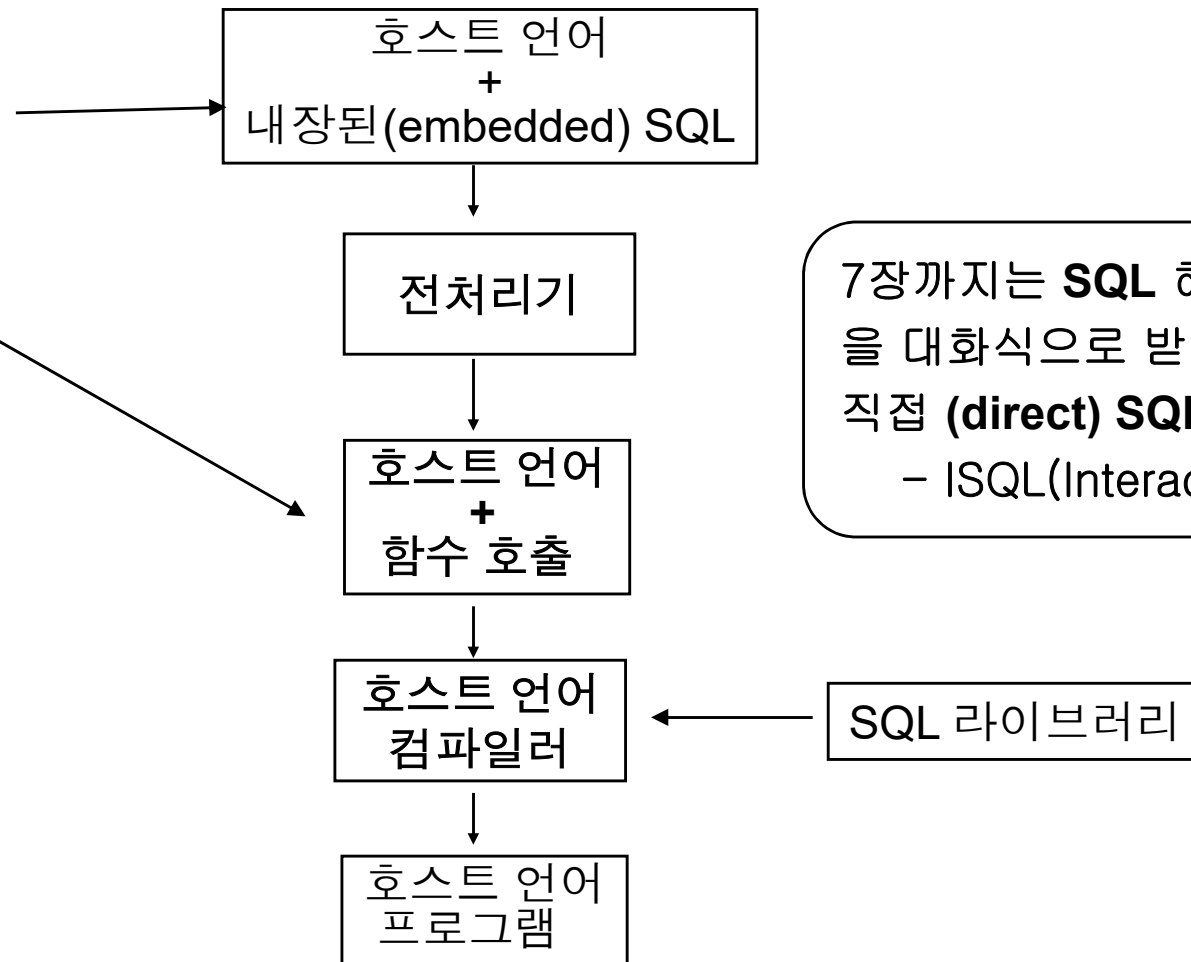
---

- ◆ 프로그래밍 환경에서의 **SQL**
- ◆ **SQL**에서의 트랜잭션
- ◆ **SQL** 환경
- ◆ **SQL2**에서의 보안과 사용자 권한



# 프로그래밍 환경에서의 SQL

## ■ SQL문들이 내장된 프로그램의 처리 과정



7장까지는 **SQL** 해석기가 **SQL** 명령문을 대화식으로 받아들여 수행하는 직접 (**direct**) **SQL** 예제들을 다루었다.  
- ISQL(Interactive SQL) v.s. ESQL

# 프로그래밍 환경에서의 SQL (계속)

---

## ◆ 임피던스(impedence) 불일치 문제

- **SQL**의 데이터 모델이 호스트 언어의 모델과 다르다.
  - » 대부분의 프로그래밍 언어는 집합을 직접 표현하지 못한다.
  - » **SQL**은 포인터, 배열 등의 데이터 타입을 직접 사용할 수 없다.
- **SQL**은 다양한 계산 기능과 출력 형태를 지원하지 않는다.
  - » 숫자 **n**의 계승(**factorial**)을 계산하는 **SQL** 질의를 작성할 수 없다
  - » **SQL**은 그래픽과 같은 편리한 모양으로 출력을 직접 포맷할 수 없다.
- ☞ **DB** 프로그래밍에는 **SQL**과 기존의 프로그래밍 언어가 모두 필요하다.
- ☞ **SQL**과 함께 사용되는 기존의 언어를 호스트 언어(**host language**)라 함
  - **ESQL/C(Embedded SQL/C)** : **C** 프로그램에 내장된 **SQL** 처리 도구
  - **Oracle**에서는 **Pro\*C/C++**이라 함

# 프로그래밍 환경에서의 SQL (계속)

---

## ◆ SQL/호스트 언어 인터페이스

- 호스트 변수(**host variable**) (또는 공유 변수(**shared variable**))
  - » DB와 호스트 언어 프로그램 사이의 정보 교환
  - » **SQL** 문에서 사용될 때에는 변수 앞에 콜론을 붙인다.
  - » 호스트 언어 문에서는 콜론 없이 그대로 사용된다.
- 호스트 언어 내에서 : **EXEC SQL** <sql 문>

## ■ SQLSTATE

- 다섯 문자 배열인 특수 변수
- **SQL** 라이브러리 함수가 호출될 때마다, 함수 실행시에 발생한 문제를 나타내는 코드가 **SQLSTATE** 변수에 저장
  - » '00000' : 아무런 문제가 없음, **SUCCESS**
  - » '01000' : 경고(**warning**)가 있음, **SUCCESS with Warning**
  - » '02000' : **SQL** 질의 결과에 해당하는 튜플을 찾을 수 없음, **Not Found**

# 프로그래밍 환경에서의 SQL (계속)

## ◆ 선언부(declare section)와 호스트 변수

[예] 사용자로부터 스튜디오의 이름과 주소를 입력 받아 적절한 튜플을 Studio 릴레이션에 삽입하는 C 함수의 개략적인 형태.

```
Void  getStudio()  {
```

선언부  
(host variable)

```
    EXEC SQL BEGIN DECLARE SECTION;
```

```
        char studioName[15], studioAddr[50];
```

```
        char SQLSTATE[6];
```

```
    EXEC SQL END DECLARE SECTION;
```

```
        /* 스튜디오 이름과 주소를 입력하라는 요청을 프린트하고, 이 입력을  
        studioName과 studioAddr 변수로 읽어들인다. */
```

내장된  
SQL 문

```
    EXEC SQL INSERT INTO Studio (name, address)
```

```
        VALUES (:studioName, :studioAddr);
```

```
    if (strcmp(SQLSTATE, "00000")) /* 실행결과 확인 */
```

```
        printf("Insert Error !\n");
```

```
}
```

# 프로그래밍 환경에서의 SQL (계속)

---

## ◆ 내장(embed)될 수 있는 SQL 문

- 결과를 반환하지 않는(즉, 질의가 아닌) 모든 SQL 문
  - » 테이블이나 뷰와 같은 스키마 요소를 생성, 변경, 삭제시키는 SQL문

## ◆ SELECT-FROM-WHERE 질의는 직접 내장될 수 없음

- 임피던스 불일치 : 질의는 튜플들의 백(bag)을 결과로 생성하는 반면, 호스트 언어들은 백 데이터 타입을 직접 지원하지 않음

## ■ 질의의 결과를 호스트 언어 프로그램과 연결

- 단일행 선택연산(single-row select) : 질의가 하나의 튜플만 생성
- 커서(cursor) : 커서는 결과 릴레이션의 모든 튜플들을 튜플 단위로 가리킨다. 각 튜플은 차례대로 호스트 변수로 인출(fetch)되어 호스트 언어 프로그램에 의해 처리될 수 있다.

