# Advanced SQL

# Advanced SQL

- How to make SQL work: Basic and Intermediate SQL
- How to make SQL work better: Advanced SQL
  - SQL extensions: stored procedure/non relational data
  - Database management: system catalog, monitoring, and security
  - Performance and execution plan
    - Index
    - Join
  - Transaction

# User Defined Function and Stored Procedure

# Function in PostgreSQL

- Provides a result based on the given input
- Used in SQL statements to provide complex data processing functionalities

# User Defined Function

- Users can define their own functions if the functions provided by PostgreSQL do not meet the need.
  - Similar to the functions defined by PostgreSQL
  - But defined by users as a collection of SQL code
  - Has a return value
    - RETURN statement

# User Defined Function

```
CREATE FUNCTION function_name (function_arguments)
RETURNS return_type
AS
    $$
        DECLARE function_variables
        BEGIN
            function_statements
        END;
    $$
LANGUAGE PLPGSQL;
```

# Function in PostgreSQL

```
CREATE FUNCTION function_name
```

- Declare the function object name
- Can also add "`OR REPLACE`" after "`CREATE`"

```
CREATE OR REPLACE FUNCTION function_name
```

# Function in PostgreSQL

```
CREATE FUNCTION function_name (function_arguments)
```

- Arguments that can be used in the function statements

  ```
  Argument_Name Argument_Data_Type
  ```

  For example:

  ```
  iName VARCHAR
  ```

# Function in PostgreSQL

```
RETURNS return_type
```

- Defines the data type that is returned by the function

  For example:

  ```
  RETURNS VARCHAR
  ```

# Function in PostgreSQL

```
AS
    $$
        DECLARE function_variables
        BEGIN
            function_statements
        END;
    $$
```

- AS block is used to define the function logic.
- $$ is used to identify the boundary of AS block.

# Function in PostgreSQL

`$$`

- Defines the boundary of AS block.
- Can have a name, e.g.,

```
$My_Code_Block$
    DECLARE ...
    BEGIN ... END;
$My_Code_Block$
```

# Function in PostgreSQL

```
DECLARE function_variables
```

- Defines the variables used in this function.
- Each variable is a pair of variable name and data type combination, e.g.,

```
iAffi VARCHAR
```

# Function in PostgreSQL

```
BEGIN function_statements END;
```

- Defines the execution logic used in this function.
- Composed of one or multiple SQL statements separated by semicolon.
- Must have a RETURN statement in the end, returning a value that has the data type defined in RETURNS block.

# Function in PostgreSQL

```
LANGUAGE PLPGSQL
```

- Defines the language used in this function. Usually PLPGSQL.
- This block can be placed in front of AS block.

# Coding Function

- Passing parameters into functions
- Using SQLs with variables
  - Insert/Update/Delete table in functions
  - Avoid creating permanent objects in stored procedure
  - Use RAISE NOTICE to print out messages for debugging
- SELECT into a value and return it

# Stored Procedures

- Similar to functions, but do not return a value.

```
CREATE PROCEDURE procedure_name (procedure_arguments)
AS
    $$
        DECLARE procedure_variables
        BEGIN
            procedure_statements
        END;
    $$
LANGUAGE PLPGSQL
```

# Functions and Stored Procedures

- Functions and Stored Procedure:
  - Important: a significant portion of the SQL usage is functions and stored procedures
  - Procedure code varies by DBMS

# Trigger

# Trigger

- Functions can be triggered by a certain event
  - INSERT, UPDATE, DETELE
- Triggers can access:
  - Previous data (DELETE, UPDATE)
  - New data (INSERT, UPDATE)

# Trigger Definition

- Create a function that returns TRIGGER data type
  - Use OLD to represent the old data row, and NEW to represent the new data row
- Create triggers to define:
  - Table involved
  - Event involved: INSERT, UPDATE, DELETE, etc
  - Action involved: the trigger function name

# Trigger

- Triggers may have serious performance implications. Use with caution.

# Cursor

# Procedural Processing of Data

- So far all SQL statements that we have learned work on data *sets*
  - Set based algorithms (Relational Algebra) cannot handle processing between rows because there is no order between objects in a set.
  - Procedural processing capability is required for row-by-row processing.
- CURSOR: processing one row at a time from a dataset

# CURSOR

- Cursor definition and usage: Vendor specific, but basic logic is the same
    1. Define cursor based on query, usually with ORDER BY
    2. Start/initialize cursor
    3. Loop through cursor to process each row
        a. Store previous row in variables if necessary
- Must be defined inside a transactional block, such as a function.

# CURSOR

- Define cursor based on query

```
DECLARE
    cursor1 CURSOR FOR SELECT * FROM Instructor;
DECLARE
    cursor2 CURSOR (iAffi VARCHAR) FOR
       SELECT Instructor_Name
       FROM Instructor
       WHERE Instructor_Affiliation = iAffi;
```

# CURSOR

- Start/initialize cursor

```
OPEN cursor2('Jedi');
```

# CURSOR

- Loop through cursor to process each row

```
FOR recordvar IN cursor2('Jedi') LOOP
    RAISE NOTICE 'The instructor is from %', recordvar
END LOOP;
```

# CURSOR

- Close cursor

```
CLOSE cursor2;
```

- Most RDBMS will automatically close cursor after the transaction block

# Cons of CURSOR

- Students with programming background like cursor, but ...
- SQL operation (relational operation) is much faster than cursor.
  - Most cursors can be replaced by Windows function now
  - Try to think like Relational DBMS
  - Cursor should be the last resort