Chapter 5: Advanced SQL

Revision by Gun-Woo Kim

Dept. of Computer Science and Engineering

Hanyang University

Database System Concepts, 6th Ed.

Chapter 5: Advanced SQL

Accessing SQL From a Programming Language

JDBC and ODBC

Functions and Procedural Constructs <- 여기까지 중간고사 범위

Triggers

Advanced Aggregation Features

JDBC and ODBC

개념을 묻는 질문이 나옴

API (application-program interface) for a program to interact with a database server

Application makes calls to

- Connect with the database server
- Send SQL commands to the database server
- Fetch tuples of result one-by-one into program variables

ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic

- Other API's such as ADO.NET sit on top of ODBC

JDBC (Java Database Connectivity) works with Java

Procedural Constructs in SQL

Procedural Extensions and Stored Procedures

SQL provides a **module** language

- Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.

Stored Procedures

- Can store procedures in the database
- then execute them using the **call** statement 오라클에서는 call대신 exec를 사용
- permit external applications to operate on the database without knowing about internal details

Functions and Procedures

SQL:1999 supports functions and procedures

Functions/procedures can be written in SQL itself, or in an external programming language.

Functions are particularly useful with specialized data types such as images and geometric objects.

- Example: functions to check if polygons overlap, or to compare images for similarity. 비전처리같은 딥러닝에 필요

Some database systems support **table-valued functions**, which can return a relation as a result.

SQL:1999 also supports a rich set of imperative constructs, including

Loops, if-then-else, assignment

Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

SQL Functions

Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
returns integer
begin
    declare d_count integer;
    select count (* ) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```

Find the department name and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 1
```

함수를 질의에서 사용 가능

Table Functions

SQL:2003 added functions that return a relation as a result

Example: Return all accounts owned by a given customer

```
create function instructors_of (dept_name char(20)
        returns table (  ID varchar(5),
                         name varchar(20),
                         dept_name varchar(20),
                         salary numeric(8,2))
return table
        (select ID, name, dept_name, salary
         from instructor
         where instructor.dept_name = instructors_of.dept_name)
```

Usage

```
        select *
        from table (instructors_of ('Music'))
```

SQL Procedures

The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),
                                  out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

리턴문이 없음

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

오라클에서는 call대신 exec

Procedures and functions can be invoked also from dynamic SQL

SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

Procedural Constructs

Warning: most database systems implement their own variant of the standard syntax below

read your system manual to see what works on your system

Compound statement: **begin ... end**,

May contain multiple SQL statements between **begin** and **end**.

Local variables can be declared within a compound statements

Whileand **repeat** statements :

```
declare n integer default 0;
while n < 10 do
   set n = n + 1
end while

repeat
   set n = n − 1
until n = 0
end repeat
```

Procedural Constructs (Cont.)

For loop

Permits iteration over all results of a query

Example:

```
declare n  integer default 0;
for r  as
      select budget from department
       where dept_name = 'Music'
 do
      set n = n - r.budget
 end for
```

Procedural Constructs (cont.)

이런구문을 적으라고는 안함

Conditional statements (**if-then-else**)
SQL:1999 also supports a **case** statement similar to C case statement

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      bonus := 50;
    END IF;

    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);
END p;
```

Procedural Constructs (cont.)

Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
…
 ..  signal out_of_classroom_seats
end
```

The handler here is **exit --** causes enclosing **begin..end** to be exited

Other actions possible on exception

여기까지가 시험범위

Triggers

Triggers

A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must:

- Specify the conditions under which the trigger is to be executed.
- Specify the actions to be taken when the trigger executes.

Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

- Syntax illustrated here may not work exactly on your database system; check the system manuals

Trigger Example

E.g. *time_slot_id* is not a primary key of *timeslot,* so we cannot create a foreign key constraint from *section* to *timeslot.*

Alternative: use triggers on *section* and *timeslot* to enforce integrity constraints

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
            select time_slot_id
            from time_slot)) /* time_slot_id not present in time_slot */
begin
    rollback
end;
```

