CS2102: Database Systems

Lecture 8 — Programming with SQL

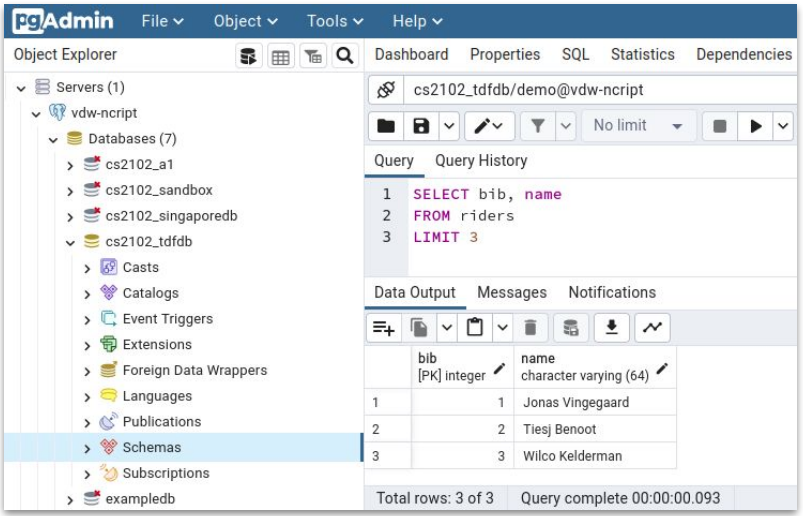
Overview

- **Writing Database Applications**
  - **Motivation**
  - Statement Level Interface
  - Call Level Interface
  - SQL Injection Attacks
- SQL Functions and Procedures
  - Motivation & Overview
  - Main Parameters: arguments, language, return values
  - Assignments & Control structures
  - Cursors
- Summary

Using SQL So Far

- Interactive SQL: directly writing SQL statements to an interface
  - Command line interface
    e.g., PostgreSQL's **psql** [1]
  - Graphical user interface
    e.g., PostgreSQL's **pgAdmin** [2]

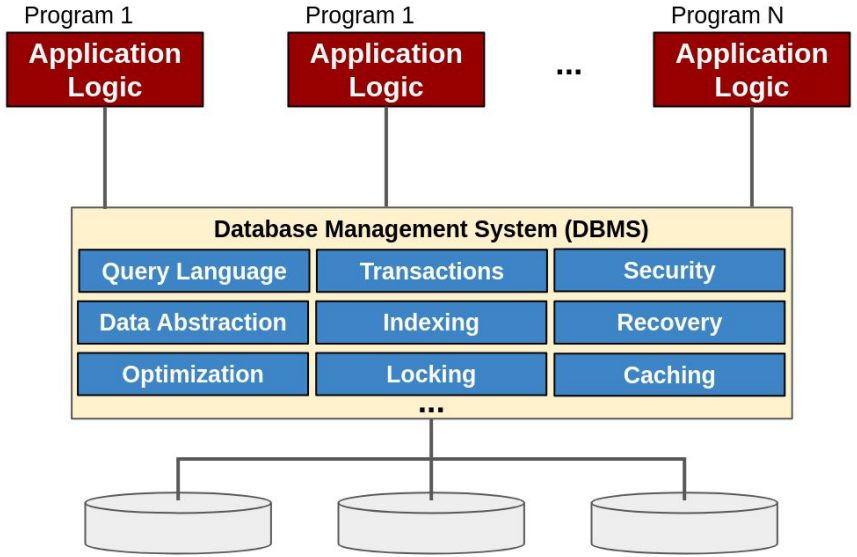
```
cs2102_tdfdb=# SELECT bib, name FROM riders LIMIT 3;
 bib |       name
-----+-----------------
   1 | Jonas Vingegaard
   2 | Tiesj Benoot
   3 | Wilco Kelderman
(3 rows)
```



[1] https://www.postgresql.org/docs/current/static/app-psql.html
[2] https://www.pgadmin.org/

What We Want: Writing Applications

- Non-interactive SQL
  - SQL statements are included in an application written in a host language
  - All operations can be executed
    (INSERT, UPDATE, DELETE, SELECT, etc.)
- 2 main alternatives
  - Statement Level Interface (SLI)
  - Call Level Interface (CLI)



Overview

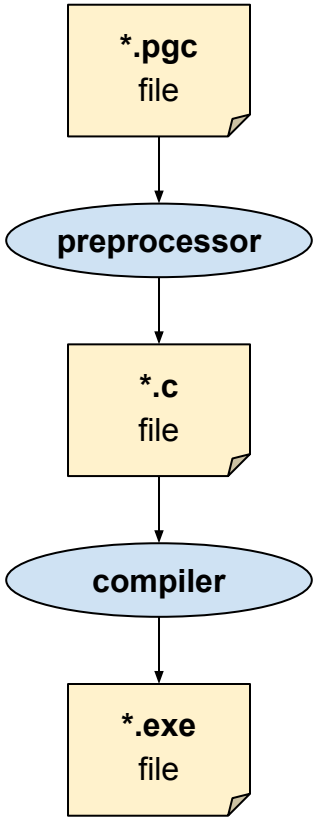
- **Writing Database Applications**
  - Motivation
  - **Statement Level Interface**
  - Call Level Interface
  - SQL Injection Attacks
- SQL Functions and Procedures
  - Motivation & Overview
  - Main Parameters: arguments, language, return values
  - Assignments & Control structures
  - Cursors
- Summary

Statement Level Interface (SLI)

- Statement Level Interface (SLI)
  - Code is a mix of host language statements and SQL statements
  - Examples: Embedded SQL, Dynamic SQL

- SLI — basic process (here using C)

1) **Write** code that mixes host language with SQL
   (the normal C compiler will not understand this code!)
2) **Preprocess** code using a preprocessor
   (understand SQL statements and converts them to pure C)
3) **Compile** code into an executable program



SLI — Common Steps

Declaration

- Declare variables
- Can be used by host language & SQL
  (other variables only usable by host language)

Connection

- Connect to database with credentials

Execution

- Prepare queries (in case of dynamic SQL)
- Execute queries (might require cursors)
- Operate on result

Deallocation

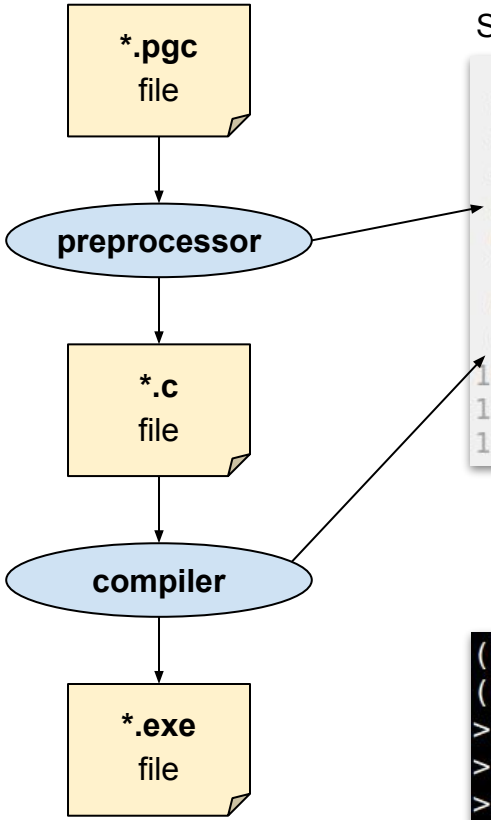
- Release all resources
- If needed: commit any changes to db

