



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

**Batch: D2                      Roll No.:**  
**16010122323**

**Experiment No.: 06**

**Grade: AA / AB / BB / BC / CC / CD / DD**

**Title: Queries based on Procedure, Function and Cursor**

**Objective:** To be able to functions and procedures on the table.

**Expected Outcome of Experiment:**

CO 3 : Use SQL for Relational database creation, maintenance and query

processing **Books/ Journals/ Websites referred:**

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g. Black book, Dreamtech Press 2. [www.db-book.com](http://www.db-book.com)
3. Korth, Silberchatz, Sudarshan : "Database Systems Concept", 5<sup>th</sup> Edition , McGraw Hill
4. Elmasri and Navathe, "Fundamentals of database Systems", 4<sup>th</sup> Edition, PEARSON Education.

**Resources used:** Postgresql

### **Theory**

#### **Procedures:**

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.



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

### 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.

Use of stored procedures can reduce network traffic between clients and servers, because the commands are executed as a single batch of code. This means only the call to execute the procedure is sent over a network, instead of every single line code being sent individually.

```
CREATE [ OR REPLACE ] PROCEDURE
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = }
default_expr ] [, ...] ] )
  { LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY
DEFINER
  | SET configuration_parameter { TO value | = value | FROM
CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
  } ...
```

Syntax:

Parameters

**Name:** The name (optionally schema-qualified) of the procedure to create.

**Argmode:** The mode of an argument: IN, INOUT, or VARIADIC. If omitted, the default is IN. (OUT arguments are currently not supported for procedures. Use INOUT instead.)

**Argname:** The name of an argument.

**Argtype:** The data type(s) of the procedure's arguments (optionally schema-qualified), if any. The argument types can be base, composite, or domain types, or can reference the type of a table column.

Depending on the implementation language it might also be allowed to specify “pseudo types” such as cstring. Pseudo-types indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

The type of a column is referenced by writing `table_name.column_name%TYPE`. Using this feature can sometimes help make a procedure independent of changes to the definition of a table.

**default\_expr:** An expression to be used as default value if the parameter is not specified. The expression has to be coercible to the argument type of the parameter. All input parameters following a parameter with a default value must have default values as well.

**lang\_name :** The name of the language that the procedure is implemented in. It can be `sql`, `c`, `internal`, or the name of a user-defined procedural language, e.g. `plpgsql`. Enclosing the name in single quotes is deprecated and requires matching case.

**TRANSFORM { FOR TYPE type\_name } [, ... ] }**

Lists which transforms a call to the procedure should apply. Transforms convert between SQL types and language-specific data types; see `CREATE TRANSFORM`. Procedural language implementations usually have hardcoded knowledge of the built-in types, so those don't need to be listed here. If a procedural language implementation does not know how to handle a type and no transform is supplied, it will fall back to a default behavior for converting data types, but this depends on the implementation.

**[EXTERNAL] SECURITY INVOKER**

**[EXTERNAL] SECURITY DEFINER**

**SECURITY INVOKER** indicates that the procedure is to be executed with the privileges of the user that calls it. That is the default. **SECURITY DEFINER** specifies that the procedure is to be executed with the privileges of the user that owns it.

The key word **EXTERNAL** is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all procedures not only external ones.

A **SECURITY DEFINER** procedure cannot execute transaction control statements (for example, `COMMIT` and `ROLLBACK`, depending on the language).

**configuration\_parameter**

**value:** The `SET` clause causes the specified configuration parameter to be set to the specified value when the procedure is entered, and then restored to its prior value when the procedure exits. `SET FROM CURRENT` saves the value of the parameter that is current when `CREATE PROCEDURE` is executed as the value to be applied when the procedure is entered.



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

If a SET clause is attached to a procedure, then the effects of a SET LOCAL command executed inside the procedure for the same variable are restricted to the procedure: the configuration parameter's prior value is still restored at procedure exit. However, an ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command: the effects of such a command will persist after procedure exit, unless the current transaction is rolled back.

If a SET clause is attached to a procedure, then that procedure cannot execute transaction control statements (for example, COMMIT and ROLLBACK, depending on the language).

**Definition:**

A string constant defining the procedure; the meaning depends on the language. It can be an internal procedure name, the path to an object file, an SQL command, or text in a procedural language. It is often helpful to use dollar quoting to write the procedure definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the procedure definition must be escaped by doubling them.

