PL/pgSQL
Procedural programming language in PostgreSQL

Jeffry Hartanto
jhartanto@comp.nus.edu.sg

28 February 2022

Outline

1. Quick Recap on SQL
2. Motivation
3. Host language + SQL
4. PL/pgSQL Part I (mini break)
5. PL/pgSQL Part II (mini break)
6. SQL Injection

01 Quick Recap on SQL

01 Quick Recap on SQL

- So far, we have learnt …

SQL 1

DDL

ALTER

CREATE

DROP

CONSTRAINT

DML

INSERT

SELECT

UPDATE

DELETE

SQL 2

LIMIT

ORDER BY

NATURAL JOIN

LEFT/RIGHT JOIN

UNION

FULL JOIN

EXCEPT

INTERSECT

EXISTS

IN

ANY

SCALAR SUBQUERY

SQL 3

GROUP BY

HAVING

MAX

AVG

MIN

CASE

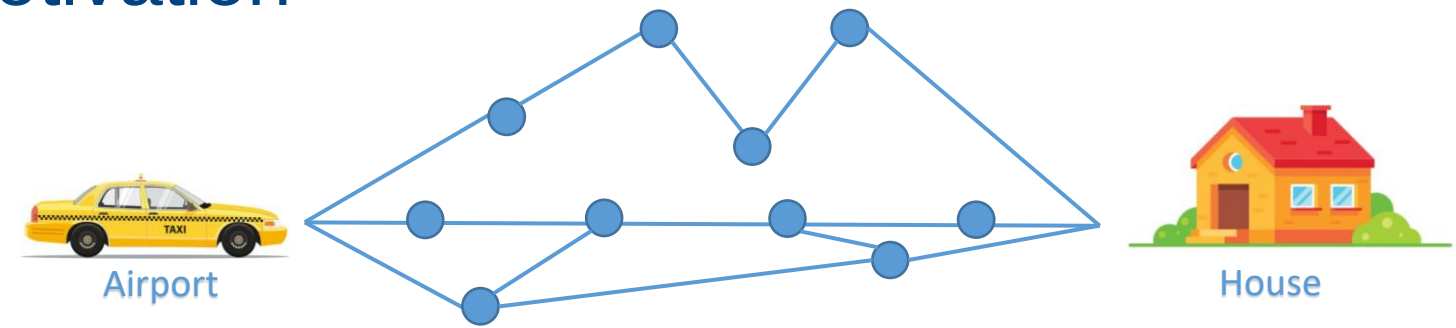
COALESCE

CTE

VIEWS

02 Motivation

02 Motivation



Declarative vs. Procedural

- Declarative specifies the "**what**", whereas Procedural specifies the "**how**".
- Declarative tends to require less lines of codes for solving a *generic* query as compared to Procedural. E.g., find a student with highest grade.
- Declarative may require a complex solution for solving a very *specific* query as compared to Procedural. E.g., <next slide>.

02 Motivation

Based on this ranking system of cryptocurrencies,
I want to have daily report of *first three coins* that are
down by more than 5% and are within 2 ranks apart.

We will do it!

Rank	Symbol	Changes
1	BTC	-6%
2	ETH	+3%
3	DOGE	-6%
4	ZIL	+10%
5	XMR	-1%
6	SHIB	-8%
7	ADA	+1%
8	LTC	-7%
9	XRP	-7%
10	BNB	-6%

Rank	Symbol	Changes
6	SHIB	-8%
8	LTC	-7%
9	XRP	-7%

Possible to use SQL?

Any easier way?

02 Motivation

Based on this ranking system of cryptocurrencies,
I want to have daily report of *first three coins* that are
down by more than 5% and are within 2 ranks apart.

We will do it!

Generally, it is *easier* to use a *procedural language* for problems that require *very specific* traversal of the data.

Two possible solutions:

3. Host language + SQL
(Java, C, Python, etc.)

4. PL/pgSQL

03 Host language + SQL

(***not*** used in the project.)

