CS2102

Database Systems

L07: Stored Procedures/Functions

Requirement

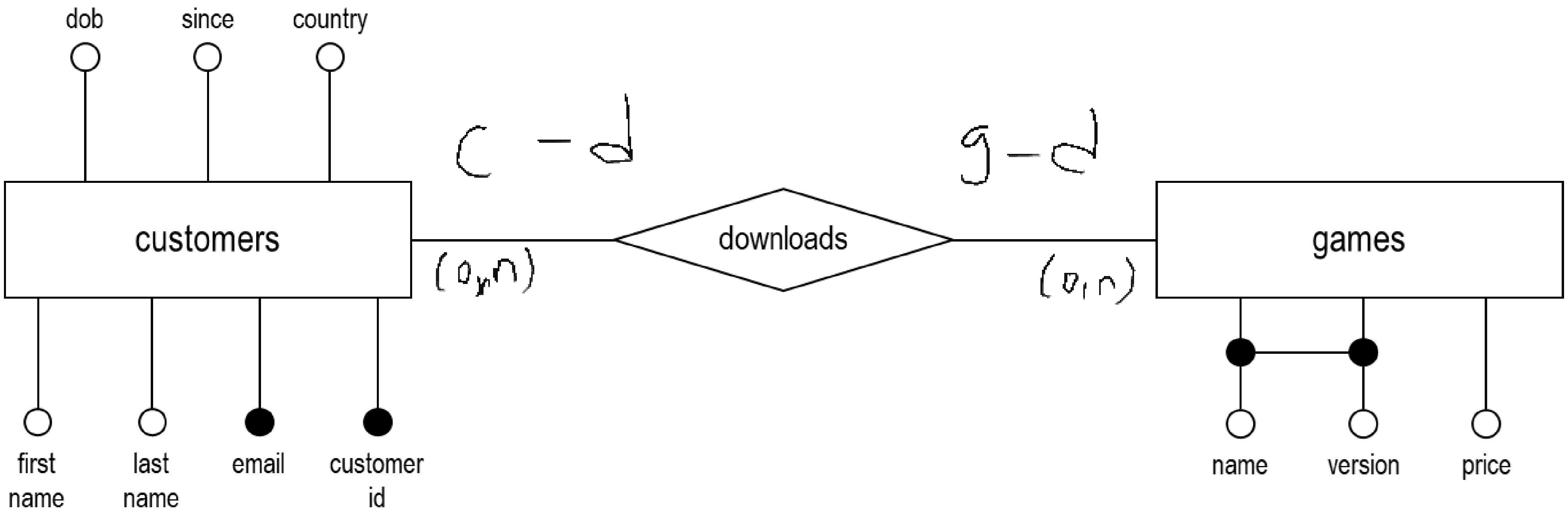
Game Store Requirement

Our company, **Apasaja Pte Ltd**, has been commissioned to develop an application to manage the data of an online app store. We want to store several items of information about our customers such as their first name, last name, date of birth, e-mail, date and country of registration to our online sales service and the customer identifier that they have chosen.

We also want to manage the list of our products, games, their name, their version, and their price. The price is fixed for each version of each game. Finally, our customers buy and download games. We record which version of which game each customer has downloaded. It is not essential to keep the download date for this application.

Design

Entity-Relationship Diagram



Schema

Customers

```
CREATE TABLE IF NOT EXISTS customers (
  first_name VARCHAR(64) NOT NULL,
  last_name VARCHAR(64) NOT NULL,
  email VARCHAR(64) UNIQUE NOT NULL,
  dob DATE NOT NULL,
  since DATE NOT NULL,
  customerid VARCHAR(16) PRIMARY KEY,
  country VARCHAR(16) NOT NULL
);
```

Game Store ① ② ③ ❹ ⑤ | Business Rule | Attempt

Schema

Games

```
CREATE TABLE IF NOT EXISTS games(
  name VARCHAR(32),
  version CHAR(3),
  price NUMERIC NOT NULL,
  PRIMARY KEY (name, version)
);
```