Source file: **staticquery.pgc**

```c
int main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char v_bib[32], v_name[32];
    const char *target = "cs2102_tdfdb@localhost";
    const char *user = "demo";
    const char *passwd = "demo";
    EXEC SQL END DECLARE SECTION;

    // Connect to database
    EXEC SQL CONNECT TO :target USER :user USING :passwd;

    // Declare cursor
    EXEC SQL DECLARE cursor CURSOR FOR
    SELECT bib, name FROM riders LIMIT 3;

    // Open cursor
    EXEC SQL OPEN cursor;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    // Loop through cursor and display results
    for(;;) {
        EXEC SQL FETCH NEXT FROM cursor INTO :v_bib, :v_name;
        printf(">>> bib: %s, name: %s\n", v_bib, v_name);
    }

    // Cleanup (close cursor, commit, disconnect)
    EXEC SQL CLOSE c;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;

    return 0;
}
```

Static SQL = fixed query

SLI — Preprocessing, Compiling, Running Code



Script file: **compile-static-query**

```
1  #!/bin/bash
2
3  # Run ecpg preprocessor to convert C program with embedded SQL statements
4  # to normal C code; replaces the SQL invocations with special function calls.
5  ecpg staticquery.pgc
6
7  # Compile generated C code; requires to include all header files the compiler
8  # needs to understand the special function calls (files come with PostgreSQL).
9  gcc -g -I/usr/include/postgresql -c staticquery.c
10
11 # Build output to executable file; also needs access to the header files.
12 gcc -o staticquery staticquery.o -L/usr/include/postgresql -lecpg
```

```
(base) vdw@vdw-ncript:~/share/dev/cpp/sli-demo$ ./compile-static-query
(base) vdw@vdw-ncript:~/share/dev/cpp/sli-demo$ ./staticquery
>>> bib: 1, name: Jonas Vingegaard
>>> bib: 2, name: Tiesj Benoot
>>> bib: 3, name: Wilco Kelderman
```

SLI — Dynamic SQL

- Dynamic SQL
  - SQL query is generated at runtime
  - Example on the right: number of riders are specified as command line parameter

```
(base) vdw@vdw-ncript:~/share/dev/cpp/sli-demo$ ./compile-dynamic-query
(base) vdw@vdw-ncript:~/share/dev/cpp/sli-demo$ ./dynamicquery 15
>>> bib: 1, name: Jonas Vingegaard
>>> bib: 2, name: Tiesj Benoot
>>> bib: 3, name: Wilco Kelderman
>>> bib: 4, name: Sepp Kuss
>>> bib: 5, name: Christophe Laporte
>>> bib: 6, name: Wout Van Aert
>>> bib: 7, name: Dylan Van Baarle
>>> bib: 8, name: Nathan Van Hooydonck
>>> bib: 11, name: Tadej Pogačar
>>> bib: 12, name: Mikkel Bjerg
>>> bib: 14, name: Felix Grossschartner
>>> bib: 15, name: Vegard Stake Laengen
>>> bib: 16, name: Rafal Majka
>>> bib: 17, name: Marc Soler
>>> bib: 18, name: Matteo Trentin
```

```c
int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
    char v_bib[32], v_name[32];
    char *sql = "SELECT bib, name FROM riders LIMIT ?";
    const char *target = "cs2102_tdfdb@localhost";
    const char *user = "demo";
    const char *passwd = "demo";
    const char *limit = argv[1];
    EXEC SQL END DECLARE SECTION;

    // Connect to database
    EXEC SQL CONNECT TO :target USER :user USING :passwd;

    // Prepare query (create query from string)
    EXEC SQL PREPARE query FROM :sql;

    // Declare cursor
    EXEC SQL DECLARE cursor CURSOR FOR query;

    // Open cursor using the LIMIT value in the query
    EXEC SQL OPEN cursor USING :limit;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    // Loop through cursor and display results
    for(;;) {
        EXEC SQL FETCH NEXT FROM cursor INTO :v_bib, :v_name;
        printf(">>> bib: %s, name: %s\n", v_bib, v_name);
    }

    // Prepared statement no longer needed; deallocate it
    EXEC SQL DEALLOCATE PREPARE q;

    // Cleanup (close cursor, commit, disconnect)
    EXEC SQL CLOSE c;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;

    return 0;
}
```

Overview

- **Writing Database Applications**
  - Motivation
  - Statement Level Interface
  - **Call Level Interface**
  - SQL Injection Attacks
- SQL Functions and Procedures
  - Motivation & Overview
  - Main Parameters: arguments, language, return values
  - Assignments & Control structures
  - Cursors
- Summary

Call Level Interface (CLI)

- Call Level Interface (CLI)
  - Application is completely written in host language ➜ no preprocessor needed
  - SQL statements are strings passed as arguments to host language procedures or libraries
  - Examples: ODBC (Open DataBase Connectivity), JDBC (Java DataBase Connectivity)

- Examples: libraries for Python
  - pyodbc: connects to any DBMS with ODBC support
  - psycopg: connects to PostgreSQL
  - cx_Oracle: connects to Oracle
  - MYSQLdb: connects to MySQL

CLI — Static SQL Example

Declaration

Connection

Execution

Deallocation

```python
import psycopg # Host language library (here psycopg for Python)

# Connect to database
connection = psycopg.connect("host=localhost dbname=cs2102_tdfdb user=demo password=demo")

# Create cursor
cursor = connection.cursor()

# Open cursor by executing query (string parameter passed to execute() method)
cursor.execute("SELECT bib, name FROM riders LIMIT 3")

# Loop over all results until no next tuple is returned
while True:
    row = cursor.fetchone()
    if row is None:
        break
    print(f">>> bib: {row[0]}, name: {row[1]}")

# Cleanup
cursor.close()
connection.commit()
connection.close()
```

```
>>> bib: 1, name: Jonas Vingegaard
>>> bib: 2, name: Tiesj Benoot
>>> bib: 3, name: Wilco Kelderman
```

CLI — Dynamic SQL Example

Declaration

Connection

Execution

Deallocation

```python
import psycopg # Host language library (here psycopg for Python)

# Set a user-defined value (here: maximum number of riders returned)
limit = 5

# Connect to database
connection = psycopg.connect("host=localhost dbname=cs2102_tdfdb user=demo password=demo")

# Create cursor
cursor = connection.cursor()

# Open cursor by executing query (string parameter passed to execute() method)
cursor.execute("SELECT * FROM riders LIMIT %s", (limit,))

# Loop over all results until no next tuple is returned
while True:
    row = cursor.fetchone()
    if row is None:
        break
    print(f">>> bib: {row[0]}, name: {row[1]}")

# Cleanup
cursor.close()
connection.commit()
connection.close()
```