# 프로그래밍 환경에서의 SQL (계속)


---

## ◆ 단일행 선택연산 문장 : SELECT INTO

- 키워드 **INTO**와 호스트 변수들의 리스트
- 하나의 튜플을 생성하는 질의에서 사용

```
Void printNetWorth() {  
    /* 선언부 */  
  
    EXEC SQL SELECT netWorth  
        INTO :presNetWorth  
        FROM Studio, MovieExec  
        WHERE presC# = cert# AND Studio.name = :studioName;  
}
```

질의 결과를 저장



- ☞ 질의의 결과로 튜플이 전혀 없거나 둘 이상의 튜플이 생기면 호스트 변수에는 어떤 값도 저장되지 않으며 해당 코드가 변수 **SQLSTATE**에 기록된다.

# 커서

---

- 커서(**cursor**)는 둘 이상의 튜플을 생성하는 질의를 위해 사용
  - » 커서는 릴레이션을 튜플 단위로 참조할 수 있으며, 한 순간에는 하나의 튜플을 가리킨다.
- ◆ 커서 선언

```
EXEC SQL DECLARE <커서 이름> CURSOR FOR <질의>
```
- ◆ 커서 열기(**open**)
  - 릴레이션의 첫번째 튜플을 검색할 수 있는 위치로 커서를 초기화

```
EXEC SQL OPEN <커서 이름>
```
- ◆ 인출 문(**fetch statement**)
  - 커서가 움직이는 영역인 릴레이션에서 다음 튜플을 가져온다.

```
EXEC SQL FETCH (FROM) <커서 이름> INTO <변수 리스트>
```
- ◆ 커서 닫기(**close**)

```
EXEC SQL CLOSE <커서 이름>
```



# 커서 (계속)

[예] 각 10자리 단위 (즉 10, 100, 1000, ...) 별로 이 단위에 해당하는 재산을 가진 영화 임원들의 수를 알아보고자 한다. (예를 들어, 재산이 100만 단위(즉 7자리수)인 영화 임원은 열 명 있다는 등).

```
EXEC SQL DECLARE execCursor CURSOR FOR
        SELECT netWorth FROM MovieExec;
EXEC SQL OPEN execCursor;
for (i=0; i < 15; i++) counts[i] = 0;
while(1) {
    EXEC SQL FETCH FROM execCursor INTO :worth;
    if (NO_MORE_TUPLES) break;
    digits = 1;
    while (worth /= 10 > 0) digits++;
    if (digits <= 14) counts[digits]++;
}
EXEC SQL CLOSE execCursor;
/* 결과 출력 */
```

```
#define NO_MORE_TUPLES
!(strcmp(SQLSTATE, "02000"))
```

# 커서 (계속)

---

## ◆ 커서를 통한 변경 : WHERE CURRENT OF <cursor-name>

- 튜플의 갱신이나 삭제

[예] 영화 임원의 재산이 1000 미만이면 삭제하고, 그렇지 않으면 재산을 두 배로 갱신한다.

```
EXEC SQL OPEN   execCursor;
while(1) {
    EXEC SQL FETCH FROM execCursor INTO :worth;
    if (NO_MORE_TUPLES) break;
    if (worth < 1000)
        EXEC SQL DELETE FROM MovieExec
            WHERE CURRENT OF execCursor;
    else
        EXEC SQL UPDATE MovieExec
            SET netWorth = 2 * netWorth
            WHERE CURRENT OF execCursor;
}
EXEC SQL CLOSE execCursor;
```

# 커서 (계속)

---

- ◆ 튜플의 인출 순서 : ORDER BY
  - 명시된 순서로 튜플들을 인출

```
EXEC SQL DECLARE  movieStarCursor CURSOR FOR
                SELECT  title, year, studioName, starName
                FROM      Movie, StarsIn
                WHERE     title = movieTitle  AND  year = movieYear
                        AND length >= :Length
                ORDER  BY  year,  title;
```



# 커서 (계속)

---

## ◆ 병행수행 갱신(**concurrent updates**)들로부터의 보호

### ■ INSENSITIVE CURSOR

- 커서를 열고 참조하는 동안에는 병행수행에 의해 이 릴레이션에 생긴 변화가 그 커서에 반영되지 않도록 한다.

```
EXEC SQL DECLARE movieStarCursor INSENSITIVE CURSOR FOR
```

» **movieStarCursor**가 열린 후부터 닫힐 때까지 **Movie**와 **StarsIn** 릴레이션에 대한 변화가 이 커서에 영향을 주지 못하도록 보장한다.

### ■ FOR READ ONLY

- 이 커서를 통해 삽입, 삭제, 갱신 등을 하지 않을 것이라고 선언

```
EXEC SQL DECLARE movieStarCursor CURSOR FOR
      SELECT title, year, studio, starName
      FROM Movie, StarsIn
      WHERE title = movieTitle AND year = movieYear
      ORDER BY year, title
      FOR READ ONLY;
```

# 커서 (계속)

---

## ◆ 전후이동(**Scrolling**) 커서 : SCROLL CURSOR

- 릴레이션의 튜플들 사이를 움직이는 방법을 선택할 수 있는 커서
- 전후이동 커서에서 **FETCH** 문을 위한 선택 사항
  - **NEXT**와 **PRIOR** : 현재 커서 위치에서 그 다음 튜플이나 이전 튜플
  - **FIRST**와 **LAST** : 첫번째나 마지막 튜플
  - **RELATIVE** <integer>
    - » 양수면 그 값만큼 전방(**forward**)으로 이동하고, 음수면 후방(**backward**)으로 이동
  - **ABSOLUTE** <integer>
    - » 양수면 릴레이션의 맨 앞에서부터 세어진 자리를 가리키며, 음수면 릴레이션의 맨 뒤로부터 거꾸로 세어진 자리

# 커서 (계속)

---

[예] 마지막 튜플부터 인출하여 튜플들의 리스트를 반대 순서로 얻을 수 있도록 예제를 변경.

```
Void changeWorth () {  
    . . .  
    EXEC SQL DECLARE execCursor SCROLL CURSOR FOR  
        SELECT netWorth FROM MovieExec;  
    EXEC SQL OPEN  execCursor;  
    EXEC SQL FETCH LAST FROM execCursor INTO :worth;  
    while(1) {  
        . . .  
        EXEC SQL FETCH PRIOR FROM execCursor INTO :worth;  
    }  
    EXEC SQL CLOSE execCursor;  
}
```

# 동적 SQL

---

- 동적(dynamic) SQL : SQL을 다른 언어에 내장시키는 일반적인 방법
  - SQL 문이 컴파일 시점에 정해지지 않을 때 사용
  - 호스트 언어 프로그램은 읽어들이는 문자열을 받아, 실행 가능한 SQL 문장으로 변경시키고 그 문장들을 실행하도록 SQL 시스템에게 지시

[예] 대화식(interactive) SQL 해석기

» SQL 문을 사용자로부터 입력받아 실행하는 프로그램

◆ **PREPARE : EXEC SQL PREPARE <sql 변수> FROM <문자열>**

- 문자열을 SQL 문으로 변환. SQL 문의 구문이 분석된 후 SQL 시스템에 의해 적절한 실행 방식이 결정된다.