03 Host language + SQL

- Let's use C language as an example.
- There are two types of mixing:

Statement-level Interface

C
+ SQL

Call-level Interface

C
only

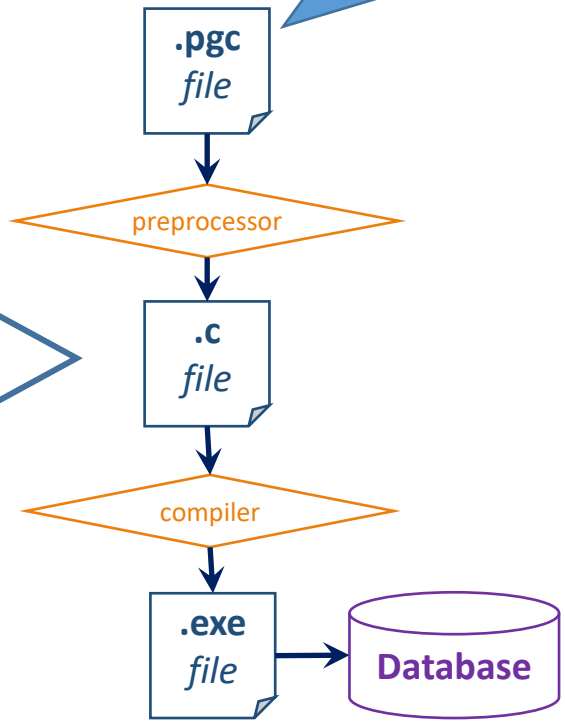
03 Statement-level Interface

C
+ SQL

How does **.pgc** file look like?

Basic idea

1. **Write** a program that mixes host language with SQL.
2. **Preprocess** the program using a preprocessor.
3. **Compile** the program into an executable code.



03 Statement-level Interface

C
+ SQL

"Scores"

Name	Mark
Alice	92
...	...

.pgc
file

```
void main() {
    EXEC SQL BEGIN DECLARE SECTION;
      char name[30]; int mark;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT @localhost USER john;
    // some code that assigns values to
    // name and mark.
    EXEC SQL INSERT INTO
      Scores (Name, Mark) VALUES (:name, :mark);
    EXEC SQL DISCONNECT;
}
```

Declaration

Connection

Host language

Query execution

Disconnect

The SQL query above is fixed, i.e., **static SQL**.
Can we generate the SQL query during runtime?
Yes, it is called **Dynamic SQL**.

03 Statement-level Interface

C
+ SQL

"Scores"

Name	Mark
Alice	92
...	...

.pgc
file

```
void main() {
    EXEC SQL BEGIN DECLARE SECTION;
      char *query; char name[30]; int mark;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT @localhost USER john;
    // some code that assigns values to
    // name and mark

    // assign any SQL statement to the query,
    // the query may include name and/or mark.
    EXEC SQL EXECUTE IMMEDIATE :query;
    EXEC SQL DISCONNECT;
}
```

Declaration

Connection

Host language

Query execution

Disconnect

What if we want to use C only?

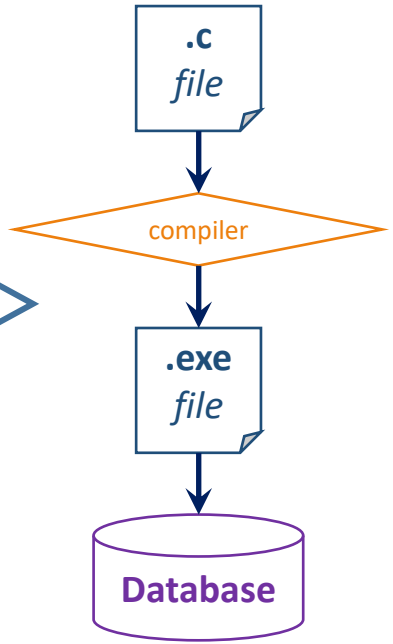
03 Call-level Interface

C *only*

Basic idea

1. **Write** in host language only*.
2. **Compile** the program into an executable code.

*Need to load a library that provides APIs to access the DB, e.g., libpq, psqlODBC, JDBC, ODBC, etc.