**obj\_file, link\_symbol**

This form of the AS clause is used for dynamically loadable C language procedures when the procedure name in the C language source code is not the same as the name of the SQL procedure. The string obj\_file is the name of the shared library file containing the compiled C procedure, and is interpreted as for the LOAD command. The string link\_symbol is the procedure's link symbol, that is, the name of the procedure in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL procedure being defined. When repeated CREATE PROCEDURE calls refer to the same object file, the file is only loaded once per session. To unload and reload the file (perhaps during development), start a new session.

**Example:**

We will use the following accounts table for the demonstration:

```
CREATE TABLE accounts (  
  id INT GENERATED BY DEFAULT AS IDENTITY,  
  name VARCHAR(100) NOT NULL,  
  balance DEC(15,2) NOT NULL,  
  PRIMARY KEY(id)  
);  
  
INSERT INTO accounts (name, balance)
```



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

```
VALUES ('Bob', 10000) ;
```

```
INSERT INTO accounts (name, balance)  
VALUES ('Alice', 10000) ;
```

The following example creates stored procedure named transfer that transfer specific amount of money from one account to another.

```
CREATE OR REPLACE PROCEDURE transfer (INT, INT, DEC)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
-- subtracting the amount from the sender's account  
UPDATE accounts  
SET balance = balance - $3  
WHERE id = $1;  
  
-- adding the amount to the receiver's account  
UPDATE accounts  
SET balance = balance + $3  
WHERE id = $2;  
COMMIT;  
END;  
$$;  
CALL stored_procedure_name(parameter_list);  
CALL transfer(1,2,1000);
```

## Functions

The basic syntax to create a function is as follows – CREATE

[OR REPLACE] FUNCTION function\_name (arguments)

RETURNS return\_datatype

language plpgsql

AS

\$variable\_name\$

declare

-- variable declaration

begin

-- stored procedure body

end; \$\$

**function-name** specifies the name of the function.



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

[OR REPLACE] option allows modifying an existing function.

The function must contain a return statement.

**RETURN** clause specifies that data type you are going to return from the function. The return\_datatype can be a base, composite, or domain type, or can reference the type of a table column.

**function-body** contains the executable part.

**The AS keyword** is used for creating a standalone function.

**plpgsql** is the name of the language that the function is implemented in. Here, we use this option for PostgreSQL, it Can be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name can be enclosed by single quotes.

Example

### **Cursors**

Rather than executing a whole query at once, it is possible to set up a cursor that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

Before a cursor can be used to retrieve rows, it must be opened. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

#### **OPEN FOR query**

Syntax: OPEN unbound\_cursorvar [ [ NO ] SCROLL ] FOR

query; example:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key =  
mykey;
```



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

### **OPEN FOR EXECUTE**

Syntax: OPEN unbound\_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE

query\_string [ USING expression [, ... ] ];

example:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' ||  
quote_ident(tabname) || ' WHERE col1 = $1' USING keyvalue;
```

### **Opening a Bound Cursor**

Syntax: OPEN bound\_cursorvar [ ( [ argument\_name := ] argument\_value [, ...] ) ];

Examples (these use the cursor declaration examples above):

```
OPEN curs2;
```

```
OPEN curs3(42);
```

```
OPEN curs3(key := 42);
```

Because variable substitution is done on a bound cursor's query, there are really two ways to pass values into the cursor: either with an explicit argument to OPEN, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN. For example, another way to get the same effect as the curs3 example above is

```
DECLARE
```

```
key integer;
```

```
curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

```
BEGIN
```

```
key := 42;
```

```
OPEN curs4;
```



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

## **Using Cursors**

### **FETCH**

Syntax: `FETCH [ direction { FROM | IN } ] cursor INTO target;`

Examples:

`FETCH curs1 INTO rowvar;`

`FETCH curs2 INTO foo, bar, baz;`

`FETCH LAST FROM curs3 INTO x, y;`

`FETCH RELATIVE -2 FROM curs4 INTO x;`

### **MOVE**

`MOVE [ direction { FROM | IN } ] cursor;`

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

Examples:

`MOVE curs1;`

`MOVE LAST FROM curs3;`

`MOVE RELATIVE -2 FROM curs4;`

`MOVE FORWARD 2 FROM curs4;`

### **UPDATE/DELETE WHERE CURRENT OF**

`UPDATE table SET ... WHERE CURRENT OF cursor;`

`DELETE FROM table WHERE CURRENT OF cursor;`

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. There are restrictions on what the cursor's query can be (in particular, no grouping) and it's best to use FOR UPDATE in the cursor. For more information see the DECLARE reference page.





**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

An example:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF  
curs1; CLOSE
```