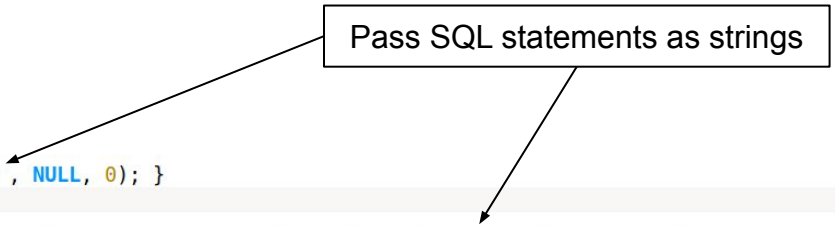
```
>>> bib: 1, name: Jonas Vingegaard
>>> bib: 2, name: Tiesj Benoot
>>> bib: 3, name: Wilco Kelderman
>>> bib: 4, name: Sepp Kuss
>>> bib: 5, name: Christophe Laporte
```

SLI vs. CLI — Discussion

- Crude simplification: SLI = CLI in disguise
  - SLI preprocessor generates CLI code

Source file: **staticquery.c** (generated by preprocessor)

```c
/* Processed by ecpg (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1)) */
/* These include files are added by the preprocessor */
#include <ecpglib.h>
#include <ecpgerrno.h>
#include <sqlca.h>
/* End of automatic include section */

int main()
{
    /* exec sql begin declare section */
    char v_bib [ 32 ] , v_name [ 32 ] ;
    const char * target = "cs2102_tdfdb@localhost" ;
    const char * user = "demo" ;
    const char * passwd = "demo" ;
    /* exec sql end declare section */

    // Connect to database
    { ECPGconnect(__LINE__, 0, target , user , passwd , NULL, 0); }

    // Open cursor
    { ECPGdo(__LINE__, 0, 1, NULL, 0, ECPGst_normal, "declare cursor cursor for select bib , name from riders limit 3", ECPGt_EOIT, ECPGt_EORT);}

    ...

    return 0;
}
```



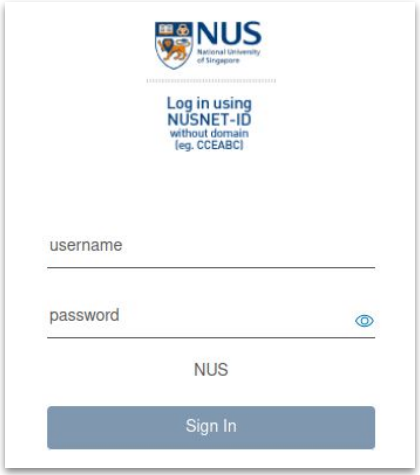
Overview

- **Writing Database Applications**
  - Motivation
  - Statement Level Interface
  - Call Level Interface
  - **SQL Injection Attacks**
- SQL Functions and Procedures
  - Motivation & Overview
  - Main Parameters: arguments, language, return values
  - Assignments & Control structures
  - Cursors
- Summary

SQL Injection Attack

- SQL Injection Attack
  - Class of cyber attacks on dynamic SQL
  - Goal of attack: execute unintended (typically malicious) SQL statements
  - Typical cause: dynamic queries are generated by merging/concatenating strings

- Common attack point
  - Omnipresent form fields in Web interfaces
  - Entered values define some SQL statement



CLI Example Revisited

Parameter value of intended range

Generation of query by merging strings

```python
import psycopg # Host language library (here psycopg for Python)

# Set a user-defined value (here: max. number of riders returned)
limit = 5

# Connect to database
connection = psycopg.connect("host=localhost dbname=cs2102_tdfdb user=demo password=demo")

# Create cursor
cursor = connection.cursor()

# Open cursor by executing query (string parameter passed to execute() method)
#cursor.execute("SELECT * FROM riders LIMIT %s", (limit,))  # Prepared Statement (under the hood)
cursor.execute("SELECT * FROM riders LIMIT " + str(limit))  # Query as a result from merging strings

# Loop over all results until no next tuple is returned
while True:
    row = cursor.fetchone()
    if row is None:
        break
    print(f">>> bib: {row[0]}, name: {row[1]}")

# Cleanup
cursor.close()
connection.commit()
connection.close()
```

```
>>> bib: 1, name: Jonas Vingegaard
>>> bib: 2, name: Tiesj Benoot
>>> bib: 3, name: Wilco Kelderman
>>> bib: 4, name: Sepp Kuss
>>> bib: 5, name: Christophe Laporte
```

SQL Injection Attack Demo

Malicious parameter value by an attacker*

Generation of query by merging strings

```python
import psycopg # Host language library (here psycopg for Python)

# Set a user-defined value (here by an attacker)
limit = "5; INSERT INTO attack_log DEFAULT VALUES;"

# Connect to database
connection = psycopg.connect("host=localhost dbname=cs2102_tdfdb user=demo password=demo")

# Create cursor
cursor = connection.cursor()

print("Submitted statement:", "SELECT * FROM riders LIMIT " + str(limit))

# Open cursor by executing query (string parameter passed to execute() method)
#cursor.execute("SELECT * FROM riders LIMIT %s", (limit,))  # Prepared Statement (under the hood)
cursor.execute("SELECT * FROM riders LIMIT " + str(limit))  # Query as a result from merging strings

# Loop over all results until no next tuple is returned
# (Results from "real" query are still returned but omitted here)

# Cleanup
cursor.close()
connection.commit()
connection.close()
```

```
Submitted statement: SELECT * FROM riders LIMIT 5; INSERT INTO attack_log DEFAULT VALUES;
```

This command would have thrown an error!

* finding such values for a successful attack generally requires some insider knowledge and/or a lot of trial-&-error

SQL Injection Attacks — Key Takeaway Message

> ⚠ Warning
>
> - Don't manually merge values to a query: hackers from a foreign country will break into your computer and steal not only your disks, but also your cds, leaving you only with the three most embarrassing records you ever bought. On cassette tapes.
> - If you use the `%` operator to merge values to a query, con artists will seduce your cat, who will run away taking your credit card and your sunglasses with them.
> - If you use `+` to merge a textual value to a string, bad guys in balaclava will find their way to your fridge, drink all your beer, and leave your toilet seat up and your toilet paper in the wrong orientation.
> - You don't want to manually merge values to a query: use the provided methods instead.