03 Call-level Interface

C *only*

"Scores"

Name	Mark
Alice	92
...	...

.c *file*

```
void main() {
```

```
char *query; char name[30]; int mark;
```

Declaration

```
connection C("dbname = testdb user = postgres \
  password = test hostaddr = 127.0.0.1 \
  port = 5432");
```

Connection

```
// assign any SQL statement to the query,
// the query may include name and/or mark.

work W(C);
W.exec(query);
W.commit();
```

Query execution

```
C.disconnect();
```

Disconnect

```
}
```

Flash Quiz

Is this a static or dynamic SQL?

03 Summary

- Statement-level Interface

C
+ SQL

  - Code is written in a mix of host language and SQL.
    - Static SQL has fixed queries.
    - Dynamic SQL generates queries at runtime.
  - Code is pre-processed before compiled into an executable program.

- Call-level Interface

C
only

  - Code is written in host language only.
    - Need a library that provides APIs to run the SQL queries.
  - Code is directly compiled into an executable program.

What if we want to use SQL only?

04 PL/pgSQL Part I

04 PL/pgSQL

SQL *only*

- SQL-based Procedural Language for PostgreSQL
  - Server-side Programming
  - ISO standard: SQL/PSM (Persistent Stored Modules).
  - It standardizes syntax and semantics of SQL Procedural Language.
  - Different vendors have different implementations:
    - Oracle `PL/SQL`
    - SQL Server `TransactSQL`

Let's learn a new programming language!

04 PL/pgSQL

SQL *only*

- Why do we want to use this?
  - Code reuse.
  - Ease of maintenance.
  - Performance.
  - Security (will be discussed near the end).

Before that, let's first learn about …

Functions and Procedures

04 Functions

SQL *only*

Converts students' marks to grades.

"Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Grading scheme:

[70, 100] → A
[60, 70) → B
[50, 60) → C
[0, 50) → F

"Scores"

Name	Grade
Alice	A
Bob	B
Cathy	C
David	F

```
SELECT Name, CASE
    WHEN Mark >= 70 THEN 'A'
    WHEN Mark >= 60 THEN 'B'
    WHEN Mark >= 50 THEN 'C'
    ELSE 'F' END AS Grade
FROM Scores;
```

Can we abstract away the conversion with a function?

04 Functions

SQL *only*

General Syntax

```
CREATE OR REPLACE FUNCTION <name>
  (<param> <type>, <param> <type>, ...)
RETURNS <type> AS $$

  <code goes here>

$$ LANGUAGE sql;
```

Function name

Function parameters and type, if any

Return type

Enclosed within dollar-quote or '

Main body of the function

The language used by the code

```
<type>:
- all data types in SQL.
- a tuple/a set of tuples.
- custom tuples.
- triggers.
- etc.
```

```
CREATE OR REPLACE FUNCTION convert(Mark INT)
RETURNS CHAR(1) AS $$
  SELECT CASE
      WHEN Mark >= 70 THEN 'A'
      WHEN Mark >= 60 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
  END;
$$ LANGUAGE sql;
```

```
-- Call the function
SELECT convert(66);
SELECT * FROM convert(66);
```

Flash Quiz: How to use this for all records in "Scores"?

```
SELECT … FROM Scores;
```

04 Functions

SQL *only*

"Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

```
CREATE OR REPLACE FUNCTION convert(Mark INT)
RETURNS CHAR(1) AS $$
  SELECT CASE
      WHEN Mark >= 70 THEN 'A'
      WHEN Mark >= 60 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F' END;
$$ LANGUAGE sql;

SELECT Name, convert(mark) AS Grade FROM Scores;
```

"Scores"

Name	Grade
Alice	A
Bob	B
Cathy	C
David	F

Why do we want to use this?