◆ **EXECUTE : EXEC SQL EXECUTE <sql 변수>**

- 준비된 SQL 문을 실행

# 동적 SQL (계속)

## ◆ EXECUTE IMMEDIATE

**EXEC SQL EXECUTE IMMEDIATE** < 문자열 타입의 수식이나 호스트 변수 >

– **PREPARE**와 **EXECUTE**를 하나의 문장으로 결합

- ☞ 한 번만 실행되는 **SQL** 문을 위해서는 **EXECUTE IMMEDIATE**가 적합하다. 그러나 여러 번 반복되는 **SQL**문장은 **PREPARE**와 **EXECUTE**를 사용하는 것이 바람직하다.

```
void readquery() {  
    EXEC SQL BEGIN DECLARE SECTION;  
        char *query;  
    EXEC SQL END DECLARE SECTION;  
    /* 사용자로부터 질의의 입력을 기다린다.  
       malloc등을 사용하여 변수의 공간을 확보한다.  
       호스트 변수 :query가 질의의 첫번째 문자를  
       가리키도록 한다. */  
    EXEC SQL PREPARE SQLquery FROM :query;  
    EXEC SQL EXECUTE SQLquery;  
}
```

**EXEC SQL EXECUTE IMMEDIATE :query**



# Oracle에서 동적 SQL의 예

---

```
void executeQuery(char Selector) {
EXEC SQL BEGIN DECLARE SECTION;
    char query[100], title[50], studio[50];
    int    length;
EXEC SQL END DECLARE SECTION;

scanf("%d %s", &length, title);
sprintf(query, "SELECT studioName FROM Movie WHERE title = :v1 ");
if (Selector == 'A') strcat(query, " AND length > :v2 ");
EXEC SQL PREPARE execSQL FROM :query;
if (Selector == 'A')
    EXEC SQL EXECUTE execSQL INTO :studio USING :title, :length;
else {
    EXEC SQL DECLARE execCursor CURSOR FOR execSQL;
    EXEC SQL OPEN execCursor USING :title;
    while(1) {
        EXEC SQL WHENEVER NOT FOUND DO break;
        EXEC SQL FETCH execCursor INTO :studio;
        .....
    }
```



# PHP에서 Oracle 사용하기

---

```
<?php
if(!($conn = oci_connect("scott","tiger", "course")))
    print "<b>Connection Failed </b>\n";
$stmt = oci_parse($conn,"select title, year, length from movie");
oci_define_by_name($stmt,"TITLE",$title);
oci_define_by_name($stmt,"YEAR",$year);
oci_define_by_name($stmt,"LENGTH",$length);
oci_execute($stmt);
print "<TABLE bgcolor=#abbcab border=1>\n";
print "<TR align=left><TH> TITLE <TH> YEAR <TH> LENGTH</TR>\n";
while (oci_fetch($stmt)) {
    print "<TR> <TD> $title <TD> $year <TD> $length </TR>\n";
}
print "</TABLE>\n";
oci_free_statement($stmt);
oci_close($conn);
?>
```

# Java에서 Oracle 사용하기

---

```
import java.sql.*;

class GetMovie {
    public static void main (String args [])
        throws SQLException, ClassNotFoundException {

        Connection conn = DriverManager.getConnection (
            "jdbc:oracle:thin:@cs.ks.ac.kr:1521:course", "scott", "tiger");
        // Create a Statement
        Statement stmt = conn.createStatement ();

        ResultSet rset = stmt.executeQuery (
            "select title, year from movie where length > 100");

        while (rset.next ()) {
            System.out.print (rset.getString ("title") + " | ");
            System.out.println (rset.getInt ("year"));
        }
    }
}
```

# SQL에서의 트랜잭션

---

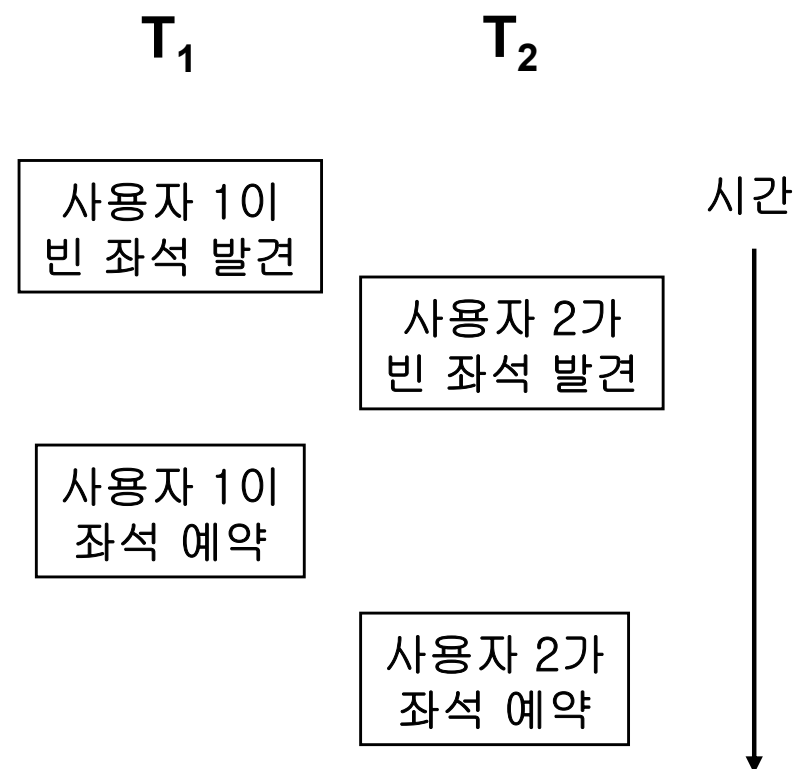
## ◆ 직렬화가능성(**Serializability**)

- 하나의 작업이 완전히 종료한 후에 다음 작업을 시작하는 처리 방식을 직렬실행(**serial schedule**)이라 부른다.
- 병행수행되어 실제로 직렬 실행은 아니지만 직렬 실행된 것과 같은 결과를 낼 때 그러한 실행을 직렬화가능한 실행 (**serializable schedule**)이라고 말한다.
  - » N개 트랜잭션들이 병행수행된 결과가 직렬실행을 가능한 N!개 결과중 하나와 일치하는 경우
  - » 데이터베이스의 병행수행 제어(**concurrency control**)에서 정확성의 중요한 기준
  - » 각 작업의 연산들이 **interleaving**되어 실행된다.