Trigger Example Cont.

```
create trigger timeslot_check2 after delete on timeslot
    referencing old row as orow
    for each row     오라클에서는 ...ing old as ...임
    when (orow.time_slot_id not in (
                select time_slot_id
                from time_slot)
                /* last tuple for time slot id deleted from time slot */
            and orow.time_slot_id in (
                select time_slot_id
                from section)) /* and time_slot_id still referenced from section*/
    begin
       rollback
    end;
```

Triggering Events and Actions in SQL

Triggering event can be **insert**, **delete** or **update**

Triggers on update can be restricted to specific attributes

E.g., after update of *takes* **on** *grade*

Values of attributes before and after an update can be referenced

referencing old row as : for deletes and updates

referencing new row as : for inserts and updates

Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
 begin atomic
        set nrow.grade = null;
 end;
```

오라클에서는 atomic키워드가 존재안함으로 생략하고 써야함

nrow에대해서 begin키워드 안에서 뭔가 값을 설정하거나 가져올경우 변수명이나 값 앞에 ':'콜론을 붙여줘야함 예 가져올때 :nrow.grade; ,설정할때 nrow.grade = :null;

Trigger to Maintain credits_earned value

```
create trigger credits_earned after update of takes on
(grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred= tot_cred +
        (select credits
         from course
         where course.course_id= nrow.course_id)
    where student.id = nrow.id;
end;
```

Advanced Aggregation Features

Ranking

Ranking is done in conjunction with an order by specification.

Suppose we are given a relation
student_grades(ID, GPA)
giving the grade-point average of each student

Find the rank of each student.

select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s_rank*
from *student_grades*

An extra **order by** clause is needed to get them in sorted order

select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s_rank*
from *student_grades*
order by *s_rank* 오라클은 자동으로 랭킹순서대로 정렬해 출력함

Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3

dense_rank does not leave gaps, so next dense rank would be 2

Ranking

Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

```
select ID, (1 + (select count(*)
                 from student_grades B
                 where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

Ranking (Cont.)

Ranking can be done within partition of the data.

"Find the rank of students within each department."

```
select ID, dept_name,
        rank () over (partition by dept_name order by GPA desc)
                as dept_rank
 from dept_grades
 order by dept_name, dept_rank;
```

partition by => 거의 group by

Multiple **rank** clauses can occur in a single **select** clause.

Ranking is done *after* applying **group by** clause/aggregation

Can be used to find top-n results

More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

오라클에서는 limit 키워드가 지원안됨

Ranking (Cont.)

Other ranking functions:

- **percent_rank** (within partition, if partitioning is done) 백분위 소숫점으로 표현
- **cume_dist** (cumulative distribution) 0과 1사이의 값으로 출력
  - fraction of tuples with preceding values
- **row_number** (non-deterministic in presence of duplicates)

SQL:1999 permits the user to specify **nulls first** or **nulls last**

select *ID*,
rank ( ) **over** (**order by** *GPA* **desc nulls last**) **as** *s_rank*
from *student_grades*

default는 nulls last로 값의 맨 마지막에 null을 정렬해 순위매김

Ranking (Cont.)

For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.

E.g.,

select *ID*, **ntile**(4) **over** (**order by** *GPA* **desc**) **as** *quartile*
from *student_grades;*

Windowing

Used to smooth out random variations.

E.g., **moving average**: "Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day"

Window specification in SQL:

Given relation *sales(date, value)*

select *date,* ***sum***(*value*) **over**
(**order by** *date* **between rows** 1 **preceding and** 1 **following**)
from *sales*

Windowing

Examples of other window specifications:

unbounded preceding: 1번째 row

between rows unbounded preceding and current

rows unbounded preceding

unbounded following : 가장 마지막 row

range between 10 **preceding and current row**

- All rows with values between current row value –10 to current value

range interval 10 **day preceding**

- Not including current row

Windowing (Cont.)

Can do windowing within partitions

E.g., Given a relation *transaction* (*account_number, date_time, value*), where value is positive for a deposit and negative for a withdrawal

"Find total balance of each account after each transaction on the account"

```
select account_number, date_time,
    sum (value) over
            (partition by account_number
            order by date_time
            rows unbounded preceding)
   as balance
from transaction
order by account_number, date_time
```