- Code reuse.
- Ease of maintenance.
- Performance.

```
SELECT Name, convert(Mark) FROM Scores;
SELECT Name
FROM Scores WHERE convert(Mark) = 'B';
```

compiled

```
CREATE OR REPLACE FUNCTION convert
  (Mark INT)
RETURNS CHAR(1) AS $$
  SELECT CASE
      WHEN Mark >= 75 THEN 'A'
      WHEN Mark >= 65 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

Let's learn more about functions.

04

Functions

"Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

```
CREATE OR REPLACE FUNCTION convert
  (Mark INT)
RETURNS CHAR(1) AS $$
  SELECT CASE
      WHEN Mark >= 75 THEN 'A'
      WHEN Mark >= 65 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

SQL *only*

How to return a tuple?

```
CREATE OR REPLACE FUNCTION GradeStudent
  (Grade CHAR(1))
RETURNS Scores AS $$

  SELECT    *
  FROM      Scores
  WHERE     convert(Mark) = Grade;

$$ LANGUAGE sql;
```

Flash Quiz: What is the output of this SQL query?

```
SELECT GradeStudent('C');
```

How to return more than one tuple?

```
CREATE OR REPLACE FUNCTION GradeStudents
  (Grade CHAR(1))
RETURNS SETOF Scores AS $$
  …
$$ LANGUAGE sql;
```

How to return a custom tuple?

04 Functions

"Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

```
CREATE OR REPLACE FUNCTION convert
  (Mark INT)
RETURNS CHAR(1) AS $$
  SELECT CASE
      WHEN Mark >= 75 THEN 'A'
      WHEN Mark >= 65 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

SQL *only*

Default

```
CREATE OR REPLACE FUNCTION CountGradeStudents
  (IN Grade CHAR(1),
   OUT Grade CHAR(1),
   OUT Count INT)
RETURNS RECORD AS $$

  SELECT      Grade, COUNT(*)
  FROM        Scores
  WHERE       convert(Mark) = Grade;


$$ LANGUAGE sql;
```

Flash Quiz: What is the output of this SQL query?

```
SELECT CountGradeStudents('C');
```

How to return a set of custom tuples?

```
CREATE OR REPLACE FUNCTION CountGradeStudents
  (…)
RETURNS SETOF RECORD AS $$
  …
$$ LANGUAGE sql;
```

Can we simplify the params for custom tuples? Yes!

04 Functions

"Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

```
CREATE OR REPLACE FUNCTION convert
  (Mark INT)
RETURNS CHAR(1) AS $$
  SELECT CASE
      WHEN Mark >= 75 THEN 'A'
      WHEN Mark >= 65 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

SQL *only*

Can we simplify the params for custom tuples? Yes!

```
CREATE OR REPLACE FUNCTION CountGradeStudents()
RETURNS TABLE(MARK CHAR(1), COUNT INT) AS $$

  SELECT      convert(Mark), COUNT(*)
  FROM        Scores
  GROUP BY    convert(Mark);

$$ LANGUAGE sql;
```

```
SELECT CountGradeStudents();
```

Can a function returns "nothing"?

```
CREATE OR REPLACE FUNCTION AddGradeAttr()
RETURNS VOID AS $$
  ALTER TABLE Scores ADD COLUMN IF NOT EXISTS
    Grade CHAR(1) DEFAULT NULL;
  UPDATE Scores SET Grade = convert(Mark);
  SELECT * FROM Scores;

$$ LANGUAGE sql;
```

Throws an error because Grade is unidentified.

```
SELECT AddGradeAttr();
```

Can't we use procedure for this? Yes!

04 Procedures

SQL *only*

General Syntax

Procedure name

Procedure parameters and type, if any

```
CREATE OR REPLACE PROCEDURE <name>
  (<param> <type>, <param> <type>, ...)
AS $$

  <code goes here>

$$ LANGUAGE sql;
```

Main body of the function

Enclosed within dollar-quote or '

The language used by the code

<type>:
- all data types in SQL.
- a tuple/a set of tuples.
- custom tuples.
- triggers.
- etc.

```
CREATE OR REPLACE PROCEDURE AddGradeAttr()
AS $$

  ALTER TABLE Scores ADD COLUMN IF NOT EXISTS
    Grade CHAR(1) DEFAULT NULL;
  SELECT * FROM Scores;

$$ LANGUAGE sql;
```

```
CALL AddGradeAttr();
```