# SQL에서의 트랜잭션 (계속)

[예] 항공편과 가용한 좌석에 대한 릴레이션을 읽어서 특정 좌석이 가용한지 알아보고 가용하면 예약한다.

```
Void chooseSeat() {  
    EXEC SQL SELECT occupied INTO :occ  
        FROM Flights  
        WHERE fltNum = :flight  
            AND fltDate = :date  
            AND fltSeat = :seat;  
    if (!occ) {  
        EXEC SQL UPDATE Flights  
            SET occupied = 'B1';  
        WHERE fltNum = :flight  
            AND fltDate = :date  
            AND fltSeat = :seat;  
        /* 좌석을 배정하고 사용자에게 좌석이 배정되었다는  
        것을 알려주는 C와 SQL 코드 */  
    }  
    else  
        /* 가용하지 않다는 것을 알려주는 C와 SQL 코드 */  
}
```



2명의 사용자가 한 좌석에 예약 가능 :  
data inconsistency

# SQL에서의 트랜잭션 (계속)

---

## ◆ 직렬화가능성의 보장

- **Flights** 테이블을 읽고 있는 동안 다른 트랜잭션이 변경하지 못하도록 한다.
  - » **LOCK(잠금)** : 모든 데이터의 접근을 제어
    - ❶ 테이블을 읽기 전에 **Flights** 테이블에 **READ\_LOCK**을 건다.
    - ❷ 다른 트랜잭션이 **Flight** 테이블을 변경시키기 위해 **WRITE\_LOCK**을 걸려고 시도한다 : **READ\_LOCK**과 **WRITE\_LOCK**은 동시에 걸리지 못한다.
      - **WRITE\_LOCK**을 걸려고 시도한 트랜잭션은 **READ\_LOCK**이 해제될 때까지 기다려야 한다 : **Blocked**
  - » **READ\_LOCK (Shared Lock)**과 **WRITE\_LOCK(eXclusive Lock)**은 서로 호환되지 않는다. : **incompatible**
- 데이터베이스에 대한 모든 연산을 하기 전에 연관된 **LOCK**을 요청해서 성공해야 한다.

# SQL에서의 트랜잭션 (계속)

- ◆ 원자성(**atomicity**) : 모두 실행되거나 하나도 실행되지 않음
  - All(Commit) or Nothing(ROLLBACK, ABORT)

```
void transfer() {  
    EXEC SQL SELECT balance INTO :balance1  
        FROM Accounts  
        WHERE acctNo = :acct1;  
    if (balance1 >= amount) {  
        EXEC SQL UPDATE Accounts  
            SET balance = balance + :amount  
            WHERE acctNo = :acct2;  
        EXEC SQL UPDATE Accounts  
            SET balance = balance - :amount  
            WHERE acctNo = :acct1;  
        EXEC SQL COMMIT  
    }  
    else /* 이체를 위한 자금이 부족하다는 메시지를  
        출력하는 c 코드 */  
        EXEC SQL ROLLBACK  
}
```

[예] 두 계정 번호와 금액을  
입력으로 받아, 조건이 만족하면  
그 금액을 첫번째 계정에서 두  
번째 계정으로 이체

만약 이 부분에서 H/W나 S/W  
붕괴(**crash**)가 발생하면  
어떠한 일이 발생할까?

# SQL에서의 트랜잭션 (계속)

## ◆ Logging

시간

T\_35: Abort

System Failure

Disk Crashed (2020-06-10.12:13)

```
STARTUP DBMS
BOT_35
T_35:UPD ACCOUNTS 10000 10500
BOT_49
T_35:DEL CUSTOMERS 700523-1128332
...
T_49:INS ACCOUNTS
...
EOT35 - COMMIT;
...
BACKUP DB (2020-06-10.01:30)
...
SHUTDOWN DBMS
```

Log 기록



# SQL에서의 트랜잭션 (계속)

---

## ◆ 트랜잭션

– 원자적으로 수행되어야 하는 데이터베이스에 대한 연산들의 집합

☞ 트랜잭션은 데이터베이스나 스키마에 대한 질의나 조작을 하는 **SQL** 문장이 실행될 때 시작된다. 그러나 트랜잭션의 끝은 반드시 명시해 주어야 한다

### ■ **COMMIT : EXEC SQL COMMIT;**

» 트랜잭션이 성공적으로 끝나도록 한다.

» 일단 **COMMIT**이 되면 다른 트랜잭션에게 변경된 데이터가 제공될 수 있다.

» **COMMIT**된 데이터는 다른 트랜잭션이 영향을 주기 전까지는 보존된다 : **durability**

### ■ **ROLLBACK : EXEC SQL ROLLBACK;**

» 포기(**abort**), 즉 강제 종료시킨다. 트랜잭션 내의 **SQL** 문장에 의해 일어난 모든 변화는 원상복구(**undo**), 즉 복귀(**rolled back**)된다.

# SQL에서의 트랜잭션 (계속)

---

## ◆ 읽기 전용 트랜잭션(read-only transaction)

- 다른 트랜잭션과 병렬로 실행 가능

EXEC SQL SET TRANSACTION READ ONLY;

- » 트랜잭션이 시작되는 곳의 바로 앞에 위치시킨다.
- » SET TRANSACTION READ WRITE 도 있다.

## ◆ 손상가능 읽기(dirty read)

- 손상가능 데이터(dirty data)

- » 아직 완료(commit)되지 않은 트랜잭션에 의해 기록된 데이터

- 손상가능 읽기(dirty read)

- » 손상가능 데이터를 읽는 것

- » 이를 쓴 트랜잭션이 나중에 포기(rollback)될 수도 있다.

- 손상가능 읽기 방지 방법 : 트랜잭션 종료까지 잠금을 해제하지 않는다.

# Dirty Read의 예

---

## Transaction 1

**A = A + 10;**

...

;

## Transaction 2

...

**A = A - 20;**

...

**B = A + B;**

...

**COMMIT;**

~~ROLLBACK~~

## Database

**A = 10**

**B = 10**

# SQL에서의 트랜잭션 (계속)

[예] 계좌1에서 계좌2로 이체.

[단계 1] 돈을 계좌2에 더한다

[단계 2] 계좌1이 충분한 돈을 가지고 있는지 검사

(a) 충분한 돈이 없으면, 계좌2에서 그 돈을 제거하고 포기(rollback)

(b) 충분한 돈이 있으면, 계좌1에서 그 돈을 빼고 완료(commit)

■ 세 계좌 A1, A2, A3, 각각에 \$100, \$200, \$300이 있을때

T1: 계좌 A1에서 계좌 A2로 \$150을 이체하는 트랜잭션

T2: 계좌 A2에서 계좌 A3로 \$250을 이체하는 트랜잭션

• 가능한 시나리오

단계	T1	T2	A1	A2	A3	비고
1		A3 = A3+250	100	200	550	
1	A2 = A2+150		100	<b>350</b>	550	Dirty Data(A2)
2-(b)		A2의 잔액 충분	100	<b>350</b>	550	Dirty Read(A2)
2-(a)	A1의 잔액 부족		100	350	550	
2-(b)		A2=A2-250; <b>COMMIT</b>	100	100	550	
2-(a)	<b>ROLLBACK</b>		100	<b>-50</b>	550	A2=A2-150

■ Dirty Data를 읽은 T2도 함께 rollback되어야 한다. : 연쇄포기(Cascading Rollback)

# SQL에서의 트랜잭션 (계속)

---

[예] 좌석 선택 함수를 다음과 같이 약간 변경하자.

1. 가용한 임의의 좌석을 찾고 이를 예약한다.

가용한 좌석이 없으면, 포기한다.

2. 고객에게 그 좌석의 승인 여부를 물어 본다.

만일 고객이 승인하면, 완료한다.

그렇지 않으면, 그 좌석을 취소한 다음, 다른 좌석을 얻기 위해 단계 1을 반복한다.

❖ 좌석 **S**에 대해 : **T1** 예약 후 **T2** 예약 시도 후 포기, **T1** 고객에 의해 거부

» **T2**는 손상가능 데이터를 읽었음 : 심각한 문제는 없다 !!!

☞ 손상 가능 읽기는 경우에 따라 심각한 문제를 초래할 수도 있고 초래하지 않을 수도 있다.

☞ 손상가능 읽기의 용도 : 비교적 정확하지 않은 가장 최신 데이터 사용시

# SQL에서의 트랜잭션 (계속)

---

## ◆ SQL2에서의 네 가지 고립성 수준(isolation level)

### ■ Level 0 : 비완료된-읽기(read-uncommitted)

» **commit**되지 않은 데이터를 읽기 때문에 **Blocking** 기간이 짧다.

