DTS207TC Database Development and Design

Lecture 3
Chap 5 Advanced SQL

Di Zhang, Autumn 2025

*Page titles with * will not be assessed*

Outline

- Accessing SQL From a Programming Language
- Functions and Procedures
- Triggers
- Recursive Queries

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

1

The content of this section of the textbook is outdated. JDBC/ODBC is not commonly used in Python. Python usually uses its language-specific protocol DB-API.

*Other interfaces of DB

- DB without SQL
  - HTTP: postgres-meta
  - Structured text: GraphQL
  - NLP: sequel.sh

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 Python.

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
```

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

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'))
```

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);
```

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.
- Compound statement: **begin** … **end,**
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- While and repeat statements:
  - **while** boolean expression **do**
    sequence of statements ;
    end while
  - **repeat**
    sequence of statements ;
    until boolean expression
    end repeat

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

Example procedure

Xi'an Jiaotong-Liverpool University 西交利物浦大學

- Registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
- Signaling of exception conditions, and declaring handlers for exceptions

3

```
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.

External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
  - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL\can be executed by these functions.
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out count integer)
language C
external name   '/usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power.
- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space.
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance.
  - Direct execution in the database system's space is used when efficiency is more important than security.

*Other interfaces of DB

- SQL without DB
  - SQL on Pandas
  - SQL for data flows: Flink SQL
  - SQL for ML: https://sql-machine-learning.github.io/

4

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;

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
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

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

*Event-driven Programming

- In computer programming, event-driven programming also known as event-based programming is a programming method in which the flow of the program is determined by sensor outputs or user actions (such as mouse clicks, key presses) or by messages from other programs or threads running on the computer.

*Trigger is an instance of event-driven

- Database trigger is nothing more than “an event-driven callback living inside the database kernel”.

Event-driven architecture (EDA)	Database trigger
Event source	DML/DDL/database-level events on a table or column (INSERT UPDATE, DELETE, CREATE, LOGON …)
Event object	The “transition table” or :OLD / :NEW pseudo-records generated by the kernel for that change
Event bus / dispatcher	The trigger manager, which hands control to the bound trigger when a given event point fires
Listener / handler	The PL/SQL, T-SQL, pgSQL, etc. procedural code you write inside the trigger body
Sync vs async	99 % of triggers run **synchronously** inside the same transaction, but you can also write the event into a queue and let an asynchronous job consume it (Oracle AQ, pgmq, RabbitMQ plug-in …), turning the DB into an “EDA inside the box”
Callback rules	Trigger timing (BEFORE / AFTER / INSTEAD OF), granularity (ROW / STATEMENT), number of invocations—exactly the filtering conditions you specify when registering an event handler

**Why the instruction-driven is always the mainstream?

- From the machine's perspective, they are the equivalent. But,...
- The human brain has evolved through natural selection and contains specialized cognitive modules that address specific survival and reproductive challenges.
- For instance,people typically complete tasks step by step in a certain order.This sequential thinking stems from:
  - In hunting activities,focusing on one task and proceeding according to a plan is more conducive to survival.
  - Sequential knowledge is repeatable, thus it can be better learned and passed down.
  - Humans tend to choose predictable environments and settle in calm areas rather than those with intense changes.

Recursion in SQL

- 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

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.
- Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

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

7

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

8

- Fixed-Point: No change with iterations

*Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

- A recursive algorithm takes one step toward solution and then recursively call itself to further move. The algorithm stops once we reach the solution.
- Since called function may further call itself, this process might continue forever. So it is essential to provide a base case to terminate this recursion process.

*Example: Factorial Numbers

FACTORIAL

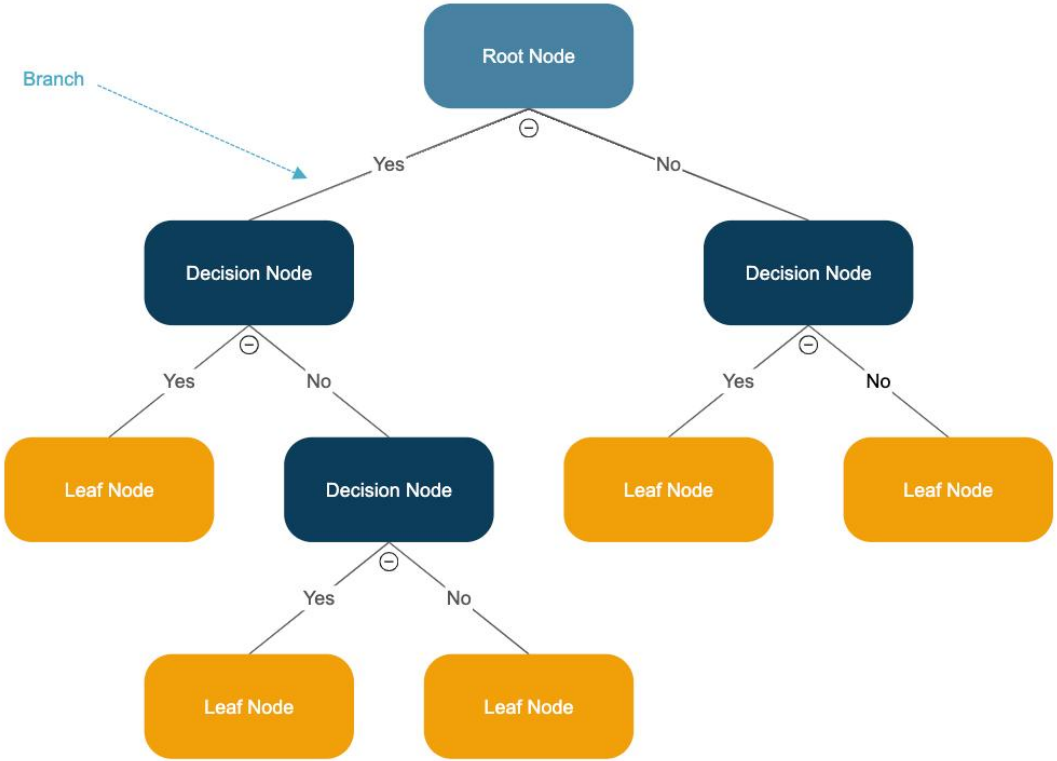
n!

The factorial of a non-negative integer, n, is the product of all positive integers less than or equal to n.

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5,040
8! = 40,320
9! = 362,880
10! = 3,628,800
11! = 39,916,800
12! = 479,001,600
13! = 6,227,020,800
14! = 87,178,291,200
15! = 1,307,674,368,000

9

*Example: Decision Tree



10