04

Summary

SQL *only*

- SQL Functions
  - Returns a value.
    - SQL data types, a tuple, set of custom tuples, etc.
  - `CREATE OR REPLACE FUNCTION <function_name>(...)`
  - `SELECT <function_name>(...) or SELECT * FROM <function_name>(...).`
- SQL Procedures
  - No return value.
  - `CREATE OR REPLACE PROCEDURE <function_name>(...).`
  - `CALL <function_name>(...).`

05 PL/pgSQL Part II

```
CREATE OR REPLACE FUNCTION <name>
  (<param> <type>, ...)
RETURNS <type> AS $$
  <code goes here>
$$ LANGUAGE sql;
```

```
CREATE OR REPLACE FUNCTION <name>
  (<param> <type>, ...)
RETURNS <type> AS $$
DECLARE
    <variables go here>
BEGIN
    <code goes here>
END;
$$ LANGUAGE plpgsql;
```

05

PL/pgSQL Part II

SQL *only*

- Previous functions or procedures are limited to executing one or more SQL queries sequentially.
- PL/pgSQL is more powerful than that as it has variables and control structure.
- List of control structure:
  - `IF ... ELSIF ... THEN ... ELSE ... END IF`
  - `EXIT ... WHEN ...`
  - `LOOP ... END LOOP`
  - `WHILE ... LOOP ... END LOOP`
  - `FOR ... IN ... LOOP ... END LOOP`

Let's start with Variables and Control Structure.

05 Variables

"Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

SQL *only*

```
CREATE OR REPLACE FUNCTION splitMarks
  (IN name1 VARCHAR(20), IN name2 VARCHAR(20),
  OUT mark1 INT, OUT mark2 INT)
RETURNS RECORD AS $$
DECLARE
  temp INT := 0;
BEGIN
  SELECT mark INTO mark1 FROM Scores
  WHERE name = name1;
  SELECT mark INTO mark2 FROM Scores
  WHERE name = name2;

  temp := (mark1 + mark2) / 2;

  UPDATE Scores SET mark = temp
  WHERE name = name1 OR name = name2;
  RETURN; --optional
END;
$$ LANGUAGE plpgsql;
```

```
SELECT splitMarks('Alice', 'Bob');
```

How to return a set of custom tuples?

```
CREATE OR REPLACE FUNCTION splitMarks
  (IN name1 VARCHAR(20), IN name2 VARCHAR(20))
RETURNS TABLE(mark1 INT, mark2 INT) AS $$
DECLARE
  temp INT := 0;
BEGIN

  --code is omitted

  RETURN QUERY SELECT mark1, mark2;
  RETURN NEXT;
END;
$$ LANGUAGE plpgsql;
```

! RETURN NEXT/QUERY does **not** exit the function

05 Control Structure

"Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

SQL *only*

```
CREATE OR REPLACE FUNCTION splitMarks
  (IN name1 VARCHAR(20), IN name2 VARCHAR(20))
RETURNS TABLE(Mark1 INT, Mark2 INT) AS $$
DECLARE
  temp INT := 0;
BEGIN
  SELECT mark INTO mark1 FROM Scores
  WHERE name = name1;
  SELECT mark INTO mark2 FROM Scores
  WHERE name = name2;

  temp := (mark1 + mark2) / 2;

  <control structure code>

  UPDATE Scores SET mark = temp
  WHERE name = name1 OR name = name2;
  RETURN QUERY SELECT mark1, mark2;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT splitMarks('Alice', 'Bob');
```

```
IF temp > 60 THEN      temp := temp / 2;
ELSIF temp > 50 THEN   temp := temp - 20;
ELSE                   temp := temp - 10;
END IF;
```

```
WHILE temp > 30 LOOP
    temp := temp / 2;
END LOOP;
```

```
LOOP
    EXIT WHEN temp < 30;
    temp := temp / 2;
END LOOP;
```

in Imperative Language ...

```
while (true) {
  if (temp < 30)
    break;
}
```

```
FOREACH d IN ARRAY denoms LOOP
    temp := temp / d;
END LOOP;
```

Declaration

```
d INT;
denoms INT[] :=
ARRAY[1, 2, 3];

*index starts at 1.
```

05 Is that all?

SQL *only*

Based on this ranking system of cryptocurrencies,
I want to have daily report of *first three coins* that are
down by more than 5% and are within 2 ranks apart.