`SET TRANSACTION READ WRITE (or READ ONLY)`

`ISOLATION LEVEL READ UNCOMMITTED;`

» 손상가능 읽기를 허용

### ■ Level 1 : 완료된-읽기(read-committed)

» **commit**된 데이터만 읽기 때문에 상대적으로 응답시간이 길다.

`SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`

» 손상가능 데이터(즉, 비완료된 데이터)의 읽기를 금지

# SQL에서의 트랜잭션 (계속)

---

- **Level 2 : 반복가능-읽기(repeatable-read)**

`SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;`

- 한 트랜잭션 내에서 반복 수행된 같은 질의는 같은 결과를 생성
- 팬텀(phantom) 튜플문제를 제외하면 직렬화가능과 같다.

- **Level 3 : 직렬화가능(serializable)**

`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`

- **SQL2**의 디폴트 선택사항

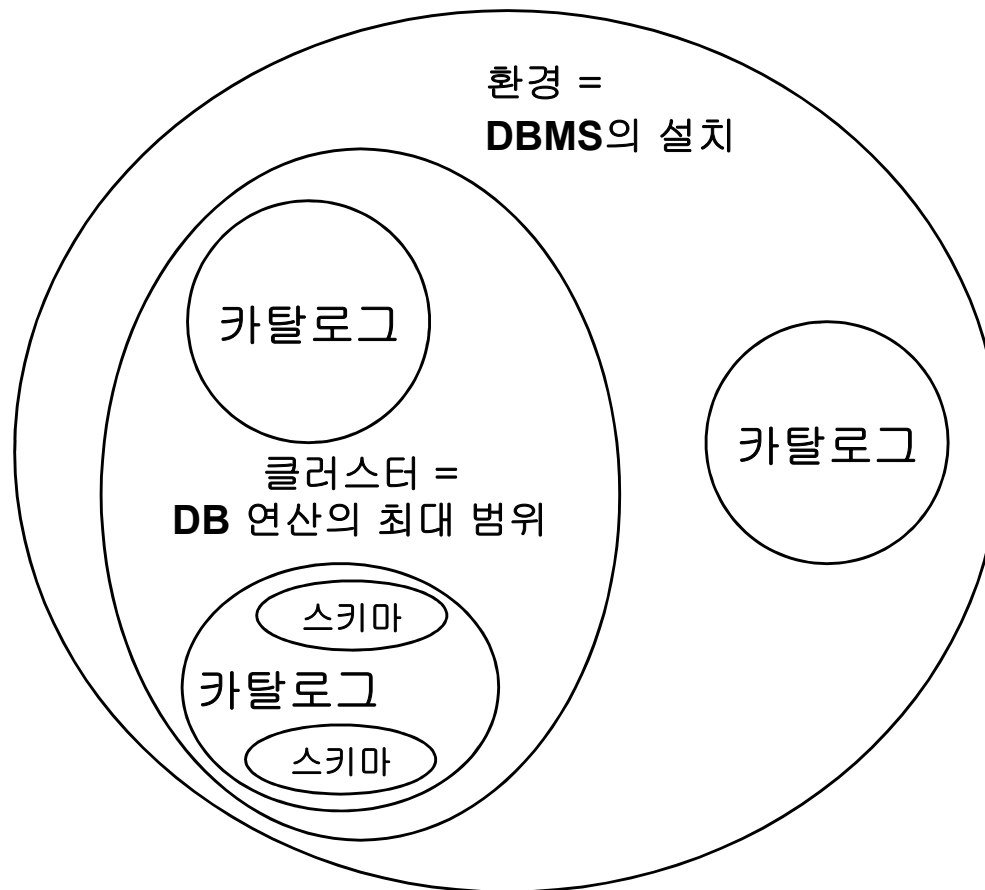
☞ **팬텀 튜플**

- 트랜잭션이 수행되는 도중에, 다른 트랜잭션에 의해 삽입되어 질의 결과에 포함되는 튜플
- 반복가능-읽기는 어떤 튜플이 처음에 검색되면, 그 질의가 반복될 때 그 튜플이 다시 검색된다는 것만을 보장한다.

# SQL 환경

## ◆ SQL 환경(environment)

- 데이터가 저장되고 SQL 연산이 실행될 수 있는 기반(framework)





# SQL 환경 (계속)

## ◆ 스키마 (릴레이션 스키마가 아닌 데이터베이스 스키마를 가리킴)

- 테이블, 뷰, 무결성 단정, 도메인 정보 등의 컬렉션
- 전체 시스템을 구성하는 기본 단위

**CREATE SCHEMA** <스키마 이름> <요소 선언들>

[예] 영화에 관한 다섯 개 릴레이션을 포함하는 스키마의 선언.

```
CREATE SCHEMA MovieSchema
CREATE DOMAIN CertDomain ...
/* 다른 도메인 선언들 */
CREATE TABLE MovieStar ...
/* 다른 네 개의 테이블 선언들 */
CREATE VIEW MovieProd ...
/* 다른 뷰 선언들 */
CREATE ASSERTION RichPres ...
```

테이블이 어떤 스키마에 속하는지 알려주어야 한다.

## ❖ SET SCHEMA : 현재 스키마(current schema)를 변경

**SET SCHEMA** <스키마 이름>

# SQL2의 스키마 요소들

---

- 기본 스키마 요소들
  - 테이블, 뷰, 도메인, 무결성 단정
- ◆ 문자 집합(**character set**)
  - 기호들의 집합과 이들을 부호화(**encoding**)하는 방법
- ◆ 문자 집합에 대한 대조 순서(**collation**)
  - 문자들 간에 “보다 작은(<)” 관계를 명시
- ◆ 번역(**translation**)
  - 한 문자 집합에 속하는 문자들을 다른 문자 집합의 문자들로 바꾸는 방법
- ◆ 부여 문(**GRANT statement**)
  - 스키마에 대한 접근 권한을 다룸
- ◆ 스키마 요소의 완전한 이름
  - **MovieCatalog.MovieSchema.Movie**

# SQL 환경 (계속)

---

## ◆ 카탈로그(catalog)

- 스키마들의 컬렉션
- 각 카탈로그는 *INFORMATION\_SCHEMA*를 포함

### ☞ *INFORMATION\_SCHEMA*

» 카탈로그 내의 모든 스키마들에 관한 정보

☞ **SQL2**에는 카탈로그를 생성하는 표준화된 방법이 정의되어 있지 않다.

