CS2102 Database Systems

Slides adapted from Prof. Chan Chee Yong

LECTURE 07A

STORED PROCEDURE, FUNCTIONS, AND TRIGGERS

PREVIOUS LECTURE

SQL

- Conceptual evaluation of queries
  - Query:

```
SELECT      DISTINCT select-list
FROM        from-list
WHERE       where-condition
GROUP BY    groupby-list
HAVING      having-condition
ORDER BY    orderby-list
LIMIT       limit-spec
OFFSET      offset-spec
```

1. Compute the cross-product of the tables in **`from-list`**
2. Select the tuples in the cross-product that evaluate to TRUE for the **`where-condition`**
3. Partition the selected tuples into groups using the **`groupby-list`**
4. Select the groups that evaluate to TRUE for the **`having-condition`** condition
5. For each selected group, generate an output tuple by selecting/computing the attributes/expressions that appear in the **`select-list`**
6. Remove any duplicate output tuples because of DISTINCT
7. Sort the output tuples based on the **`orderby-list`**
8. Remove the appropriate output tuples based on the **`limit-spec`** & **`offset-spec`**

PREVIOUS LECTURE

Summary

- A subquery is an SQL query that may be added into
  - select clause
    - Scalar subqueries
    - One column and one tuple
  - from clause
    - Common table expression
  - where clause
    - `EXISTS`; `IN`; `ANY/SOME`; `ALL`

- Transaction
- Stored procedures and functions
  - Introduction
  - Creation
  - Usage
  - plpgsql
- Triggers
  - Creation
  - Learn by example

Overview

- Transaction
- Stored procedures and functions
  - Introduction
  - Creation
  - Usage
  - plpgsql
- Triggers
  - Creation
  - Learn by example

Transaction

Transaction

Introduction

- A transaction consists of *one or more update/retrieval operations* (i.e., SQL statements)
  - An abstraction of a logical unit of work

Syntax

- `BEGIN code { COMMIT | ROLLBACK }`
- The `BEGIN` command starts a new transaction
- Each transaction must end with either
  - `COMMIT` **success** → *update database*
  - `ROLLBACK` **failure** → *restore database to state before* `BEGIN`

Transaction

Example

- Transfer $1000 from `aid` = 1 to `aid` = 2
  - What happens when error occurred?

```
BEGIN
UPDATE accounts
SET balance = balance
+ 1000
WHERE accountID = 2;

UPDATE accounts
SET balance = balance
– 1000
WHERE accountID = 1;
COMMIT
```

VS

```
UPDATE accounts
SET balance = balance
+ 1000
WHERE accountID = 2;

UPDATE accounts
SET balance = balance
– 1000
WHERE accountID = 1;
```

Transaction

ACID properties

- **Atomicity** — Either all the effects of the transactions are reflected in the database or none are
  all or none
- **Consistency** — The execution of a transaction in isolation preserves the consistency of the database
  user-defined property should be preserved
- **Isolation** — The execution of a transaction is isolated from the effects of other concurrent transaction execution
  can run concurrently
- **Durability** — The effect of a committed transaction persists in the database even in the presence of system failures
  commit is permanent

Transaction

Constraint check

- By default constraints are checked immediately at the end of SQL statement execution
  - A violation will cause the statement to be rollbacked
- Can be deferred to the end of transaction execution
  - A violation will cause the transaction to be aborted
- The type of constraint checking should be specified as part of constraint declaration

```
CREATE TABLE students (
  name    varchar(100),
  CHECK (name IS NOT NULL)
      DEFERRABLE INITIALLY DEFERRED
);
```

Transaction

Circular reference?

- Given `T1 (a, b)` and `T2 (a, b)` such that
  - `T1.a` is a foreign key that refers to `T2.a`
  - `T2.a` is a foreign key that refers to `T1.a`
- How?
  - ```
    ALTER TABLE T1 ADD FOREIGN KEY (a)
    REFERENCES T2 (a) DEFERRABLE INITIALLY DEFERRED;
    ```
  - ```
    ALTER TABLE T2 ADD FOREIGN KEY (a)
    REFERENCES T1 (a) DEFERRABLE INITIALLY DEFERRED;
    ```
  - Transaction!