Game Store ① ② ③ ④ ❺ | Business Rule | Attempt

Schema

Downloads

```
CREATE TABLE downloads(
  customerid VARCHAR(16)
    REFERENCES customers(customerid)
    ON UPDATE CASCADE ON DELETE CASCADE,
  name VARCHAR(32),
  version CHAR(3),
  PRIMARY KEY (customerid, name, version),
  FOREIGN KEY (name, version)
    REFERENCES games(name, version)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

Game Store | Business Rule ❶ ❷ | Attempt

y m d
0021 -02 -03
21

Constraint

Age Restrictions

Our company decides to enforce a suitability scheme to limit access to certain games based on their content to certain audiences based on their age.

For example, a customer who is not yet 21 cannot download the game 'Domainer'.

Underage Customers

DATE

```
SELECT c.customerid
FROM customers c
WHERE EXTRACT(year FROM AGE(dob)) < 21;
```

Component

customerid
:
Adam2000
Steve1999
:

Constraint

Age Restrictions

Our company decides to enforce a suitability scheme to limit access to certain games based on their content to certain audiences based on their age.

For example, a customer who is not yet 21 cannot download the game 'Domainer'.

Domainer Customers

```sql
SELECT DISTINCT c.customerid,
  EXTRACT(year FROM AGE(dob)) AS age
FROM customers c
  NATURAL JOIN downloads d
WHERE d.name = 'Domainer';
```

customerid	age
:	:
Adam2000	13
Steve1999	14
:	:

Game Store | Business Rule | Attempt ❶ ❷

Check?

Downloads

```sql
CREATE TABLE downloads(
  customerid VARCHAR(16)
    REFERENCES customers(customerid)
    ON UPDATE CASCADE ON DELETE CASCADE
    CHECK (customerid NOT IN (
      SELECT c.customerid FROM customers c NATURAL JOIN downloads d
      WHERE c.customerid = d.customerid AND d.name = 'Domainer'
      AND EXTRACT(year FROM AGE(dob)) < 21)),
  name VARCHAR(32),
  version CHAR(3),
  PRIMARY KEY (customerid, name, version),
  FOREIGN KEY (name, version)
    REFERENCES games(name, version)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

Game Store | Business Rule | Attempt ❶ ❷

Assertion?

Downloads

```
CREATE ASSERTION r21 CHECK (
  NOT EXISTS (
    SELECT c.customerid
    FROM customers c NATURAL JOIN downloads d
    WHERE d.name = 'Domainer'
      AND EXTRACT(year FROM AGE(dob)) < 21
  ))
```

Assertion Constraints

SQL92 defines **assertions** to define integrity constraints outside of a table definition. Assertions can declare constraints that involve multiple tables.

Assertions are **NOT** yet implemented in PostgreSQL.

Standard

has no return

has return

SQL:1999

The **SQL:1999** standard introduced the concept of **stored procedures** and **stored functions**. This enables creation and execution of code directly within the database.

PL/pgSQL procedures and functions can be called from SQL and can call SQL.

Advantages

- Implemented and maintained in a **single place**.
- Executed on server-side with **powerful machine**.
- Minimize **network latency**.

Disdvantages

- Did not benefit from **optimization** by DBMS.
- Underutilizing **clients' computing resources**.
- Code is not **portable**.

Calculate Age?

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

Calculate Age?

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

1

1 9 (

no string

Note

Function definition is between **$$** *(dollar quoted string)*. We can add **tag** such as **tag** as long as we close with the same **tag**.

Calculate Age?

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

Note

Blocks are not based on indentation *(unlike Python)* or with curly bracket *(unlike C)*. Instead, they open with keyword and closed with END.

Calculate Age?

optional

Parameter

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

return type

Note

The function name is **calculate_age**. It accepts **one parameter** named **dob** with type **DATE**. Other types are available.

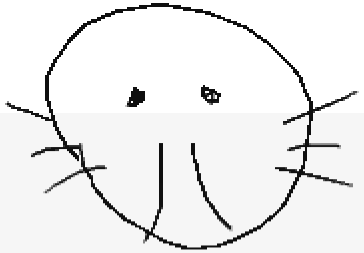
Calculate Age?