Source file: psycopg docs

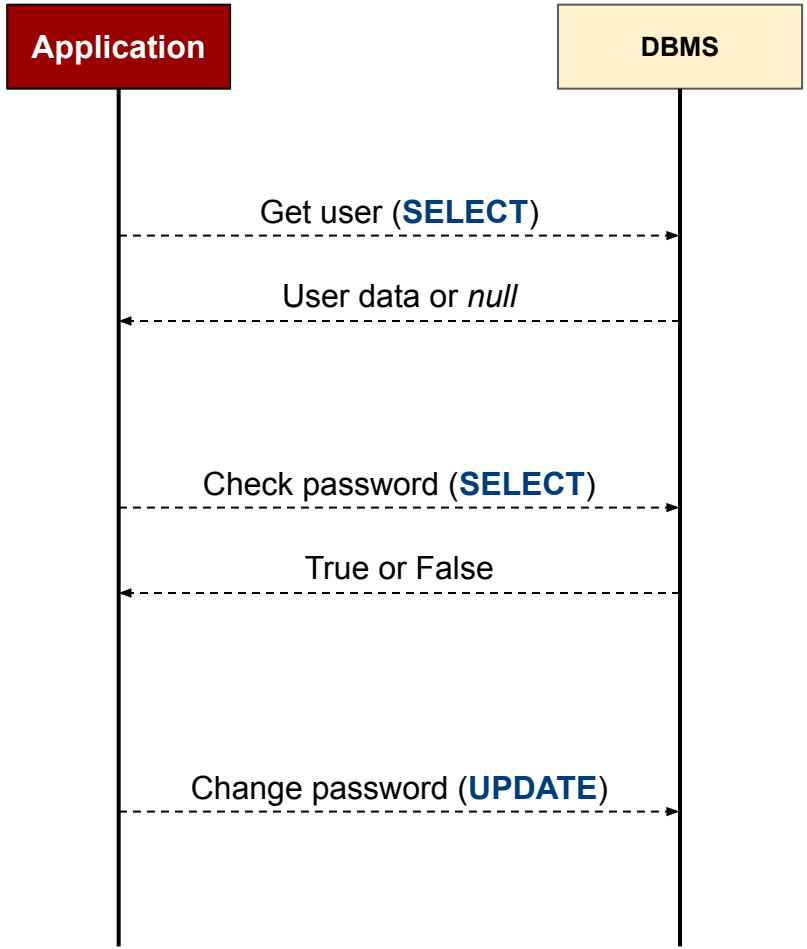
Overview

- Writing Database Applications
  - Motivation
  - Statement Level Interface
  - Call Level Interface
  - SQL Injection Attacks
- **SQL Functions and Procedures**
  - **Motivation & Overview**
  - Main Parameters: arguments, language, return values
  - Assignments & Control structures
  - Cursors
- Summary

Motivation

- Very common in practice
  - Tasks requiring multiple DB operations
  - Any combination of Reads and Writes
- Example: update user password
  - Check that user does exist
  - Check that new password differs from old one
  - If all OK, update password

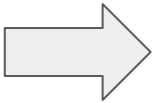
➜ 3 separate requests/accesses to DB that belong together to perform a task



Motivation

- What are the obvious problems?
  - Application and DB may run on different machines ➜ poor performance / DB becomes bottleneck
  - Different DB operations only loosely connected ➜ difficult to ensure "all or nothing" behavior

- What would we like to do?
  - Move (some) application logic into DB
  - Group DB operations that form a task together within a execution
  - Treat task as a single DB operation



Stored functions and procedures

- Collection of SQL statements and procedural logic
- Precompiled and reusable code
- Allow execute multiple database operations as a single unit

Motivation

- Why do we want/need procedural logic?
  - Application logic that requires assignments, conditionals, or loops
  - Queries that cannot be expressed using basic SQL

id	name	points	graduated
1	Bob	94	TRUE
2	Eve	82	FALSE
3	Sam	65	FALSE
4	Liz	86	TRUE
5	Tom	90	TRUE
6	Sue	94	FALSE
7	Zac	75	FALSE
8	Ida	84	TRUE
9	Leo	91	FALSE
10	Pam	70	FALSE

Sort all students by points (descending) and list the differences in points in the sequence

Question: Can we write a SQL query that returns the desired result?

name	gap
Bob	0
Sue	0
Leo	3
Tom	1
Liz	4
Ida	2
Zac	2
Eve	7
Pam	5
Sam	5

SQL Functions and Procedures

- SQL-based procedural language
  - ISO standard: **SQL/PSM** (SQL/Persistent Stored Modules)
  - Unfortunately, different DBMS have their own "flavor"

Oracle:	PL/SQL
PostgreSQL:	PL/pgSQL
SQL Server:	Transact-SQL

Advantages (only if done right!)

- Better performance
- Code reuse / higher productivity
- Ease of maintenance
- Added security

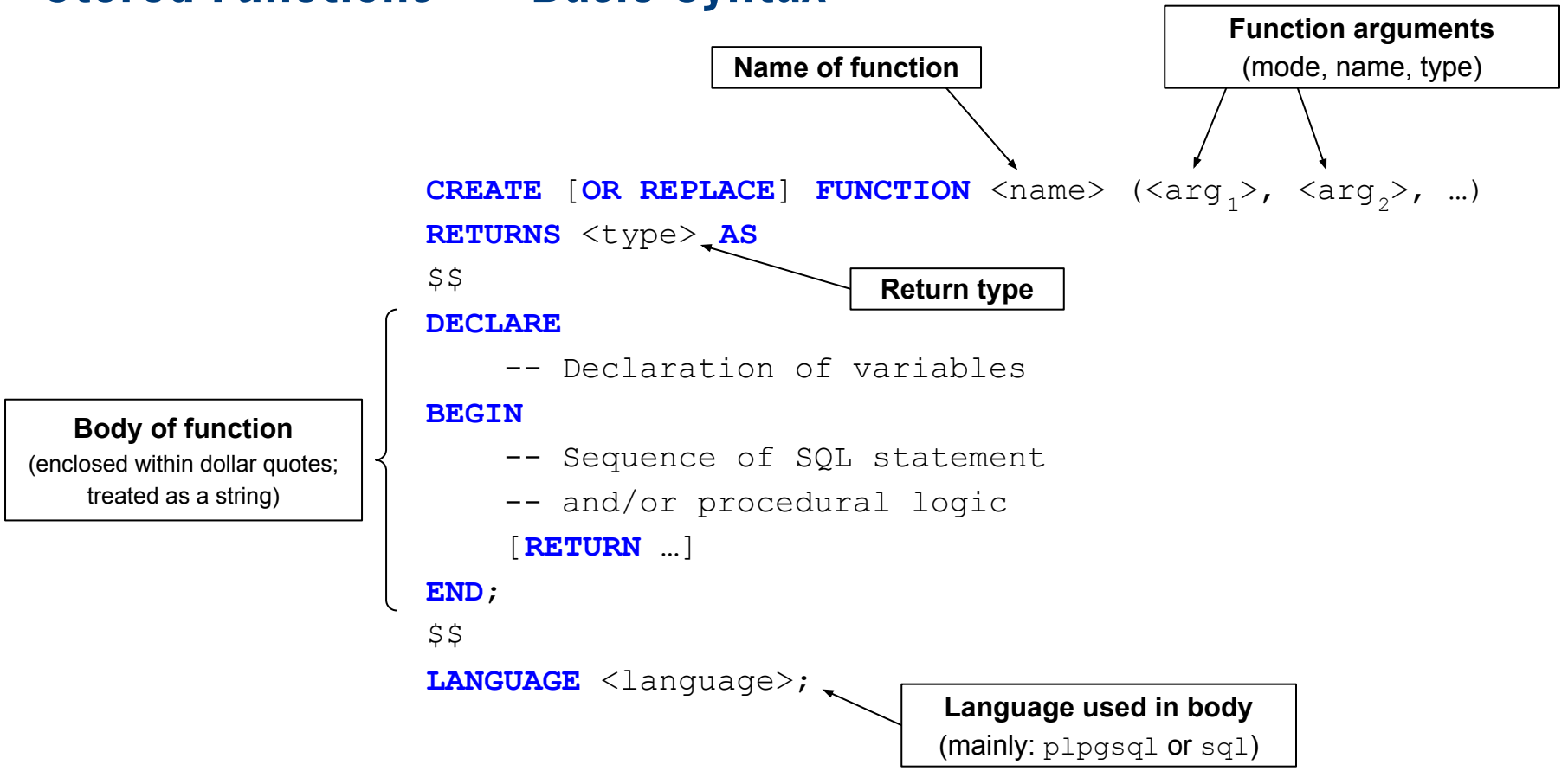
Disadvantages

- Testing & debugging more challenging
- Limited portability / vendor lock-in
- No simple versioning of code
- Not the most intuitive language