## ■ **SET CATALOG:** 현재 카탈로그(current catalog)를 설정

**SET CATALOG** <카탈로그 이름>

## ◆ 클러스터

- 어떤 질의가 수행될 수 있는 최대 범위

# SQL 환경 (계속)

---

## ◆ Oracle에서 다른 데이터베이스 연결

- CREATE DATABASE LINK를 이용하여 연결

```
CREATE DATABASE LINK test_db
```

```
CONNECT TO guest IDENTIFIED BY guest
```

```
USING 'test_db';
```

- SELECT에서 사용하기

```
SELECT title, year
```

```
FROM Movie@test_db;
```

```
/* 원격 데이터베이스의 Movie 테이블 */
```

# SQL 환경 (계속)

---

## ◆ SQL 환경에서의 클라이언트와 서버

- 클라이언트 : 사용자가 서버에 연결할 수 있도록 해 줌
- 서버 : 데이터베이스 요소들에 대한 연산을 지원

## ◆ 연결(connection)

- 클라이언트와 서버를 연결

**CONNECT TO <서버이름> AS <연결이름>**

» 사용자가 여러 연결들을 열(**open**) 수 있지만 한 시점에는 단지 하나의 연결만 활성화(**active**)될 수 있다

- 활성화되는 연결을 바꿈

**SET CONNECTION <연결이름>**

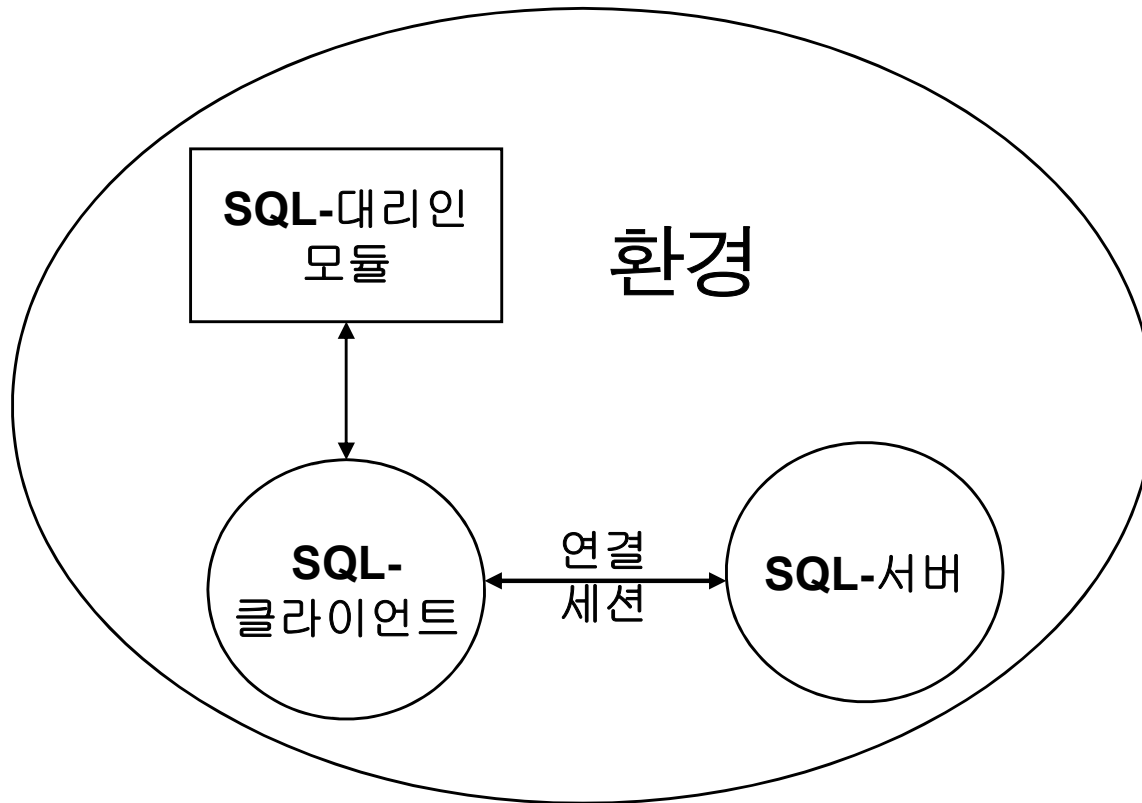
- 연결을 끊음

**DISCONNECT <연결이름>**

# SQL 환경 (계속)

## ◆ 세션(session)

- 세션은 이를 생성한 연결과 같은 상태로 존재



- 각 세션은 현재 카탈로그와 그 카탈로그 내의 현재 스키마를 가진다.
- 각 세션마다 권한을 가진(**authorized**) 사용자가 있다.

# SQL 환경 (계속)

## ◆ 모듈(module)

- 응용 프로그램을 가리키는 **SQL2** 용어
- 일반적(generic) **SQL** 인터페이스
  - 대화식 **SQL** 인터페이스
  - 각 질의나 문장은 그 자체가 모듈이다.

### ■ 내장된 **SQL**

- **ESQL** 문장들을 포함하는 호스트 프로그램
- **SQL** 문이 호스트 언어 프로그램 내에 나타난다

**EXEC SQL** <sql 문>

### ■ 진정한(true) 모듈

- 저장 함수(stored function)나 저장 프로시저(stored procedure)들의 콜렉션 (program code에 해당)

☞ **SQL 대리인(agent)** : 모듈의 실행 (process image에 해당)

**SQL** 시스템 구현 시에는 이들 모듈 중 하나 이상의 모듈을 제공하면 됨

# 저장 프로시저어/함수

---

## ◆ 저장 프로시저어/함수 (stored procedure/function)

- 사용자가 정의한 프로시저어/함수의 컴파일된 중간 코드가 **DB**에 저장
- 사용자는 저장 프로시저어/함수의 이름으로 호출하여 사용
- 책에서는 **PSM(Persistent Stored Modules)**로 언급

## ◆ 사용 예

```
CREATE OR REPLACE FUNCTION get_length
  (movietitle IN VARCHAR2,  movieyear IN SMALLINT)
RETURN INTEGER IS      return_length INTEGER;
BEGIN
  SELECT length into return_length  FROM system.movie
  WHERE lower(title) = lower(movietitle) AND movieyear = year;
  RETURN return_length;
EXCEPTION
  WHEN OTHERS THEN      RETURN  -1;
END get_length ;

-- SELECT get_length('Star Wars', 1977) FROM dual;
```



# SQL2에서 보안과 사용자 권한

---

- 권한(authorization) ID: 사용자 이름
  - 적절한 권리(privilege)가 부여(grant)된다.

## ◆ 권리

- **SELECT, INSERT, DELETE, UPDATE**

- » 해당 릴레이션에 대해 질의, 삽입, 삭제 그리고 갱신할 수 있는 권리

- **REFERENCES**

- » 무결성 제약에 있는 릴레이션을 참조할 수 있는 권리

- **USAGE**

- » 릴레이션과 무결성 단정을 제외한 스키마 요소들이나 도메인을 사용할 수 있는 권리

☞ 애틀리뷰트들의 일부만을 접근 권한의 대상으로 할 수 있다.

# SQL2에서 보안과 사용자 권한 (계속)

---

## ◆ 권리의 생성

- 소유권(**ownership**) : 소유자(**owner**)는 모든 권리들을 가진다.

## ■ 스키마 요소

- 스키마를 생성한 사용자가 소유자가 된다. 소유자는 그 스키마 요소들에 대한 모든 권리들을 가진다.

## ■ 연결

- 어떤 세션이 시작될 때 키워드 **USER**를 이용하여 사용자를 지정할 수 있으며, 이 사용자가 이 연결의 소유권을 가진다.
- **Login**할 사용자는 **SESSION**을 열 수 있는 권리를 가져야 한다.

**CONNECT TO <서버 이름> AS <연결 이름> USER <사용자 이름>**

## ■ 모듈

- 모듈이 생성될 때, 모듈의 소유자를 지정할 수 있다.