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

Note

The return type is **INTEGER** declared with keyword **RETURNS** *(with S)*. The actual return is given by the keyword **RETURN** *(without S)*.

Calculate Age?



equality check

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

Note

Assignment is done with the symbol `:=` *(the walrus operator)* because the symbol `=` is already used for equality check. This is a ***common mistake***.

IF ELSE END IF

Calculate Age?

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

ELSIF

ELSEIF

ELSE IF

Note

Selection is done with IF .. THEN .. END IF;. The condition is between IF .. THEN and the body is between THEN .. END IF;. *Optionally*, we may have ELSE.

PL/PgSQL

Calculate Age?

```
CREATE OR REPLACE FUNCTION calculate_age(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  years INTEGER;
BEGIN
  years := EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM dob);
  IF (EXTRACT(month FROM CURRENT_DATE) < EXTRACT(month FROM dob)) OR
     (EXTRACT(month FROM CURRENT_DATE) = EXTRACT(month FROM dob) AND
      EXTRACT(day FROM CURRENT_DATE) < EXTRACT(day FROM dob)) THEN
    years := years - 1;
  END IF;
  RETURN years;
END;
$$ LANGUAGE plpgsql;
```

sql

Note

The language is not SQL but **plpgsql**. We can still use SQL query inside but it allows other imperative construct. Other languages such as **PL/Python** are available with extension.

Select Clause

Function Invocation

We can invoke the function from SELECT clause using a standard function call technique.

```sql
SELECT calculate_age('2001-12-13');
```

Query

Since it can be used in SELECT clause, it can be used as part of an SQL query.

```sql
SELECT c.customerid, calculate_age(c.dob) AS age
FROM customers c
ORDER BY age;
```

While

```
CREATE OR REPLACE FUNCTION calculate_day(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  days INTEGER;
BEGIN
  days := 0;
  WHILE dob < CURRENT_DATE LOOP
    days := days + 1;   -- increment integer
    dob := dob + 1;     -- increment date
  END LOOP;
  RETURN days;
END;
$$ LANGUAGE plpgsql;
```

From Log to Exception

RAISE Keyword

PL/pgSQL provide 6 different message levels. Each of the level have different priority with the highest priority allows us to abort the transaction.

Level	Note
`RAISE DEBUG`	Generate messages of different priority Whether messages are displayed to the client depends on postgresql configuration
`RAISE LOG`	
`RAISE INFO`	
`RAISE NOTICE`	
`RAISE WARNING`	
`RAISE EXCEPTION`	Raises an error. Typically aborts the current transaction.

General Syntax

```
CREATE OR REPLACE FUNCTION <name> (<param> <type>, <param> <type>, ...)
RETURNS <type> AS $$
DECLARE
  <var> <type>;
    :
BEGIN



  <code>

END;
$$ LANGUAGE <language>;
```

Existing Schema

Return Single Tuple from Existing Table

We can return a single tuple from existing table by **specifying the table name** as the return type.

```
CREATE OR REPLACE FUNCTION most_expensive()
RETURNS games AS $$
  SELECT *
  FROM games g
  ORDER BY g.price DESC
  LIMIT 1;
$$ LANGUAGE sql;
```

```
SELECT *
FROM most_expensive();
```

name	version	price
Aerified	1.0	12

New Schema

Return an Arbitrary Single Tuple

We can return an arbitrary single tuple by **specifying OUT parameter** and returning a RECORD. There is also INOUT parameters that can act as both input and output.

```
CREATE OR REPLACE FUNCTION most_download
  (OUT name VARCHAR(32), OUT version CHAR(3))
RETURNS RECORD AS $$
  SELECT g.name, g.version
  FROM games g, downloads d
  WHERE g.name = d.name AND g.version = d.version
  GROUP BY g.name, g.version
  ORDER BY COUNT(*) DESC
  LIMIT 1;
$$ LANGUAGE sql;
```

```
SELECT *
FROM most_download();
```

name	version
Y-find	1.2

Existing Schema

Return A Table from Existing Table

We can return a table from existing table by **specifying the table name** as the return type prefixed with keyword SETOF.

```
CREATE OR REPLACE FUNCTION all_most_expensive()
RETURNS SETOF games AS $$
  SELECT *
  FROM games g
  GROUP BY g.name, g.version
  HAVING g.price >= ALL(
    SELECT g1.price FROM games g1
  );
$$ LANGUAGE sql;
```

```
SELECT *
FROM all_most_expensive();
```