    ```
    BEGIN;
    INSERT INTO T1 VALUES (1,2);
    INSERT INTO T2 VALUES (1,2);
    COMMIT;
    ```

- Transaction
- Stored procedures and functions
  - Introduction
  - Creation
  - Usage
  - plpgsql
- Triggers
  - Creation
  - Learn by example

Stored procedures and function

SELF-READ

History

History of stored procedures

- SQL 1999 standard proposes a language for stored procedures and triggers
- PostgreSQL (*and other DBMS*) can store, share, and execute code on the server
- PL/pgSQL is PostgreSQL's language specifically designed to seamlessly embed SQL code and interact with the database SQL code
- PL/pgSQL partially complies with the SQL standard
- PL/pgSQL is similar to Oracle PL/SQL, SQLServer Transact-SQL, and DB2 SQL Procedural Language as well as other languages implementing variants of the Persistent Storage Modules portion of the SQL standard

SELF-READ

History

Performance

- Stored procedures are compiled. The code is cached and shared by all users. The code is executed on the server's side (*typically a powerful machine*). They generally incur fewer data transfer across the network.
  - *What it really says: The optimization of the SQL code they contain is not adaptive. The client's computing power is underutilized.*

Productivity

- Stored procedures are shared and reused under access control. The application logic they encode is implemented and maintained in a single place.
  - *What it really says: The languages and concepts are complicated. The code is not portable from one DBMS to the next.*

Security:

- Data manipulation can be restricted to calling stored procedures under strict access control.
  - *What it really says: It is easy to make tragic mistake in the coding*

Introduction

What?

- Stored procedure is a procedure is a procedure / function / subroutine that is available to application that access a DBMS and is stored in the database
  - Includes typical control structures such as IF and WHILE

Why?

- Uses includes (*but not limited to*) data validation, access control mechanism, consolidation of logic, consistency check, etc

How?

- Create function syntax
- Function definition

Creation

Create function syntax

- `CREATE [ OR REPLACE ] FUNCTION`
  `func_name ( [ arg1 type1 [, arg2 type2 [ ... ] ] ] )`
  `RETURNS ret_type AS func_def`
  `LANGUAGE lang;`

Create procedure syntax

- `CREATE [ OR REPLACE ] PROCEDURE`
  `proc_name ( [ arg1 type1 [, arg2 type2 [ ... ] ] ] )`
  `AS proc_def`
  `LANGUAGE lang;`

Removal

- `DROP { FUNCTION | PROCEDURE } [ IF EXISTS ] name;`

❖ Reference: https://www.postgresql.org/docs/11/sql-createfunction.html

Creation

Value conversion

- CREATE OR REPLACE FUNCTION
  miles_to_km (mile numeric)
  RETURNS numeric AS
  ‘ BEGIN RETURN mile * 1.609; END; ’
  LANGUAGE plpgsql;

Insertion

- CREATE OR REPLACE PROCEDURE
  add_restaurant (name varchar(50),
  area varchar(10)) AS
  ‘ BEGIN
  INSERT INTO Restaurants VALUES (name, area);
  END; ’
  LANGUAGE plpgsql;

Creation

Dollar-quoted string

- Problem

```
CREATE OR REPLACE FUNCTION hello_world ()
RETURNS CHAR(11) AS
‘ BEGIN RETURN ‘Hello World’; END; ’
LANGUAGE plpgsql;
```

- Solution

```
CREATE OR REPLACE FUNCTION hello_world ()
RETURNS CHAR(11) AS
      BEGIN RETURN ‘Hello World’; END;
LANGUAGE plpgsql;
```

Usage

Stored functions

- Can be used in an expression
  - Given Shops (name, distance)

```
SELECT S.name, miles_to_km(S.distance)
FROM   Shops S
WHERE  miles_to_km(S.distance) < 10;
```

Stored procedure

- Can be called

```
CALL add_restaurant('Domino''s', 'West');
```

plpgsql

What can we write?