**CLOSE** cursor;

**CLOSE** closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

```
CLOSE curs1;
```

**Implementation Screenshots (Problem Statement, Query and Screenshots of Results):**

**PROCEDURE:**

```
178 CREATE OR REPLACE PROCEDURE update_donation_history(IN donor_id INT, IN donation_amount DECIMAL)  
179 LANGUAGE SQL  
180 AS $$  
181 UPDATE DONOR  
182 SET DONATION_HISTORY = DONATION_HISTORY + donation_amount  
183 WHERE DONOR_ID = donor_id;  
184 $$;  
185  
186 CALL update_donation_history(1, 5000);
```


	donor_id [PK] integer	donor_name character varying (50)	passkey character varying (16)	donation_history integer	phone_number character varying (10)	email character varying (30)
1	1	Jiya	jiya123	50000	8282828282	jiya@gmail.com
2	2	Rahul	rahul456	200000	7878787878	rahul@gmail.com
3	3	Aisha	aisha789	100000	9090909091	aisha@gmail.com

	donor_id [PK] integer	donor_name character varying (50)	passkey character varying (16)	donation_history integer	phone_number character varying (10)	email character varying (30)
1	1	Jiya	jiya123	55000	8282828282	jiya@gmail.com
2	2	Rahul	rahul456	205000	7878787878	rahul@gmail.com
3	3	Aisha	aisha789	55000	9090909091	aisha@gmail.com



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

```
201      SELECT SUM(MONEY) INTO total_donation
202      FROM DONATES_TO
203      WHERE DONATES_TO.NGO_ID = ngoid;
204
205      RETURN total_donation;
206  END;
207  $$ LANGUAGE plpgsql;
208
209  SELECT total_donation_received(1);|
82  INSERT INTO DONATES_TO (DONOR_ID, NGO_ID, MONEY)
83  VALUES
84  (1, 2, 10000),
85  (2, 1, 5000),
86  (3, 1, 20000);
87
```

	total_donation_received numeric 
1	25000.00



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

```
226
227 LOOP
228     FETCH NEXT FROM donor_cursor INTO donor_id, donor_name, donation_history;
229
230     EXIT WHEN NOT FOUND;
231
232     RAISE NOTICE 'Donor ID: %, Donor Name: %, Donation History: %', donor_id, donor_name, donation_history;
233 END LOOP;
234
235 CLOSE donor_cursor;
236 END $$;
```

```
NOTICE: Donor ID: 1, Donor Name: JIYA, Donation History: 55000
NOTICE: Donor ID: 2, Donor Name: Rahul, Donation History: 205000
NOTICE: Donor ID: 3, Donor Name: Aisha, Donation History: 55000
DO
```

```
Query returned successfully in 30 msec.
```

**Conclusion:** Learnt how to implement queries based on Procedure, Function and Cursors.



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

**Post Lab Questions:**

**1. Does Storing Of Data In Stored Procedures Increase The Access Time? Explain?**

→ Storing data within stored procedures does not inherently increase access time. However, the way in which data is stored and accessed within stored procedures can impact performance.

**Here are some considerations:**

**Data Retrieval:** If a stored procedure needs to retrieve a large amount of data from the database, storing that data within the procedure itself could potentially increase access time. This is because the data needs to be fetched from the database and transferred into the procedure's memory space before it can be processed. In such cases, it might be more efficient to perform the data retrieval directly from the database when needed.

**Processing Logic:** Storing data within a stored procedure can improve performance if the data is small and can be processed efficiently within the procedure. For example, if a stored procedure needs to perform complex calculations or data manipulation on a small dataset, it might be more efficient to store that data within the procedure rather than repeatedly fetching it from the database.

**Caching:** Some database systems optimize stored procedure execution by caching the execution plan. If the same stored procedure is called multiple times with the same input parameters, the database might reuse the cached execution plan, which can improve performance.

**Resource Consumption:** Storing large datasets within stored procedures can consume memory and other system resources. This could potentially impact the overall performance of the database server, especially if multiple instances of the stored procedure are executed concurrently.