New Schema

Return an Arbitrary Table

We can return an arbitrary table by **specifying OUT parameter** and returning a SETOF RECORD.

```
CREATE OR REPLACE FUNCTION all_most_download
  (OUT name VARCHAR(32), OUT version CHAR(3))
RETURNS SETOF RECORD AS $$
  SELECT g.name, g.version
  FROM games g, downloads d
  WHERE g.name = d.name AND g.version = d.version
  GROUP BY g.name, g.version
  HAVING COUNT(*) >= ALL ( /* ... */ );
$$ LANGUAGE sql;
```

```
SELECT *
FROM all_most_download();
```

IN	OUT	INOUT
Default	Specified	Specified
Value in	Value out	Value in Value out
Constant	*Uninitialized* variable	*Initialized* variable

New Schema

Return an Arbitrary Table

We can also return an arbitrary table by returning TABLE(..) with the attributes specified for the given table.

```
CREATE OR REPLACE FUNCTION all_most_download()
  RETURNS TABLE(name VARCHAR(32), version CHAR(3))
AS $$
  SELECT g.name, g.version
  FROM games g, downloads d
  WHERE g.name = d.name AND g.version = d.version
  GROUP BY g.name, g.version
  HAVING COUNT(*) >= ALL ( /* ... */ );
$$ LANGUAGE sql;
```

```
SELECT *
FROM all_most_download();
```

Quick Quiz

Is TABLE(..) ***always*** equivalent to SETOF RECORDS?

Variables

No More Hardcoding

Even with SQL as language, stored function is more powerful as we can remove constant and replace it with variables.

```
CREATE OR REPLACE FUNCTION find_age
  (game VARCHAR(32), age INTEGER)
RETURNS SETOF customers AS $$
  SELECT * FROM customers c
  WHERE calculate_age(c.dob) < age
    AND c.customerid IN (
      SELECT d.customerid FROM downloads d
      WHERE d.name = game
  )
$$ LANGUAGE sql;
```

```
SELECT *
FROM find_age('Domainer', 21);
```

Quiz 1

Question

Consider the stored function on the right. How many rows *(if any)* are printed below?

```
SELECT *
FROM most_expensive();
```

```
CREATE OR REPLACE FUNCTION most_expensive()
RETURNS games AS $$
  SELECT g.name, g.version
  FROM games g
  ORDER BY g.price DESC
  LIMIT 1;
$$ LANGUAGE sql;
```

	Choice	Comment	
A	1 row		?
B	More than 1 rows		?
C	Error		?

Quiz 2

Question

Consider the following valid statement to `foo`.

```sql
SELECT a, d
FROM foo(1, 2);
```

Which are possible definition(s) of `foo`?

```sql
-- Option 1 ✗
func(INOUT a INT, IN b INT, IN c INT, OUT d INT)
-- Option 2 ✓
foo(INOUT a INT, IN b INT, OUT c INT, OUT d INT)
-- Option 3 ✓
foo(OUT a INT, INOUT b INT, IN c INT, OUT d INT)
-- Option 4 ✗
foo(OUT a INT, INOUT b INT, IN c INT, INOUT d INT)
```

	Choice	Comment	
A	Option 1		?
B	Option 2		?
C	Option 3		?
D	Option 4		?
E	*None of the above*		?

Break

Back by 13:10

No Return

Procedure

We can think of a **procedure** as a function that does not return any value*.

```
CREATE OR REPLACE PROCEDURE <name> (<param> <type>, <param> <type>, ...)
AS $$
DECLARE
  <var> <type>;
    :
BEGIN
  <code>
END;
$$ LANGUAGE <language>;
```

* In most cases, this generalization is true, but we have `VOID` type and `INOUT` parameter. There are other differences which we will not go into details in this course.