Overview

- Writing Database Applications
  - Motivation
  - Statement Level Interface
  - Call Level Interface
  - SQL Injection Attacks
- **SQL Functions and Procedures**
  - **Motivation & Overview**
  - Main Parameters: arguments, language, return values
  - Assignments & Control structures
  - Cursors
- Summary

Stored Functions — Basic Syntax



PostgreSQL docs with complete syntax

Running Example Database

- Toy database: Just 1 table
  - Oversimplified table `students`

```
CREATE TABLE students (
 id SERIAL PRIMARY KEY,
 name TEXT NOT NULL,
 points INTEGER DEFAULT 0,
 graduated BOOLEAN DEFAULT FALSE
);
```

id	name	points	graduated
1	Bob	94	TRUE
2	Eve	82	FALSE
3	Sam	65	FALSE
4	Liz	86	TRUE
5	Tom	90	TRUE
6	Sue	94	FALSE
7	Zac	75	FALSE
8	Ida	84	TRUE
9	Leo	91	FALSE
10	Pam	70	FALSE

Overview

- Writing Database Applications
  - Motivation
  - Statement Level Interface
  - Call Level Interface
  - SQL Injection Attacks
- **SQL Functions and Procedures**
  - Motivation & Overview
  - **Main Parameters: arguments, language, return values**
  - Assignments & Control structures
  - Cursors
- Summary

Stored Functions — Function Arguments

- Each argument is described by 3 values
  - **Mode**: mode of argument (mainly: **`IN`**, **`OUT`**, **`INOUT`**)
  - **Name**: name of argument
    (note: names are actually not mandatory!)
  - **Type**: datatype of argument
    (`INTEGER`, `VARCHAR`, …, user-defined, etc.)

same like in most programming languages

IN	**OUT**	**INOUT**
Default	Explicitly specified	Explicitly specified
Value is passed to a function	Value is returned by a function	Value is passed to the function which returns another updated value
Behaves like constants	Behaves like an uninitialized variable	Behaves like an initialized variable
Value cannot be assigned	Value must be assigned	Value can/should be assigned

Simple Example — Add 2 Integers

Quick Quiz: In which case is the right alternative more convenient?

```
CREATE OR REPLACE FUNCTION add
    (IN a INT, IN b INT)
RETURNS INTEGER AS
$$
DECLARE
    sum INT;
BEGIN
    sum := a + b;
    RETURN sum;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT add(2, 3);
```

add
5

```
CREATE OR REPLACE FUNCTION add
    (IN a INT, IN b INT, OUT sum INT)
AS
$$
BEGIN
      sum := a + b;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT sum FROM add(2, 3);
```

sum
5

Quick Quiz

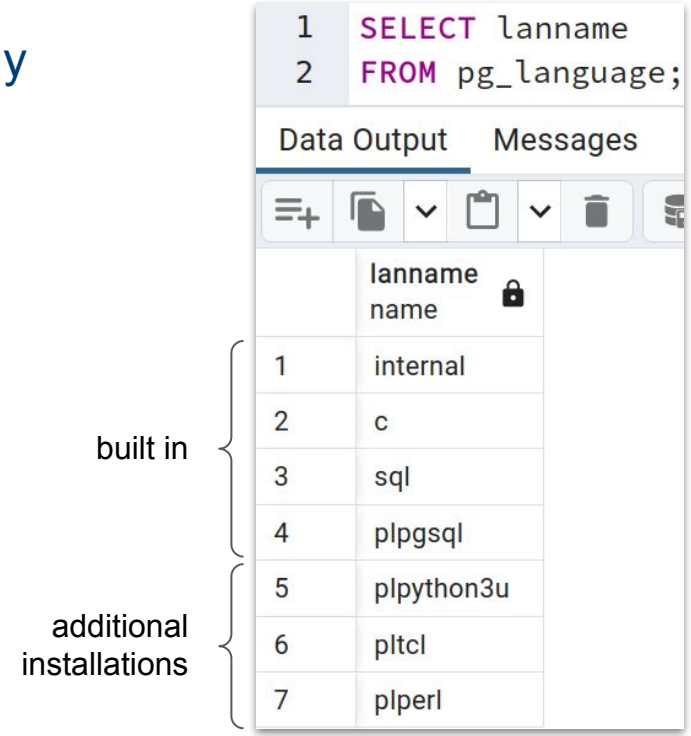
```
CREATE OR REPLACE FUNCTION add (INT, INT)
RETURNS INTEGER AS
$$
DECLARE
    sum INTEGER;
BEGIN
    sum :=
    RETURN sum;
END;
$$
LANGUAGE plpgsql;
```

How do we have to complete the function to make it valid?

Stored Functions — Language

- Explicit specification of language of function body
  - Recall: function body is interpreted as a string
  - Many supported languages: **SQL**, **PL/pgSQL**, PL/Python, PL/Java, PL/Perl, PL/TCL, C, etc.

- So, what to choose?
  - Choice of language typically depends on task
  - Focus on database operations: SQL, PL/pgSQL
    (good for code with lot of SQL queries due native support of SQL)
  - Other languages more suitable for custom tasks
    (e.g., PL/Perl for string operations, C for high performance)



Stored Functions — `sql` vs `plpgsql`

- When to choose **`sql`**?
  - Body consists of only SQL statements
    (i.e., not procedural elements required to solve task)
  - Often a wrapper of single/few SQL statements
  - Simpler syntax: no `BEGIN…END`

- When to choose **`plpgsql`**?
  - Procedural elements are required (duh!)
  - Dynamic SQL: statements generated at runtime
    (attention: take care to avoid SQL injection attacks!)
  - For trigger functions (see next lecture)

Example: Remove all students who have graduated and return the number of remaining students.

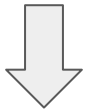
```
CREATE FUNCTION clean_students()
RETURNS INTEGER AS
$$
    -- Delete all students
    -- Who have graduated
    DELETE FROM students
    WHERE graduated = TRUE;
    -- Return remaining student count
    - If SELECT, no RETURN needed
    SELECT COUNT(*) AS num_students
    FROM students;
$$
LANGUAGE sql;
```

Note: There are many other criteria (e.g., performance, optimization, reusability, transaction handling) that are beyond our scope here.