We will do it!

Rank	Symbol	Changes
1	BTC	-6%
...	...	...
...	...	...
8	LTC	-7%
9	XRP	-7%
10	BNB	-6%

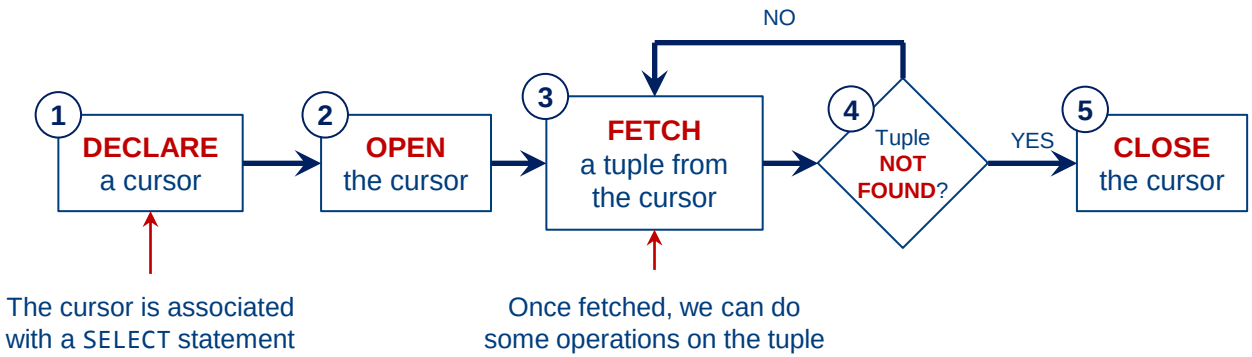
Rank	Symbol	Changes
6	SHIB	-8%
8	LTC	-7%
9	XRP	-7%

How do we traverse a query's result?

Cursor

SQL *only*

- A cursor enables us to access each individual row returned by a SELECT statement
- Workflow:



- Can use other statements at step 3 such as `MOVE, UPDATE, DELETE, etc`.

05 Cursor

SQL *only*

Start simple

Based on this ranking system of cryptocurrencies,
I want to have daily report of *first three consecutive coins* that are *down by more than 5%*.

Rank	Symbol	Changes
1	BTC	-6%
2	ETH	+3%
3	DOGE	-6%
4	ZIL	-7%
5	XMR	-8%
6	SHIB	-8%
7	ADA	+1%
8	LTC	-7%
9	XRP	-7%
10	BNB	-6%

modified

One possible solution:

1. Query the cryptos that are *< -5%* from the given ranking system.
2. Find the three consecutive coins by traversing (1).

Note that (1) is declarative and (2) is procedural.

05 Cursor

first three consecutive coins that are *down by more than 5%*.

SQL *only*

```
CREATE OR REPLACE FUNCTION consCryptosDown
  (IN n INT)
RETURNS TABLE(rank INT, sym CHAR(4)) AS $$
DECLARE
  curs  CURSOR FOR (SELECT * FROM cryptosRank
                    WHERE changes < -5);
  r1     RECORD;
  r2     RECORD;
BEGIN
  OPEN curs;

  LOOP

    <code snippet goes here>

  END LOOP;

  CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

```
FETCH curs INTO r1;
EXIT WHEN NOT FOUND;

FETCH RELATIVE (n-1)
  FROM curs INTO r2;
EXIT WHEN NOT FOUND;

IF r2.rank - r1.rank = n-1 THEN
  MOVE RELATIVE -(n) FROM curs;

  FOR c IN 1..n LOOP
    FETCH curs INTO r1;
    rank := r1.rank;
    sym  := r1.symbol;
    RETURN NEXT;
  END LOOP;

