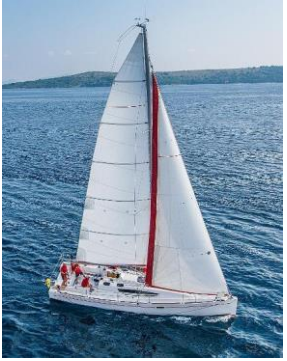

Chapter 5: Advanced SQL

Database System Concepts, 7th Ed.

Outline

- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features

Functions and Procedures

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Declaring 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
```

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

```
create function instructor_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 = instructor_of.dept_name)
```

- Usage

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

تابع اسکالر: این توابع تک خروجی هستند

```
CREATE  FUNCTION [ owner_name. ] function_name
   ( @parameter_name [AS] data_type )
RETURNS data_type
 AS
BEGIN

   function_body

   RETURN scalar_expression
END
```

تابعی که n را دریافت کرده و n*(n-1) را خروجی می دهد.

```
create function [dbo].[mul] (@n int)
returns bigint
AS
begin
 declare @r bigint
  set @r=(@n-1)*@n
 return @r
 end
```

```
select [dbo].[mul](10) as result
```

00 %

Results | Messages

	result
1	90

تابعی که شماره یک دانشجو راگرفته و نام ونام خانوادگی وی را نمایش می دهد .

```
Create FUNCTION fn_Nameret(@s# int)
 RETURNS NVARCHAR(40)
AS
BEGIN
        DECLARE @ret_Value  NVARCHAR(40);
         set @ret_Value = (SELECT  Name + ',' +Family
          FROM STD WHERE S# =@S#)

          RETURN (@ret_Value)
End
```

توابع table-valued: یک جدول خروجی می دهند.

```
CREATE  FUNCTION [ owner_name. ] function_name
    ( @parameter_name [AS] data_type )
RETURNS table
 AS
BEGIN
    function_body
    RETURN (select)
END
```

تفاوت اصلی این تعریف با توابع اسکالر در این است که نوع خروجی یک جدول تعریف شده است.

محدودیتی که روی این نوع تعریف وجود دارد این است که خروجی تابع باید توسط یک دستور select ایجاد شود.

In this example we will create an inline table-valued function that will retrieve records of all the students whose DOB is less than the DOB passed to the function.

```
CREATE FUNCTION BornBefore
 (
 @DOB AS DATETIME
 )
RETURNS TABLE
AS
BEGIN
        RETURN
        SELECT * FROM student
        WHERE DOB < @DOB
END
```

We will pass a date to "BornBefore" function. The student records retrieved by the function will have a DOB value lesser than that date.

```
SELECT
        name, gender, DOB
FROM
        dbo.BornBefore('1980-01-01')
ORDER BY
        DOB
```

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
```

- The keywords **in** and **out** are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- 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);
```

Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

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

Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

if *boolean expression*
then *statement or compound statement*
elseif *boolean expression*
then *statement or compound statement*
else *statement or compound statement*
end if

Procedure

- A stored procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs.

- Stored procedures can access or modify data in a database, but it is not tied to a specific database or object, which offers a number of advantages.

Benefits of using stored procedures

- A stored procedure provides an important layer of security between the user interface and the database. It supports security through data access controls because end users may enter or change data, but do not write procedures.

- A stored procedure preserves data integrity because information is entered in a consistent manner. It improves productivity because statements in a stored procedure only must be written once.

Benefits of using stored procedures (Cont.)

- Stored procedures offer advantages over embedding queries in a graphical user interface (GUI). Since stored procedures are modular, it is easier to troubleshoot when a problem arises in an application.

- Stored procedures are also tunable, which eliminates the need to modify the GUI source code to improve its performance. It's easier to code stored procedures than to build a query through a GUI.

نواشتن پراسیجر

```
CREATE  PROCEDURE [ owner_name. ] procedure_name
   ( @parameter_name [AS] data_type )
 AS
BEGIN

   procedure_body

END
```

مثال ۱

```
1. CREATE PROCEDURE stpInsertMember
2. @MemberName varchar(50),
3. @MemberCity varchar(25),
4. @MemberPhone varchar(15)
5. AS
6. BEGIN
7.    Insert into tblMembers (MemberName,MemberCity,MemberPhone)
                  Values (@MemberName,@MemberCity, @MemberPhone)

8.   END
```

مثال ۲

```
1. CREATE PROCEDURE stpUpdateMemberByID
2. @MemberID int,
3. @MemberName varchar(50),
4. @MemberCity varchar(25),
5. @MemberPhone varchar(15)
6.
7. AS
8. BEGIN
9.     UPDATE tblMembers
10.     Set MemberName = @MemberName,
11.         MemberCity = @MemberCity,
12.         MemberPhone = @MemberPhone
13.     Where MemberID = @MemberID
14. END
```

فراخوانی

```
1.  CREATE PROCEDURE stpUpdateMemberByID
2.  @MemberID int,
3.  @MemberName varchar(50),
4.  @MemberCity varchar(25),
5.  @MemberPhone varchar(15)
6.  AS
7.  BEGIN
8.      UPDATE tblMembers
9.      Set MemberName = @MemberName,
10.         MemberCity = @MemberCity,
11.         MemberPhone = @MemberPhone
12.     Where MemberID = @MemberID
13. END
```

```
EXEC  stpUpdateMemberByID @MemberID = 10,
 @MemberName = 'Ali' , @MemberCity = ‘Tehran‘,
@MemberPhone =‘0912912912‘;
```

سوال

- تابعی بنویسید که یک رشته به طول حداکثر ۲۰ و یک الگو به طول حداکثر ۳ دریافت نماید. این تابع باید تمام تکرارهای الگو را از رشته بزرگتر حذف کرده و رشته باقی مانده را در برگرداند. اگر با حذف یکباره دوباره رشته شامل آن الگو بود باید مجدد حذف شود.

- تابعی را در SQL SERVER بنویسید که مشابه تابع LPAD در اوراکل کار کند.

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

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **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. For example, 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;
```

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;

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Iteration Number	*Tuples in c1*
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

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*
- 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 (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;
```

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

Ranking (Cont.)

- It is often the case, especially for large results, that we may be interested only in the top-ranking tuples of the result rather than the entire list.
- For rank queries, this can be done by nesting the ranking query within a containing query whose **where** clause chooses only those tuples whose rank is lower than some specified value.
- For example, to find the top 5 ranking students based on GPA we could extend our earlier example by writing:

```
select *
from (select ID, rank() over (order by (GPA) desc) as s rank
from student grades)
where s rank <= 5;
```

Ranking (Cont.)

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
    - If there are n tuples in the partition and the rank of the tuple is r, then its percent rank is defined as $(r - 1)/(n - 1)$ (and as *null* if there is only one tuple in the partition).
  - **row_number:** sorts the rows and gives each row a unique number corresponding to its position in the sort order; different rows with the same ordering value would get different row numbers, in a nondeterministic fashion.

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:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **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
```