Stored Functions — Return Values (beyond simple values)

One Existing Tuple

```
CREATE FUNCTION get_top_student()
RETURNS students AS
$$
    SELECT *
    FROM students
    ORDER BY points DESC
    LIMIT 1;
$$
LANGUAGE sql;
```



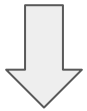
```
SELECT id, name, points
FROM get_top_student();
```

id	name	points
1	Bob	94

Stored Functions — Return Values (beyond simple values)

One Existing Tuple

```
CREATE FUNCTION get_top_student()
RETURNS students AS
$$
    SELECT *
    FROM students
    ORDER BY points DESC
    LIMIT 1;
$$
LANGUAGE sql;
```

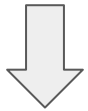


```
SELECT id, name, points
FROM get_top_student();
```

id	name	points
1	Bob	94

Set of Existing Tuples

```
CREATE FUNCTION get_enrolled_students()
RETURNS SETOF students AS
$$
    SELECT *
    FROM students
    WHERE graduated = FALSE;
$$
LANGUAGE sql;
```

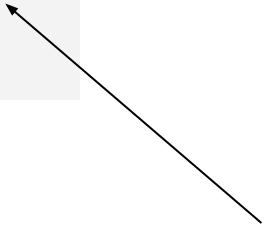


```
SELECT id, name
FROM get_enrolled_students();
```

id	name
2	Eve
3	Sam
6	Sue
7	Zac
9	Leo
10	Pam

Quick Quiz

```
CREATE FUNCTION get_top_student()
RETURNS students AS
$$
    SELECT id, name, points
    FROM students
    ORDER BY points DESC
    LIMIT 1;
$$
LANGUAGE sql;
```



This will throw an error…**why?**

Stored Functions — Return Values (beyond simple values)

One New Tuple

```
CREATE FUNCTION get_top_student_count
    (OUT points INT, OUT cnt INT)
RETURNS RECORD AS
$$
    SELECT points, COUNT(*)
    FROM students
    WHERE points = (SELECT MAX(points)
                    FROM students)
    GROUP BY points;
$$
LANGUAGE sql;
```

```
SELECT points, cnt
FROM get_top_student_count();
```

points	cnt
94	2

Important: If we use `RECORD`, we must have at least two `OUT` parameters!

Stored Functions — Return Values (beyond simple values)

One New Tuple

```
CREATE FUNCTION get_top_student_count
    (OUT points INT, OUT cnt INT)
RETURNS RECORD AS
$$
    SELECT points, COUNT(*)
    FROM students
    WHERE points = (SELECT MAX(points)
                    FROM students)
    GROUP BY points;
$$
LANGUAGE sql;
```

```
SELECT points, cnt
FROM get_top_student_count();
```

points	cnt
94	2

Set of New Tuples

```
CREATE FUNCTION get_group_counts
    (OUT graduated BOOLEAN, OUT cnt INT)
RETURNS SETOF RECORD AS
$$
    SELECT graduated, COUNT(*) as cnt
    FROM students
    GROUP BY graduated;
$$
LANGUAGE sql;
```

```
SELECT graduated, cnt
FROM get_group_counts();
```

graduated	cnt
FALSE	6
TRUE	4

Quick Quiz

```
CREATE FUNCTION get_group_counts
    (OUT graduated BOOLEAN, OUT cnt INT)
RETURNS ~~SETOF~~ RECORD AS
$$
    SELECT graduated, COUNT(*) as cnt
    FROM students
    GROUP BY graduated;
$$
LANGUAGE sql;
```

```
SELECT graduated, cnt
FROM get_group_counts();
```

What will be the result of this query?

Stored Functions — Return Values (beyond simple values)

Set of New Tuples

```
CREATE FUNCTION get_group_counts()
RETURNS TABLE(graduated BOOLEAN, cnt INT)
AS
$$
   SELECT graduated, COUNT(*)
   FROM students
   GROUP BY graduated;
$$
LANGUAGE sql;
```

```
SELECT graduated, cnt
FROM get_group_counts();
```

graduated	cnt
FALSE	6
TRUE	4

```
CREATE FUNCTION get_group_counts
    (OUT graduated BOOLEAN, OUT cnt INT)
RETURNS SETOF RECORD AS …
```

Equivalent effect…but always?

```
CREATE FUNCTION get_group_counts()
RETURNS TABLE(graduated BOOLEAN, cnt INT)
AS …
```

Quick Quiz

```
CREATE FUNCTION name (…)
RETURNS SETOF RECORD AS …
```

vs.

```
CREATE FUNCTION name (…)
RETURNS TABLE(…) AS …
```

What is a case where both alternatives are **NOT** interchangeable?

Stored Functions — Return Values (beyond simple values)

No Return Value

```
CREATE FUNCTION add_bonus (sid INT, amount INT)
RETURNS VOID AS
$$
    UPDATE students
    SET points = points + amount
    WHERE id = sid;
$$
LANGUAGE sql;
```

```
SELECT add_bonus(3, 5);
```

add_bonus

Actual output / result of SELECT statement

Resulting table (not the function output!)

id	name	points	graduated
1	Bob	94	TRUE
2	Eve	82	FALSE
3	Sam	70	FALSE
4	Liz	86	TRUE
5	Tom	90	TRUE
6	Sue	94	FALSE
7	Zac	75	FALSE
8	Ida	84	TRUE
9	Leo	91	FALSE
10	Pam	70	FALSE

Stored Procedures

- Stored procedures
  - Essentially the same syntax as for functions
  - Most obvious difference: not **`RETURNS`** clause
  - Invoked using **`CALL`** command

```
CREATE PROCEDURE add_bonus_proc(sid INT, amount INT)
AS
$$
    UPDATE students
    SET points = points + amount
    WHERE id = sid;
$$
LANGUAGE sql;
```

```
CALL add_bonus(3, 5);
```

No output / result, but table gets updated

Question: Wait, so is a procedure just a function that does not return anything?

Stored Functions vs Stored Procedures (in PostgreSQL)

- Despite similar look-&-feel, functions and procedures do differ
  - Functions must return something; procedures do not have to but still can
    (procedures can still return values by mean of **INOUT** and **OUT** parameters; the latter since Version 14)
  - Procedures can commit or roll back transactions during its execution; functions can not
  - Unlike functions, procedures cannot be invoked in DML commands (**SELECT**, **INSERT**, **UPDATE**, **DELETE**)
  - Procedures a invoked in isolation using **CALL** (functions are always invoked in SELECT statements)

```
CREATE PROCEDURE add_proc
    (IN a INT, IN b INT, OUT sum INT)
AS
$$
BEGIN
    sum := a + b;
END;
$$
LANGUAGE plpgsql;
```

```
DO
$$
DECLARE
      sum INT;
BEGIN
      CALL add_proc(2, 3, sum);
      RAISE NOTICE 'Sum: %', sum;
END
$$;
```

```
NOTICE:  Sum: 5
```

Stored Functions vs Stored Procedures (in PostgreSQL)

- Summary
  - Function must return something, but it can be `VOID`
  - Procedures can return something (using `INOUT` and `OUT` parameters)

- Best practice (most of the time)
  - Return value(s) ➜ `CREATE FUNCTION`
  - No return value ➜ `CREATE PROCEDURE`

Overview

- Writing Database Applications
  - Motivation
  - Statement Level Interface
  - Call Level Interface
  - SQL Injection Attacks
- **SQL Functions and Procedures**
  - Motivation & Overview
  - Main Parameters: arguments, language, return values
  - **Assignments & Control Structures**
  - Cursors
- Summary

Assignments

- Basic assignment with `:=`
  - `age := 29;`
  - `name := 'Alice';`

- **`SELECT … INTO …`**
  - Assignment of query result to declared variable(s)

Example: Get the mark of a student; automatically consider any bonus.

```
CREATE FUNCTION get_mark(sid INT, bonus INT DEFAULT 0)
RETURNS INTEGER AS
$$
DECLARE
    mark INTEGER;
BEGIN
    -- Get current mark of students
    SELECT points INTO mark FROM students WHERE id = sid;
    mark := mark + bonus; -- Add bonus to mark
    RETURN mark;
END;
$$
LANGUAGE plpgsql;
```

optional argument

```
SELECT get_mark(3, 10);
```

get_mark
75

```
SELECT get_mark(3);
```

get_mark
65

Quick Quiz

```
…
$$
DECLARE
    dob DATE;
BEGIN
    dob := 
    …
END;
$$
LANGUAGE plpgsql;
```

How can we set the value for `dob` **manually**?