Safe Download

```
CREATE OR REPLACE PROCEDURE
  download_game(cid VARCHAR(16), gname VARCHAR(32), gver CHAR(3))
AS $$
BEGIN




    INSERT INTO downloads VALUES (cid, gname, gver);




END; $$ LANGUAGE plpgsql;
```

Local Variable

```
CREATE OR REPLACE PROCEDURE
  download_game(cid VARCHAR(16), gname VARCHAR(32), gver CHAR(3))
AS $$
DECLARE age INTEGER;
BEGIN
  SELECT calculate_age(c.dob) INTO age FROM customers c
  WHERE c.customerid = cid;
  IF gname ≠ 'Domainer' ~~AND~~ age >= 21 THEN
    INSERT INTO downloads VALUES (cid, gname, gver);
  END IF;


END; $$ LANGUAGE plpgsql;
```

<>

OR

"allow customer to download"

⇓

equivalent to INSERT to downloads

If Not Exist

```
CREATE OR REPLACE PROCEDURE
  download_game(cid VARCHAR(16), gname VARCHAR(32), gver CHAR(3))
AS $$
BEGIN
  IF NOT EXISTS (
    SELECT c.customerid FROM customers c
    WHERE gname = 'Domainer' AND calculate_age(c.dob) < 21
      AND c.customerid = cid ) THEN
    INSERT INTO downloads VALUES (cid, gname, gver);
  END IF;


END; $$ LANGUAGE plpgsql;
```

check for violation

Insert + Delete

```
CREATE OR REPLACE PROCEDURE
  download_game(cid VARCHAR(16), gname VARCHAR(32), gver CHAR(3))
AS $$
BEGIN




    INSERT INTO downloads VALUES (cid, gname, gver);
    DELETE FROM downloads d WHERE d.customerid IN (
      SELECT c.customerid FROM customers c NATURAL JOIN downloads d1
      WHERE name = 'Domainer' AND calculate_age(c.dob) < 21
    );
END; $$ LANGUAGE plpgsql;
```

Quiz 3

Question

Consider the stored procedure `download_game`. What can go wrong with the procedure given the constraint that only customers with age >= 21 can download the game?

Write my own INSERT

You can update your DOB

You can update the game

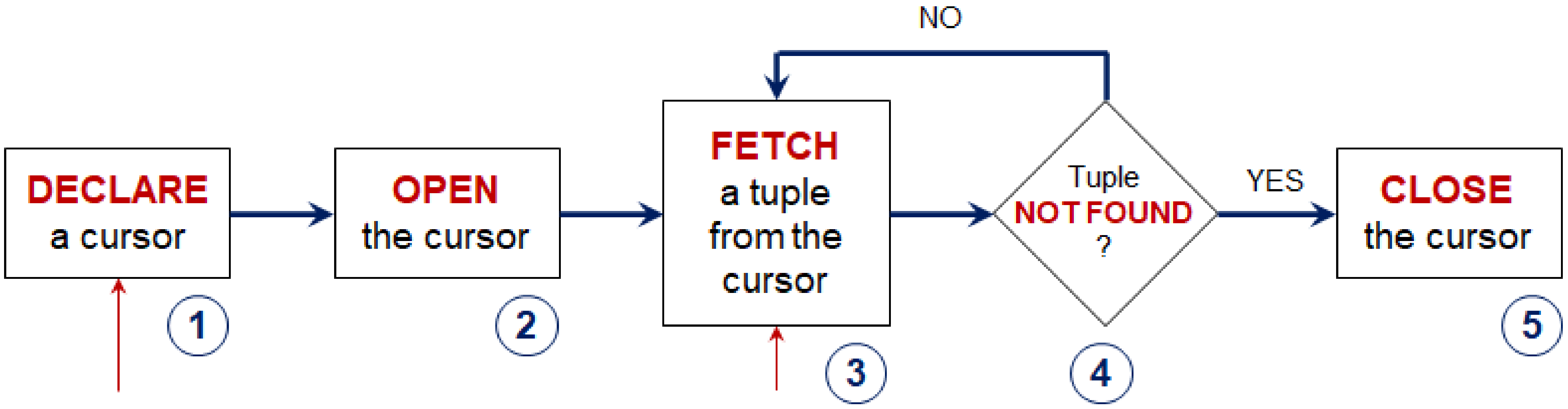
Pointer to Row

Cursor Mechanism

PL/pgSQL offers a `cursor` mechanism. Cursors are the scalable way to process data from a query.