- Declaration (before BEGIN)
- Assignment
- Selection
  - IF cond THEN stmt;
    [ELSIF cond THEN stmt [...]]
    [ELSE stmt];
- Iteration
  - WHILE cond LOOP stmt; END LOOP;
  - FOR var IN (expr ...) LOOP
    { stmt; | EXIT; | EXIT WHEN cond; } [...]
    END LOOP;
  - LOOP { stmt; | EXIT; | EXIT WHEN cond; } [...]
    END LOOP;

```
DECLARE var type;
var := expr;
```

plpgsql

Operations

- Arithmetic and bitwise
  - **Simple** `+`, `-`, `*`, `/` (integer division truncates), `%`, `^` (exponent)
  - **Bitwise** `&`, `|`, `#` (xor), `~` (not), `<<`, `>>`
  - **Others** `|/` (square root), `||/` (cube root), `@` (absolute value), `!` (factorial postfix), `!!` (factorial prefix)
    - References: https://www.postgresql.org/docs/11/functions-math.html
- Comparison
  - **Simple** `<`, `>`, `<=`, `>=`, `=`, `<>`
    - References: https://www.postgresql.org/docs/11/functions-comparison.html

plpgsql

Example

- Given two points (x1, y1) and (x2, y2), write a function to return the distance between the two points

```
CREATE OR REPLACE FUNCTION
dist (x1 numeric, y1 numeric,
      x2 numeric, y2 numeric) RETURNS numeric AS
$$ DECLARE distx numeric; disty numeric;
BEGIN
  distx := x1 - x2; disty := y1 - y2;
  RETURN |/ ((distx * distx) + (disty * disty));
END; $$ LANGUAGE plpgsql;
```

- Transaction
- Stored procedures and functions
  - Introduction
  - Creation
  - Usage
  - plpgsql
- Triggers
  - Creation
  - Learn by example

Triggers

SELF-READ

History

History of triggers

- A trigger is a procedure or function that is executed when a database event occurs on a table
  - Example: `INSERT`, `DELETE`, `UPDATE`, `CREATE TABLE`, etc
- Triggers are used to maintain integrity, propagate updates, and repair the database
- Their syntax and semantics vary from one DBMS to another

SELF-READ

History

Performance

- Triggers are compiled. The code is cached and shared by all users. The code is executed on the server's side (*typically a powerful machine*). They generally incur no data transfer across the network.
  - *What it really says: The optimization of the SQL code they contain is not adaptive. The client's computing power is underutilized.*

Productivity

- Triggers are applied to all interactions. The application logic they encode is implemented and maintained in a single place.
  - *What it really says: The languages and concepts are complicated. The code is not portable. Interactions among triggers (chain reactions) and between triggers, constraints, and transactions are difficult to control.*

Security:

- Data manipulation can be restricted to calling stored procedures under strict access control.
  - *What it really says: It is easy to make tragic mistake in the coding*

Introduction

What?

- Trigger is a procedure is a special stored procedure that is executed when a specific event occurs in the database
  - Triggers may occur before or after the event

Why?

- Uses includes (*but not limited to*) data validation, access control mechanism, consolidation of logic, consistency check, etc

How?

