Chapter 5: Advanced SQL

Instructor : Fatemeh Mansoori

Database System Concepts, 7th Ed.

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.

Define function in Postgresql

```
CREATE [OR REPLACE] FUNCTION functionName (someParameter 'parameterType')
RETURNS 'DATATYPE'
AS $_block_name_$
DECLARE
    --declare something
BEGIN
    --do something
    --return something
END;
$_block_name_$
LANGUAGE plpgsql;
```

Functions

Create a function that get the department name and returns the number of instructors in that department

```
create or replace function instructor_count(dept_name varchar(20))
returns integer
as
$$
declare
icount integer;
begin
select count(*) into icount
from instructor as i
where i.dept_name = instructor_count.dept_name;
return icount;
end;
$$
language plpgsql;
```

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 or replace function instructor_of(dn varchar(20))
returns table(id varchar(5), name varchar(20), dept_name varchar(20), salary numeric(8,2)) as
$$
begin
return query
(select * from instructor as i
 where i.dept_name = dn
);
end;
$$
language plpgsql;
```

- Usage

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

SQL Procedures

- The *dept_count* function could instead be written as procedure:
- In Postgresql :

```
create or replace procedure dept_count(in dn varchar(20), out icount integer)
as
$$
begin
    select count(*) into icount
    from instructor
    where dept_name=dn;
end;
$$
language plpgsql;
```

- 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.

  call *dept_count_proc*('Physics', *0*);

Example procedure

- Registers student after ensuring classroom capacity is not exceeded

```
create or replace procedure registeration(id student.id%type,
                                          course_id course.course_id%type,
                                         sec_id section.sec_id%type,
                                         semester section.semester%type,
                                         year section.year%type) as
$$
declare r_count int; lmt int;
begin
    select count(*) into r_count
    from takes t
    where t.course_id = registeration.course_id and t.sec_id = registeration.sec_id and
    t.semester = registeration.semester and t.year = registeration.year;
    select capacity into lmt
    from section s natural join classroom c
    where s.course_id = registeration.course_id and s.sec_id = registeration.sec_id and
    s.semester = registeration.semester and s.year = registeration.year;
    if r_count < lmt then
        insert into takes(id, course_id, sec_id, semester, year)
        values (registeration.id, registeration.course_id, registeration.sec_id,
               registeration.semester, registeration.year);
    else
        raise 'Capacity is full';
    end if;
end;
$$
language plpgsql;
```

Exception conditions

- Signaling of exception conditions, and declaring handlers for exceptions

  declare *out_of_classroom_seats* **condition**
  declare exit handler for *out_of_classroom_seats*
  begin
  …
  end

- The statements between the **begin** and the **end** can raise an exception by executing "**signal** *out_of_classroom_seats*"
- The handler says that if the condition arises he action to be taken is to exit the enclosing the **begin** **end** statement.

SQL Procedures (Cont.)

- Procedures and functions can be invoked also from dynamic SQL
- SQL allows more than one procedure of the so long as the number of arguments of the procedures with the same name is different.
- The name, along with the number of arguments, is used to identify the procedure.

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Postgresql language constructs :
- Conditional statements (**if-then-else**)

  if *boolean expression* **then**

  statement (s)

  elseif *boolean expression* **then**

  statement(s)

  else

  statement(s)

  end if

Language Constructs (Cont.)

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

Language Constructs (Cont.)

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

Language Constructs (Cont.)

- Loops:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;
```

```
LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;
```

Language Constructs (Cont.)

```
<<ablock>>
BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT ablock;  -- causes exit from the BEGIN block
    END IF;
    -- computations here will be skipped when stocks > 100000
END;
```

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

Language Constructs (Cont.)

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT done LOOP
    -- some computations here
END LOOP;
```

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

Language Constructs (Cont.)

```
DO $$
DECLARE
city_names varchar;
BEGIN
FOR city_names IN SELECT city_name FROM major_cities
        LOOP
        RAISE NOTICE '%', city_names;
        END LOOP;
END$$;
```

```
DO $$
DECLARE
emp_name record;
BEGIN
FOR emp_name IN SELECT first_name, last_name FROM employee LIMIT 10
LOOP
RAISE NOTICE '% %', emp_name.first_name,emp_name.last_name;
END LOOP;
END$$;
```

Language Constructs (Cont.)

```
FOR r IN SELECT * FROM foo
    WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
```

Triggers

Triggers

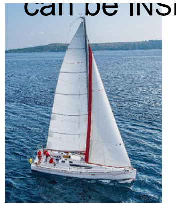
- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- 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;
```

can be INSERT , DELETE, UPDATE or TRUNCATE



Trigger in postgresql

```
CREATE TRIGGER trigger_name
   {BEFORE | AFTER} { event }
   ON table_name
   [FOR [EACH] { ROW | STATEMENT }]
       EXECUTE PROCEDURE trigger_function
```

- The event can be Insert, delete, update or truncate
- A row-level trigger is fired for each row while a statement-level trigger is fired for each transaction
- Suppose a table has 100 rows and two triggers that will be fired when a DELETE event occurs.
- If the DELETE statement deletes 100 rows, the row-level trigger will fire 100 times, once for each deleted row. On the other hand, a statement-level trigger will be fired for one time regardless of how many rows are deleted.

trigger function syntax

```
CREATE FUNCTION trigger_function()
   RETURNS TRIGGER
   LANGUAGE PLPGSQL
AS $$
BEGIN
   -- trigger logic
END;
$$
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication

When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

- Disable a trigger:
- when you disable a trigger, the trigger still exists in the database. However, the disabled trigger will not fire when an event associated with the trigger occurs

```
ALTER TABLE table_name
DISABLE TRIGGER trigger_name | ALL
```

- Enable a trigger:

```
ALTER TABLE table_name
ENABLE TRIGGER trigger_name |  ALL;
```

- Delete a trigger :

```
DROP TRIGGER [IF EXISTS] trigger_name
ON table_name [ CASCADE | RESTRICT ];
```

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

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.

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

- 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;
```

- 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

Ranking (Cont.)

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
  - **cume_dist** (cumulative distribution)
    - 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*

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*

Postgres sintax:

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
```

Windowing (Cont.)

- Postgresql syntax

```
SELECT
        product_name,
        group_name,
        price,
        LAST_VALUE (price) OVER (
                PARTITION BY group_name
                ORDER BY
                        price RANGE BETWEEN UNBOUNDED PRECEDING
                AND UNBOUNDED FOLLOWING
        ) AS highest_price_per_group
FROM
        products
```


End of Chapter 5