  CLOSE curs;
  RETURN;

END IF;

MOVE RELATIVE -(n-1) FROM curs;
```

	Rank	Symbol	Changes
curs →	1	BTC	-6%
curs →	3	DOGE	-6%
curs →	4	ZIL	-7%
curs →	5	XMR	-8%
curs →	6	SHIB	-8%
	8	LTC	-7%
	9	XRP	-7%
	10	BNB	-6%

Cursor

SQL *only*

- Cursor movement
  - `FETCH curs INTO r;`
  - `FETCH NEXT FROM curs INTO r;`
- Other variants
  - `FETCH PRIOR FROM curs INTO r;`
    - Fetch from previous row
  - `FETCH FIRST FROM curs INTO r;`
  - `FETCH LAST FROM curs INTO r;`
  - `FETCH ABSOLUTE 3 FROM curs INTO r;`
    - Fetch the 3rd tuple
  - `FETCH RELATIVE -2 FROM curs INTO r;`
  - `MOVE LAST FROM curs;`
  - `UPDATE/DELETE <table> … WHERE CURRENT OF curs;`

curs →

Rank	Symbol	Changes
1	BTC	-6%
3	DOGE	-6%
4	ZIL	-7%
5	XMR	-8%
6	SHIB	-8%
8	LTC	-7%
9	XRP	-7%
10	BNB	-6%

05

Summary

SQL *only*

- `plpgsql` Control Structures
  - Declare `DECLARE <var> <type> BEGIN`
  - Assignment `<var> := ...`
  - Selection `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`
  - Repetition `LOOP ... END LOOP`
    `WHILE ... LOOP ... END LOOP`
    - Break `EXIT WHEN ...`
- Cursor
  - Workflow: Declare → Open → Fetch → Check *(repeat)* → Close
  - `FETCH [PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n] [FROM] <cursor> INTO <var>`
  - `MOVE [PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n] [FROM] <cursor>;`
  - `[UPDATE | DELETE] … WHERE CURRENT OF <cursor>;`

05 PL/pgSQL - Practice

Based on this ranking system of cryptocurrencies,
I want to have daily report of *first three coins* that are
down by more than 5% and are within 2 ranks apart.

We will do it!

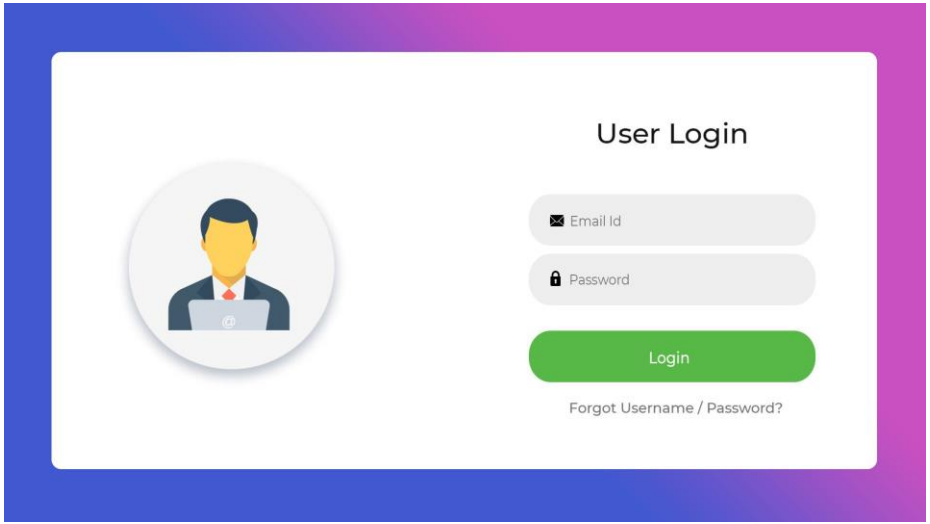
Homework. Any question?

06 SQL Injection

- Code reuse.
- Ease of maintenance.
- Performance.
- Security.

06 SQL Injection

- **What is it?**
  - A class of attacks on dynamic SQL.
- Suppose that you're developing the following login page,



06 SQL Injection

- **What is it?**
  - A class of attacks on dynamic SQL.

- **Expected case**
  - email = aa@bb.com
  - password = abcd