- Trigger syntax
- Learn by example

Creation

Create trigger syntax

- CREATE TRIGGER **trigger_name**
  { BEFORE | AFTER | INSTEAD OF }
  { **event** [OR **event** [...]] } ON **table**
  [FOR [EACH] { ROW | STATEMENT }]
  [WHEN **cond**] EXECUTE PROCEDURE **func_name**();

Special variables

- NEW (for row only), OLD (for row only)
- TG_WHEN (returns 'BEFORE', 'AFTER')
- TG_OP (returns 'INSERT', 'UPDATE', 'DELETE', 'TRUNCATE')

Removal

- DROP TRIGGER [IF EXISTS] **trigger_name**
  ON **table** [CASCADE];

Learn by example

Preliminary

- Given Games (name, price)
  - CREATE TABLE Games (
    name varvhar(50) PRIMARY KEY,
    price numeric NOT NULL CHECK (price > 0)
    );

Games

name	*price*
A	100
B	200
C	300

Learn by example

For each row

- ```
  CREATE OR REPLACE FUNCTION t_func4()
  RETURNS TRIGGER AS $$ BEGIN
  RAISE NOTICE 'Trigger 4'; RETURN NULL;
  END; $$ LANGUAGE plpgsql;
  ```
- ```
  CREATE TRIGGER trig4
  BEFORE INSERT OR UPDATE ON Games
  FOR EACH ROW WHEN (NEW.price > 100)
  EXECUTE PROCEDURE t_func4();
  ```

- Behavior
  - Message `'Trigger 4'` is displayed on psql whenever a row is inserted/updated
  - Row is NOT inserted when `price > 100`

Learn by example

For each row

- ```
  CREATE OR REPLACE FUNCTION t_func3()
  RETURNS TRIGGER AS $$ BEGIN
  RAISE NOTICE 'Trigger 3'; RETURN NEW;
  END; $$ LANGUAGE plpgsql;
  ```
- ```
  CREATE TRIGGER trig3
  BEFORE INSERT OR UPDATE ON Games
  FOR EACH ROW WHEN (NEW.price > 100)
  EXECUTE PROCEDURE t_func3();
  ```

- Behavior
  - Message `'Trigger 3'` is displayed on psql whenever a row is inserted/updated
  - Row is STILL inserted when `price > 100`

Learn by example

For each row

- ```
  CREATE OR REPLACE FUNCTION t_func2()
  RETURNS TRIGGER AS $$ BEGIN
  RETURN (NEW.name, OLD.price * 2);
  END; $$ LANGUAGE plpgsql;
  ```
- ```
  CREATE TRIGGER trig2
  BEFORE UPDATE ON Games
  FOR EACH ROW WHEN (NEW.price <= OLD.price)
  EXECUTE PROCEDURE t_func2();
  ```

- Behavior
  - When the new price is lower than old price
  - Row is inserted such that the old price is doubled

Learn by example

For each statement

- ```
  CREATE OR REPLACE FUNCTION t_func1()
  RETURNS TRIGGER AS $$ BEGIN
  RAISE NOTICE 'Trigger 1'; RETURN NULL;
  END; $$ LANGUAGE plpgsql;
  ```
- ```
  CREATE TRIGGER trig1
  BEFORE INSERT OR UPDATE ON Games
  FOR EACH STATEMENT
  EXECUTE PROCEDURE t_func1();
  ```

- Behavior
  - Message `'Trigger 1'` is displayed on psql once every statement
  - Values is STILL inserted
    - Reference: https://www.postgresql.org/docs/current/plpgsql-trigger.html

Multiple triggers

Trigger interactions

- Triggers interact with other triggers
  - Interaction happens in alphabetical order
- Example
  - Create `<t_func4(), trig4>` on `Games`
  - Update '`A`' to $75
  - Create `<t_func2(), trig2>` on `Games`
  - Update '`A`' to $50

❖ What happened here?

Summary

- A transaction starts with `BEGINS` ends with either `COMMIT` or `ROLLBACK`
  - We assume ACID property is maintained
- Stored function
  - `CREATE [ OR REPLACE ] FUNCTION func_name ...`
  - `SELECT func_name (...);`
- Stored procedure
  - `CREATE [ OR REPLACE ] PROCEDURE proc_name ...`
  - `CALL proc_name (...);`
- Triggers
  - `CREATE TRIGGER trigger_name`

```
{ BEFORE | AFTER | INSTEAD OF }
{ event [ OR event [ ... ] ] } ON table
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN cond ] EXECUTE PROCEDURE func_name();
```