Control Structures

- Conditionals:
  - 4 types of **`IF`** expressions
    - **`IF … THEN … END IF`**
    - **`IF … THEN … ELSE … END IF`**
    - **`IF … THEN … ELSIF … THEN … ELSE … END IF`**
  - 2 types of **`CASE`** expressions
    - **`CASE … WHEN … THEN … ELSE … END CASE`**
    - **`CASE WHEN … THEN … ELSE … END CASE`**

Note: PostgreSQL also offers **`ELSEIF`** as an alias for **`ELSIF`**.

- Simple Loops
  - **`LOOP … END LOOP`** (typically requires **`EXIT`**…**`WHEN`**… to jump out of loop)
  - **`WHILE … LOOP … END LOOP`**
  - **`FOR … IN … LOOP … END LOOP`**

Relevant PostgreSQL Docs

Conditional & Simple Loops — Example

Compute the sum of the first ***n*** integers;
if ***n*** is negative, return 0.

```
CREATE FUNCTION sum_n(IN n INT)
RETURNS INT AS
$$
DECLARE
    sum INT;
BEGIN
    sum := 0;
    IF n <= 0 THEN
        RETURN sum;
    END IF;
    FOR val IN 1..n LOOP
        sum := sum + val;
    END LOOP;
    RETURN sum;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT sum_n(5);
```

sum_n
15

```
SELECT sum_n(-5);
```

sum_n
0

Side Note: Errors & Messages

Compute the sum of the first ***n*** integers;
if ***n*** is negative, **raise an exception**.

```
CREATE FUNCTION sum_n(IN n INT)
RETURNS INT AS
$$
DECLARE
    sum INT;
BEGIN
    sum := 0;
    IF n <= 0 THEN
        RAISE EXCEPTION 'n<0 bad!';
    END IF;
    FOR val IN 1..n LOOP
        sum := sum + val;
    END LOOP;
    RETURN sum;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT sum_n(5);
```

sum_n
15

```
SELECT sum_n(-5);
```

```
ERROR:  n<0 bad!
CONTEXT:  PL/pgSQL function sum_n(integer) line 7 at RAISE
```

Side Note: Errors & Messages

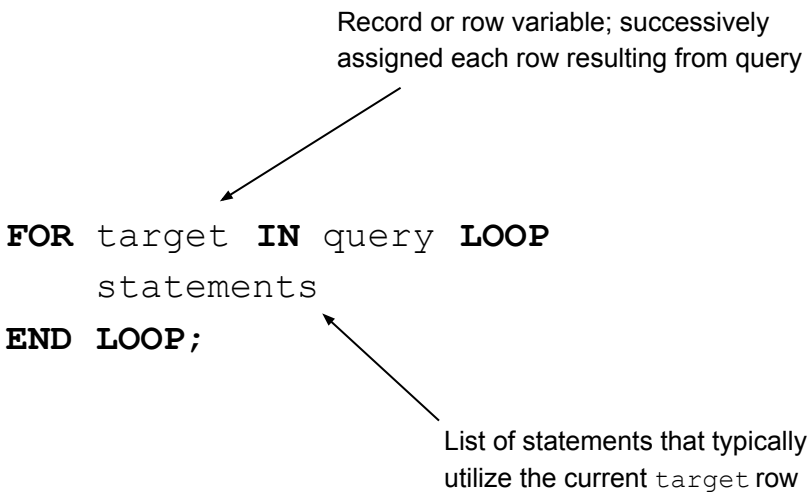
- Errors & message in PostgreSQL: **`RAISE`**
  - 6 different raise levels available in PostgreSQL

<table>
<tr><td>RAISE DEBUG</td><td rowspan="5">Generate messages of different priority levels
Whether messages of a particular priority are reported to the client, depends on the PostgreSQL configuration</td></tr>
<tr><td>RAISE LOG</td></tr>
<tr><td>RAISE INFO</td></tr>
<tr><td>RAISE NOTICE</td></tr>
<tr><td>RAISE WARNING</td></tr>
<tr><td>RAISE EXCEPTION</td><td>Raises an error
Typically aborts current transaction</td></tr>
</table>

Relevant PostgreSQL Docs

Looping through Query Results — Example

- Common use case: loop through a query result
  - Special **`FOR`** loop to iterate through the results and manipulate data
  - Basic syntax:



Looping through Query Results — Example

```
CREATE FUNCTION compute_points_gaps()
RETURNS TABLE(name TEXT, points INT, gap INT) AS
$$
DECLARE
    s RECORD; prev INT;
BEGIN
    prev := -1;
    FOR s IN SELECT *
             FROM students
             ORDER BY points DESC
    LOOP
        name := s.name;
        points := s.points;
        IF prev >= 0 THEN
            gap := prev - s.points;
        ELSE
            gap := 0;
        END IF;
        RETURN NEXT;
        prev := s.points;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT name, gap
FROM compute_points_gaps();
```

name	gap
Bob	0
Sue	0
Leo	3
Tom	1
Liz	4
Ida	2
Zac	2
Eve	7
Pam	5
Sam	5

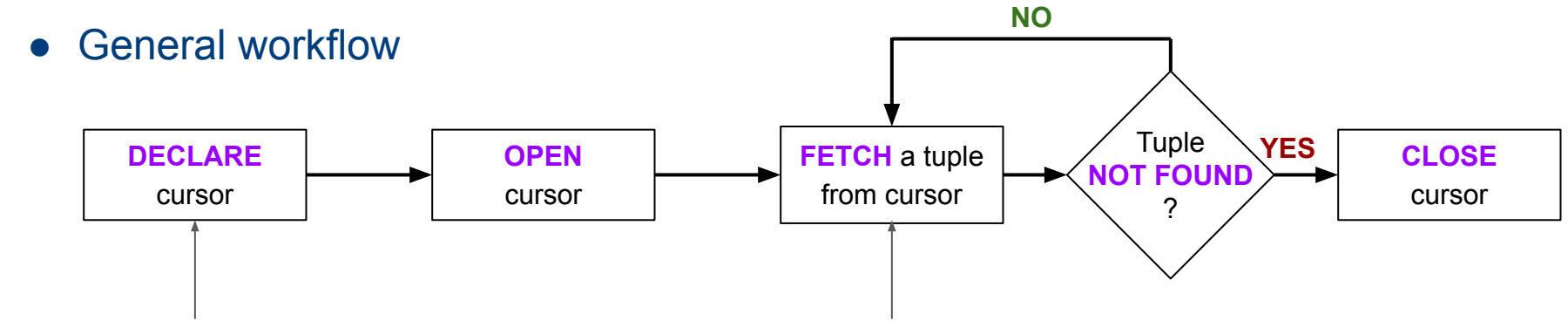
Overview

- Writing Database Applications
  - Motivation
  - Statement Level Interface
  - Call Level Interface
  - SQL Injection Attacks
- **SQL Functions and Procedures**
  - Motivation & Overview
  - Main Parameters: arguments, language, return values
  - Assignments & Control Structures
  - **Cursors**
- Summary