```
SELECT COUNT(*) FROM Users
WHERE email  = 'aa@bb.com' AND password = 'abcd';
```

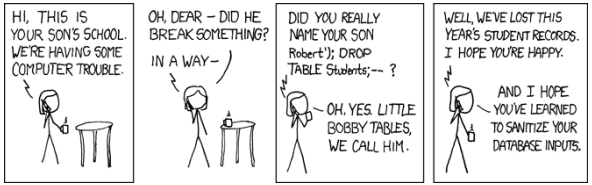
- **Malicious case**
  - email = aa@bb.com
  - password = 'OR 1 = 1 --

```
SELECT COUNT(*) FROM Users
WHERE email  = 'aa@bb.com' AND password = '' OR 1 = 1 --;
```

.pgc *file*

```
void main() {
EXEC SQL BEGIN DECLARE SECTION;
  char *query;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO @localhost USER john;

char email[100];
scanf("%s", email);
char password[100];
scanf("%s", password);

//query = "SELECT COUNT(*) FROM Users" +
//      "WHERE email = '" + name + "'" +
//      "AND password = '" + password + "';";

EXEC SQL EXECUTE IMMEDIATE :query;

EXEC SQL DISCONNECT;
}
```

```
if (count > 0) {
    //auth is successful
}
```



06 SQL Injection

- **Protect the DB!**
  - Use a function or procedure.
  - **Why?**
    - SQL function or procedure is compiled and stored in DB.
    - At runtime, anything in `email and password` are treated as strings.

```
void main() {
```

.pgc *file*

```
EXEC SQL BEGIN DECLARE SECTION;
  char *query;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO @localhost USER john;

char email[100];
scanf("%s", email);
char password[100];
scanf("%s", password);

//query = "SELECT * FROM verifyUser" +
//      "(" + name + "," + password + ");";

EXEC SQL EXECUTE IMMEDIATE :query;

EXEC SQL DISCONNECT;
```

```
}
```

```
CREATE OR REPLACE FUNCTION verifyUser
  (IN email_param TEXT, IN password_param TEXT)
RETURNS INT AS $$
  SELECT COUNT(*) FROM Users
  WHERE email  = email_param
  AND password = password_param;
$$ LANGUAGE sql;
```

Generated Query

```
SELECT COUNT(*)
FROM Users
WHERE email  = 'aa@bb.com'
AND password = '\' OR 1 = 1 --';
```

06 SQL Injection

- **Protect the DB!**
  - Use prepares statements.
  - **Why?**
    - SQL query is compiled when it is prepared.
    - At runtime, anything in `email and password` are treated as strings.

```
void main() {
    EXEC SQL BEGIN DECLARE SECTION;
      const char *query = "SELECT COUNT(*)
                           FROM Users
                           WHERE email = ?
                           AND password = ?;";
       char name[100], password[100];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO @localhost USER john;

    scanf("%s", email);
    scanf("%s", password);

    EXEC SQL PREPARE stmt FROM :query;
    EXEC SQL EXECUTE stmt USING :email, :password;
    EXEC SQL DEALLOCATE PREPARE stmt;
    EXEC SQL DISCONNECT;
}
```

Generated Query

```
SELECT COUNT(*)
FROM Users
WHERE email  = 'aa@bb.com'
AND password = '\' OR 1 = 1 --';
```

Summary

1. Quick Recap on SQL
   - "Generic" queries may be easier to be solved using SQL.
2. Motivation
   - "Specific" queries may be easier to be solved using a procedural language.
3. Host language + SQL
   - Use host procedural language to interact with the database.
4. PL/pgSQL Part I
   - Use SQL procedural language, e.g., SQL function and procedure.
5. PL/pgSQL Part II
   - Use SQL procedural language, e.g., variables, cursor, and control structure.
6. SQL Injection
   - Sanitize user inputs to avoid injection of malicious query.

THANK YOU