**Maintenance Overhead:** Storing data within stored procedures can make the code harder to maintain and update, especially if the data structures change frequently. It might be more flexible to store data in database tables and manipulate it using SQL queries, which can be easily modified without changing the stored procedures.

**2. Explain the FETCH statement in SQL cursors.**



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

→In SQL, a cursor is a database object that allows you to retrieve and manipulate the result set of a query one row at a time. The FETCH statement is used within SQL cursors to retrieve rows from the result set and make them available for processing.

**Syntax:**

The syntax of the FETCH statement varies slightly depending on the database system you're using, but the basic structure is as follows:

FETCH cursor\_name INTO variable1, variable2, ...;

- **cursor\_name:** The name of the cursor that was previously declared and opened. This cursor must have been opened using a DECLARE CURSOR statement.
- **variable1, variable2, ...:** Variables into which you want to fetch the column values of the current row. The number and data types of these variables should match the columns selected in the cursor's query.

**Functionality:**

The FETCH statement retrieves the next row from the result set associated with the cursor and assigns the column values of that row to the specified variables.

Each time you execute a FETCH statement, the cursor advances to the next row in the result set. After fetching the row, you can perform operations on the fetched data using the assigned variables.

**Types of Fetch:**

**Single Row Fetch:** Retrieves a single row from the result set. This is the most common type of fetch operation.

**Bulk Fetch:** Retrieves multiple rows from the result set at once, improving performance by reducing the number of round-trips between the application and the database.

**Handling Fetch Errors:**

If there are no more rows to fetch (i.e., the end of the result set is reached), the FETCH statement returns a special condition called SQLSTATE '02000' (or equivalent, depending on the database system).

You need to handle this condition appropriately in your code to avoid errors or unexpected behavior.



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

**Cursor Movement:**

The cursor maintains its position within the result set, advancing to the next row with each FETCH operation until it reaches the end of the result set.

You can control the cursor's position explicitly using other cursor-related statements like FETCH FIRST, FETCH LAST, FETCH PRIOR, FETCH ABSOLUTE, or FETCH RELATIVE.

**Closing Cursors:**

After fetching all the rows or when no longer needed, you should close the cursor using the CLOSE statement to release the associated resources.

**3. What is the difference between a function and a stored procedure in PostgreSQL?**

→ In PostgreSQL, both functions and stored procedures are database objects that contain a set of SQL statements and can be executed within the database. However, there are several key differences between functions and stored procedures:

**Return Type:**

**Function:** A function in PostgreSQL must return a value. It can return a single value or a set of values (in the case of set-returning functions).

**Stored Procedure:** A stored procedure does not necessarily return a value. It may execute a series of SQL statements without returning any result.

**Invocation:**

**Function:** Functions can be invoked directly within SQL statements, such as SELECT queries, and their return values can be used as part of an expression.

**Stored Procedure:** Stored procedures are typically invoked using the CALL statement or by referencing them in procedural code blocks, such as PL/pgSQL.

**Transaction Control:**

**Function:** Functions execute within the context of the current transaction. They can participate in transaction control statements (COMMIT, ROLLBACK, etc.) and are subject to transaction semantics.

**Stored Procedure:** Stored procedures can contain transaction control statements and execute within the context of the current transaction, but they do not inherently have transaction semantics. They can be explicitly designed to commit or rollback transactions, but they do not automatically participate in the surrounding transaction.



**K. J. Somaiya College of Engineering, Mumbai-77**  
(A Constituent College of Somaiya Vidyavihar University)  
**Department of Computer Engineering**

**Data Manipulation:**

**Function:** Functions can manipulate data like regular SQL queries. They can perform data retrieval, manipulation, and modification within the database.

**Stored Procedure:** Stored procedures can also manipulate data, but they are more commonly used for procedural tasks, such as complex business logic, application flow control, or encapsulating reusable code.

**Schema:**

**Function:** Functions are typically associated with a specific schema in the database and can be organized within that schema.

**Stored Procedure:** Stored procedures can also be organized within a schema, but they are not inherently tied to any particular schema.

**Language Support:**

**Function:** Functions in PostgreSQL can be implemented using various languages supported by PostgreSQL, such as SQL, PL/pgSQL (Procedural Language/PostgreSQL), PL/Python, PL/Perl, etc.

**Stored Procedure:** Stored procedures are typically implemented using PL/pgSQL, which is PostgreSQL's procedural language, although they can also be written in other languages supported by PostgreSQL.