A cursor can move in different directions and modes: `NEXT`, `LAST`, `PRIOR`, `FIRST`, `ABSOLUTE`, `RELATIVE`, `FORWARD`, `BACKWARD`, `SCROLL`, `NO SCROLL` indicate whether the cursor can be scrolled backwards or not, respectively. Cursor ***must be closed***.

Workflow



The cursor is associated with a SELECT statement at declaration

Once a tuple is fetched, we can do some operations based on it

Largest Increase

Largest Price Jump

Find the largest increase in price between version.

Largest Increase

Largest Price Jump

Find the **largest increase in price** between version.

```
CREATE OR REPLACE FUNCTION max_increase(gname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
  cur CURSOR (vname VARCHAR(32)) FOR
      SELECT g.price FROM games g WHERE g.name = vname
      ORDER BY g.version ASC;
  res NUMERIC;  prev NUMERIC;  curr NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  res := 0;  prev := 0;
  LOOP
    FETCH cur INTO curr;
    EXIT WHEN NOT FOUND;
    IF (curr - prev) >= res THEN res := (curr - prev); END IF;
    prev := curr;
  END LOOP;
  CLOSE cur;
  RETURN res;
END; $$ LANGUAGE plpgsql;
```

Largest Increase

Largest Price Jump

Find the **largest increase in price** between version.

Step 1: Declare Cursor

It is associated with a **query**. The query may take in a parameter. It can also be complex queries.

```
CREATE OR REPLACE FUNCTION max_increase(gname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
  cur CURSOR (vname VARCHAR(32)) FOR
      SELECT g.price FROM games g WHERE g.name = vname
      ORDER BY g.version ASC;
  res NUMERIC;  prev NUMERIC;  curr NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  res := 0;  prev := 0;
  LOOP
    FETCH cur INTO curr;
    EXIT WHEN NOT FOUND;
    IF (curr - prev) >= res THEN res := (curr - prev); END IF;
    prev := curr;
  END LOOP;
  CLOSE cur;
  RETURN res;
END; $$ LANGUAGE plpgsql;
```

Largest Increase

Largest Price Jump

Find the **largest increase in price** between version.

Step 2: Open Cursor

We start by **opening** the cursor using **OPEN** keyword. We can pass the parameter to the cursor.

```
CREATE OR REPLACE FUNCTION max_increase(gname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
  cur CURSOR (vname VARCHAR(32)) FOR
      SELECT g.price FROM games g WHERE g.name = vname
      ORDER BY g.version ASC;
  res NUMERIC;  prev NUMERIC;  curr NUMERIC;
BEGIN
  OPEN cur(v~~name :=~~ gname);
  res := 0;  prev := 0;
  LOOP
    FETCH cur INTO curr;
    EXIT WHEN NOT FOUND;
    IF (curr - prev) >= res THEN res := (curr - prev); END IF;
    prev := curr;
  END LOOP;
  CLOSE cur;
  RETURN res;
END; $$ LANGUAGE plpgsql;
```

Largest Increase

Largest Price Jump

Find the **largest increase in price** between version.

Step 3: Fetch Row

We fetch a row using FETCH keyword. This is usually done in a LOOP. By default, we fetch the **next element**.

```
CREATE OR REPLACE FUNCTION max_increase(gname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
  cur CURSOR (vname VARCHAR(32)) FOR
      SELECT g.price FROM games g WHERE g.name = vname
      ORDER BY g.version ASC;
  res NUMERIC;  prev NUMERIC;  curr NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  res := 0;  prev := 0;
  LOOP
    FETCH cur INTO curr;
    EXIT WHEN NOT FOUND;
    IF (curr - prev) >= res THEN res := (curr - prev); END IF;
    prev := curr;
  END LOOP;
  CLOSE cur;
  RETURN res;
END; $$ LANGUAGE plpgsql;
```

Largest Increase

Largest Price Jump

Find the **largest increase in price** between version.