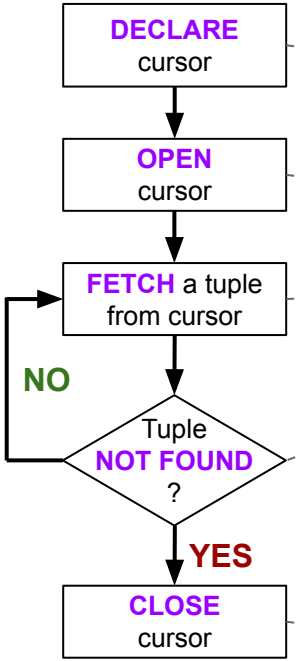
Cursors — Purpose & Basic Structure

- Purpose of cursors
  - Encapsulates the query and allows to access each individual row return by a **`SELECT`**
    (instead of executing the whole query to get the complete result at once)
  - Additional benefit: avoids memory overrun when the query result is very large
    (typically not an issue in practice since **`FOR`** loops automatically use cursors under the hood)

- General workflow



Cursors — Example




```
CREATE FUNCTION compute_points_gaps()
RETURNS TABLE(name TEXT, points INT, gap INT) AS
$$
DECLARE
    c CURSOR FOR (SELECT * FROM students ORDER BY points DESC);
    s RECORD; prev INT;
BEGIN
    prev := -1;
    OPEN c;
    LOOP
        FETCH c INTO s;
        EXIT WHEN NOT FOUND;
        name := s.name;
        points := s.points;
        IF prev >= 0 THEN
            gap := prev - s.points;
        ELSE
            gap := 0;
        END IF;
        RETURN NEXT;
        prev := s.points;
    END LOOP;
    CLOSE c;
END;
$$
LANGUAGE plpgsql;
```

Question: Uh, we can do this without cursors? So why do we need them?

Cursors — Directions

- Advantages of cursors
  - Flexibly "navigating" through query results in different **directions**
  - Two commands to navigate
    - `FETCH`: move to row and read data
    - `MOVE`: only move to row (no read)
- Possible directions

`NEXT`	Fetch the next row (default)
`PRIOR`	Fetch the prior row
`FIRST`	Fetch the first row of the query (same as `ABSOLUTE 1`)
`LAST`	Fetch the last row of the query (same as **`ABSOLUTE -1`**)
`ABSOLUTE` ***n***	• Fetch the *n*-th row of the query, if *n* >=0 • Fetch abs(*n*)-th row from the end, if *n* < 0. • **`ABSOLUTE 0`** positions before the first row
`RELATIVE` ***n***	• Fetch the *n*-th succeeding row, if *n* >= 0 • Fetch the abs(*n*)-th prior row, if n < 0 • Position before first row or after last row if *n* is out of range • **`RELATIVE 0`** re-fetches the current row, if any
`FORWARD`	Fetch the next row (same as **`NEXT`**)
`BACKWARD`	Fetch the prior row (same as **`PRIOR`**).

Relevant PostgreSQL Docs

Cursors — Example (beyond NEXT)

```
CREATE OR REPLACE FUNCTION median_points()
RETURNS NUMERIC AS
$$
DECLARE
    c CURSOR FOR (SELECT * FROM students ORDER BY points DESC);
    s1 RECORD; s2 RECORD; num_students INT;
BEGIN
    OPEN c;
    SELECT COUNT(*) INTO num_students FROM students;
    IF num_students%2 = 1 THEN
        FETCH ABSOLUTE (num_students+1)/2 FROM c INTO s1;
        RETURN s1.points;
    ELSE
        FETCH ABSOLUTE num_students/2 FROM c INTO s1;
        FETCH NEXT FROM c INTO s2;
        RETURN (s1.points+s2.points)/2;
    END IF;
    CLOSE c;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT median_points();
```

median_points
85

…	points	…
…	94	…
…	94	…
…	91	…
…	90	…
…	86	…
…	84	…
…	82	…
…	75	…
…	70	…
…	65	…

Quick Quiz

```
…
    ELSE
        FETCH ABSOLUTE num_students/2 FROM c INTO s1;
        FETCH NEXT FROM c INTO s2;
        RETURN (s1.points+s2.points)/2;
    END IF;
…
```

How could we **rewrite** this line but preserve the overall functionality?

Dynamic Cursors — Example

```
CREATE OR REPLACE FUNCTION median_points(IN has_graduated BOOLEAN)
RETURNS NUMERIC AS
$$
DECLARE
    c CURSOR (grad BOOLEAN) FOR (SELECT * FROM students
                                 WHERE graduated = grad
                                 ORDER BY points DESC);
    s1 RECORD; s2 RECORD; num_students INT;
BEGIN
    OPEN c(has_graduated);
    SELECT COUNT(*) INTO num_students
    FROM students WHERE graduated = has_graduated;
    IF num_students%2 = 1 THEN
        FETCH ABSOLUTE (num_students+1)/2 FROM c INTO s1;
        RETURN s1.points;
    ELSE
        FETCH ABSOLUTE num_students/2 FROM c INTO s1;
        FETCH NEXT FROM c INTO s2;
        RETURN (s1.points+s2.points)/2;
    END IF;
    CLOSE c;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT median_points(TRUE);
```

median_points
88

```
SELECT median_points(FALSE);
```

median_points
78

Overview

- Writing Database Applications
  - Motivation
  - Statement Level Interface
  - Call Level Interface
  - SQL Injection Attacks
- SQL Functions and Procedures
  - Motivation & Overview
  - Main Parameters: arguments, language, return values
  - Assignments & Control Structures
  - Cursors
- **Summary**

Summary

- Stored functions & procedures
  - Combine SQL statements and procedural logic into a single unit
  - Move (some) application logic into the database
  - Support query result not possible to with "normal" SQL
    (at least not with the feature set covered in this course)

- Implementation
  - Support for different programming languages (Python, Perl, Java, JavaScript, Lua, etc.)
  - Focus here: **PL/pgSQL** (PostgreSQL variation of the SQL/PSM standard)

- Next lecture: **Triggers**
  - Automatically execute functions when "interesting events" happen

Solutions to Quick Quizzes

- Slide 30: It is more convenient to return more than one value
- Slide 31: `$1+$2`
- Slide 36: We have to return all the columns of the table
- Slide 39: Works, but only one record/tuple will be returned
- Slide 41: SETOF records requires at least 2 output parameters
- Slide 48: `d = TO_DATE('2023-10-10','YYYY-MM-DD');`
- Slide 60: Only 1-line alternatives
  - `FETCH c INTO s2;`
  - `FETCH RELATIVE 1 FROM c INTO s2;`
  - `FETCH ABSOLUTE num_students/2+1 FROM c INTO s2;`