**AUTHORIZATION cs9999**

# SQL2에서 보안과 사용자 권한 (계속)

---

## ◆ SQL 문장을 실행하기 위해 필요한 권리

**INSERT INTO Studio(name)**

**SELECT DISTINCT StudioName**

**FROM Movie**

**WHERE StudioName NOT IN**

**(SELECT name**

**FROM Studio);**

- **Studio** 테이블의 애트리뷰트 **name**에 대한 **INSERT** 권리
- **Movie** 테이블의 애트리뷰트 **StudioName**에 대한 **SELECT** 권리
- **Studio** 테이블의 애트리뷰트 **name**에 대한 **SELECT** 권리

# SQL2에서 보안과 사용자 권한 (계속)

---

## ◆ 권리 검사 과정

- 각 스키마, 세션 그리고 모듈에는 사용자 즉, 권한 **ID**가 존재
  - 모든 **SQL** 연산에는 다음과 같은 두 당사자가 있다.
    - » 연산이 수행되는 데이터베이스 요소들
    - » 연산을 발생시킨 대리인(**agent**)
  - 대리인이 사용할 수 있는 권리는 현재 권한 **ID(current authorization ID)**로부터 유도
    - » 모듈 권한 **ID**
    - » 모듈 권한 **ID**가 없으면, 세션 권한 **ID**
- ☞ 현재 권한 **ID**가 어떤 **SQL** 연산을 수행하기 위해 필요한 모든 권리를 가지고 있을 때에만, 그 연산을 실행할 수 있다.

# SQL2에서 보안과 사용자 권한 (계속)

---

## ◆ Studio 테이블에 대한 **INSERT**가 가능한 경우들

- Studio 테이블의 소유자를 **janeway**라 하고 모듈에 **INSERT** 연산이 있다고 할 때 :
  - ❶ 모듈에 **AUTHORIZATION janeway** 절이 포함되어 있다면 : 사용자 **janeway**는 삽입 연산을 할 수 있다.
  - ❷ **AUTHORIZATION** 절이 모듈 내에 없다면 : 다른 사용자는 **janeway**로 **CONNECT TO**절을 이용하여 **SESSION**을 연다. (물론, **janeway**의 암호도 입력해야 함)
  - ❸ **janeway**가 다른 사용자 **sisko**에게 삽입에 관한 권리를 허용했고 모듈 내에 **AUTHORIZATION sisko** 절이 포함되어 있다면 : 사용자 **sisko**는 삽입 연산을 할 수 있다.
  - ❹ **AUTHORIZATION** 절이 모듈 내에 없고 ❸에서와 같이 삽입 권리를 **sisko**가 가지고 있다면 : **sisko**로 **CONNECT TO**절을 이용하여 **SESSION**을 열면 삽입 연산을 할 수 있다.

# SQL2에서 보안과 사용자 권한 (계속)

## ◆ 권리의 부여

- **GRANT**: 한 사용자가 어떤 권리를 다른 사용자에게 부여

**GRANT** <권리 리스트> **ON** <DB 요소> **TO** <사용자 리스트>  
**WITH GRANT OPTION** -----> option

[예] Movie와 Studio 테이블을 포함하는 MovieSchema가 Janeway에 의해 생성되었다고 할때, 권리 부여의 예.

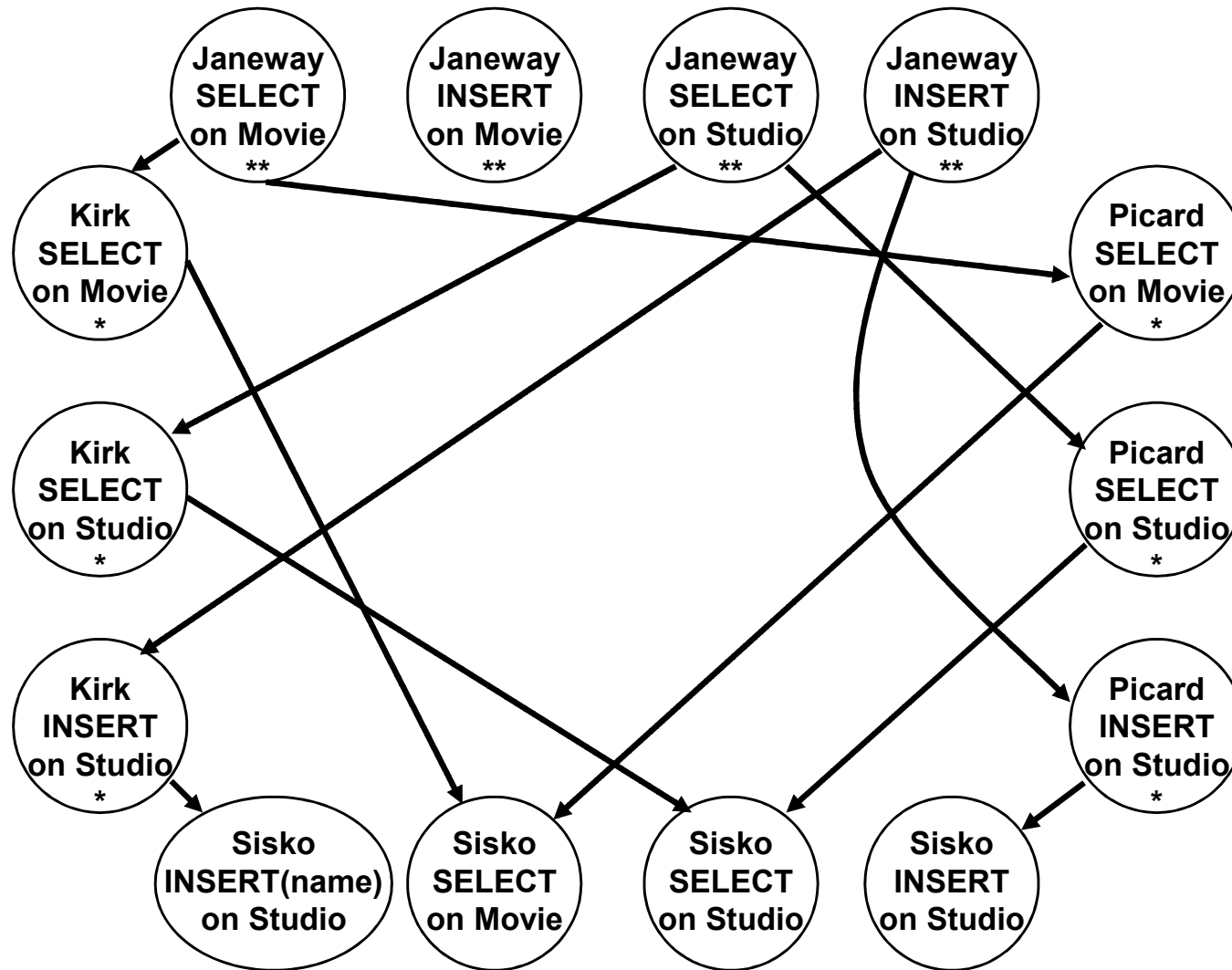
```
GRANT SELECT, INSERT ON Studio To kirk, picard
    WITH GRANT OPTION;
GRANT SELECT ON Movie TO kirk, picard
    WITH GRANT OPTION;
GRANT SELECT, INSERT ON Studio TO sisko;
GRANT SELECT ON MOVIE To sisko;
GRANT SELECT, INSERT(name) ON Studio TO sisko;
GRANT SELECT ON Movie TO sisko
```

janeway

picard

kirk

# SQL2에서 보안과 사용자 권한 (계속)



## ◆ 부여 다이어그램(**grant diagram**)

- 권리와 이들의 출처 (**origin**)를 기록
- **node**: 사용자와 권리
- **arc**: 부여자로부터 부여 받는 자로의 관계를 표시

\* **GRANT OPTION**

\*\* 소유권

# SQL2에서 보안과 사용자 권한 (계속)

---

## ◆ 권리의 해제(revoke)

**REVOKE** <권리 리스트> **ON** <데이터베이스 요소> **FROM** <사용자 리스트>

### ■ 연쇄 해제 : **CASCADE**

» 지정된 권리들이 해제되면, 해제된 권리들로부터 부여되었던 모든 권리들도 (같은 권리를 또 다른 사용자로부터 부여 받은 일이 없으면) 함께 해제된다.