Step 4: Check Row

We **stop** and **exit the loop** if there is no more row.

`EXIT WHEN NOT FOUND`

```
while (true) {
    if (notfound) {
        break;
    }
}
```

```
CREATE OR REPLACE FUNCTION max_increase(gname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
  cur CURSOR (vname VARCHAR(32)) FOR
      SELECT g.price FROM games g WHERE g.name = vname
      ORDER BY g.version ASC;
  res NUMERIC;  prev NUMERIC;  curr NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  res := 0;  prev := 0;
  LOOP
    FETCH cur INTO curr;
    EXIT WHEN NOT FOUND;
    IF (curr - prev) >= res THEN res := (curr - prev); END IF;
    prev := curr;
  END LOOP;
  CLOSE cur;
  RETURN res;
END; $$ LANGUAGE plpgsql;
```

Largest Increase

Largest Price Jump

Find the **largest increase in price** between version.

Step 5: Close Cursor

Do not forget to close the cursor with `CLOSE` keyword. Otherwise, we have memory leak.

```
CREATE OR REPLACE FUNCTION max_increase(gname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
  cur CURSOR (vname VARCHAR(32)) FOR
      SELECT g.price FROM games g WHERE g.name = vname
      ORDER BY g.version ASC;
  res NUMERIC;  prev NUMERIC;  curr NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  res := 0;  prev := 0;
  LOOP
    FETCH cur INTO curr;
    EXIT WHEN NOT FOUND;
    IF (curr - prev) >= res THEN res := (curr - prev); END IF;
    prev := curr;
  END LOOP;
  CLOSE cur;
  RETURN res;
END; $$ LANGUAGE plpgsql;
```

Largest Increase

Largest Price Jump

Find the **largest increase in price** between version.

Computation

We find the difference with previous row and update our result accordingly. This is a **stateful** computation.

```
CREATE OR REPLACE FUNCTION max_increase(gname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
  cur CURSOR (vname VARCHAR(32)) FOR
      SELECT g.price FROM games g WHERE g.name = vname
      ORDER BY g.version ASC;
  res NUMERIC;  prev NUMERIC;  curr NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  res := 0;  prev := 0;
  LOOP
    FETCH cur INTO curr;
    EXIT WHEN NOT FOUND;
    IF (curr - prev) >= res THEN res := (curr - prev); END IF;
    prev := curr;
  END LOOP;
  CLOSE cur;
  RETURN res;
END; $$ LANGUAGE plpgsql;
```

0

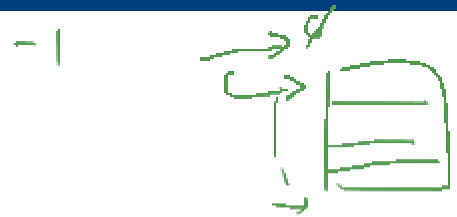
9 1.0 $\boxed{2}$

curr=2 9 1.1 5

prev=2]=>3 9 2.0 12

curr=5

Flexible Navigation



Cursor Direction

A useful mental model is to think of `FETCH` performing two operations at the same time.

1. **Move** the cursor to a new location.
2. **Retrieve** the row at the new location.

Using this mental model, on `OPEN`, the cursor is pointing to a location **before** the first row.

Keyword	**Movement**
`NEXT/FORWARD`	To the next row *(default)*
`PRIOR/BACKWARD`	To the previous row
`FIRST`	To the first row
`LAST`	To the last row
`ABSOLUTE` *n*	To the index ***n*** *(starting from 0, can be negative)*
`RELATIVE` *n*	Forward by ***n*** rows *(or backward if negative)*

```
postgres=# exit
Press any key to continue . . .
```