### ■ 제한 해제 : **RESTRICT**

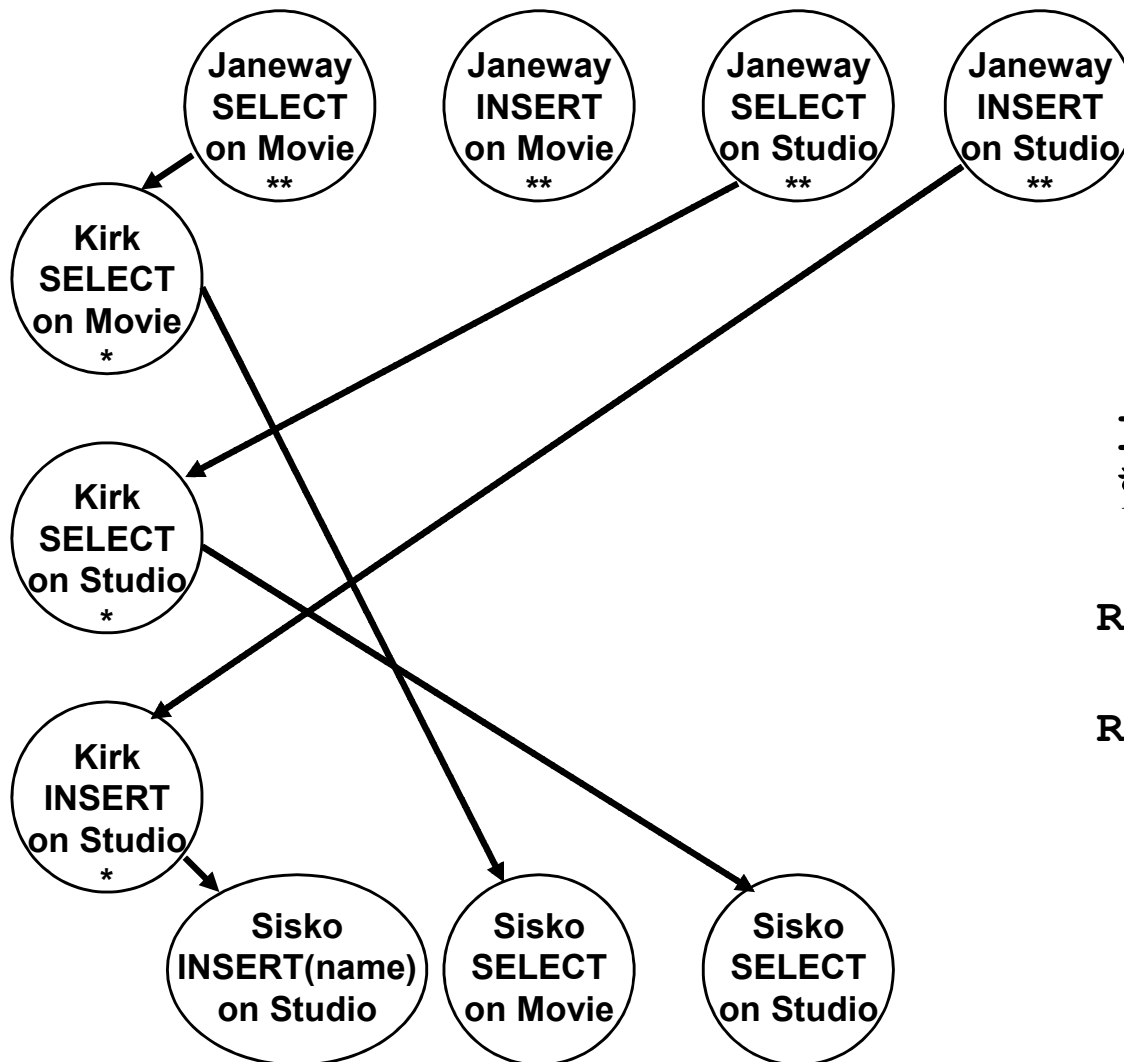
» 연쇄 해제 규칙에 의해 다른 사용자들의 권리가 해제되는 상황이 발생하면, 이 해제 문장이 실행되어서는 안된다.

### ■ **REVOKE GRANT OPTION FOR**

» 권리의 부여 선택권만 제거



# SQL2에서 보안과 사용자 권한 (계속)



janeway가 picard에게 부여  
했던 권리들을 해제.

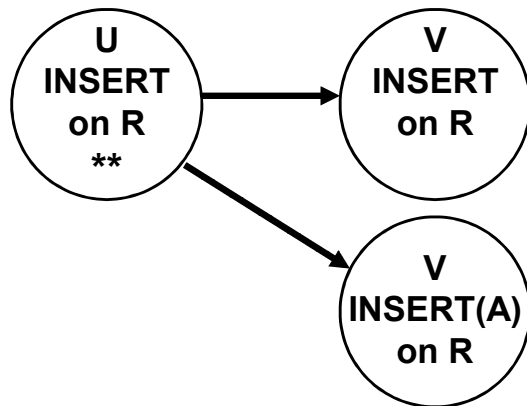
```
REVOKE SELECT, INSERT ON Studio  
FROM picard CASCADE;  
REVOKE SELECT ON Movie  
FROM picard CASCADE;
```

# SQL2에서 보안과 사용자 권한 (계속)

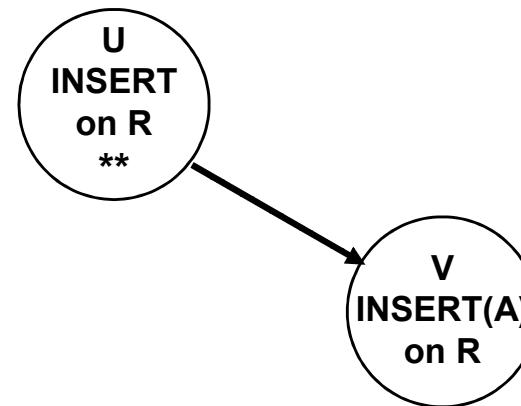
[예] 사용자 U가 릴레이션 R의 소유자라고 하자.

단계 주체 동작

1	U	GRANT	INSERT	ON	R	TO	V
2	U	GRANT	INSERT (A)	ON	R	TO	V
3	U	REVOKE	INSERT	ON	R	FROM	V RESTRICT



단계 2 수행 후



단계 3 수행 후

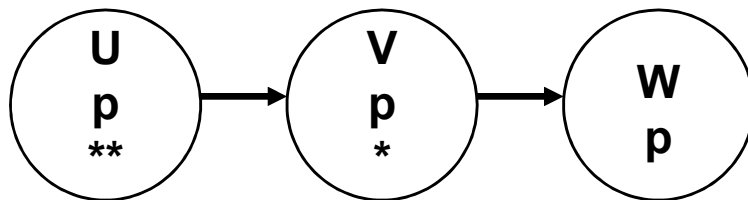
- INSERT와 INSERT(A)는 서로 다른 권한으로 취급된다

# SQL2에서 보안과 사용자 권한 (계속)

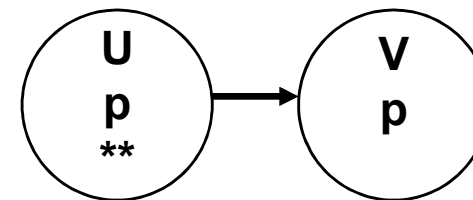
[예] 소유자 U 가 V에게 부여 선택권과 함께 권리 p를 부여하고,  
그 후 부여 선택권만 해제하는 예제.

단계 주체 동작

- 1 U GRANT p TO V WITH GRANT OPTION
- 2 V GRANT p TO W
- 3 U REVOKE GRANT OPTION FOR p FROM V CASCADE



단계 2 수행 후



단계 3 수행 후