Database

Programming

Application Layer

Case Study

Game Store

Requirement

Game Store Requirement

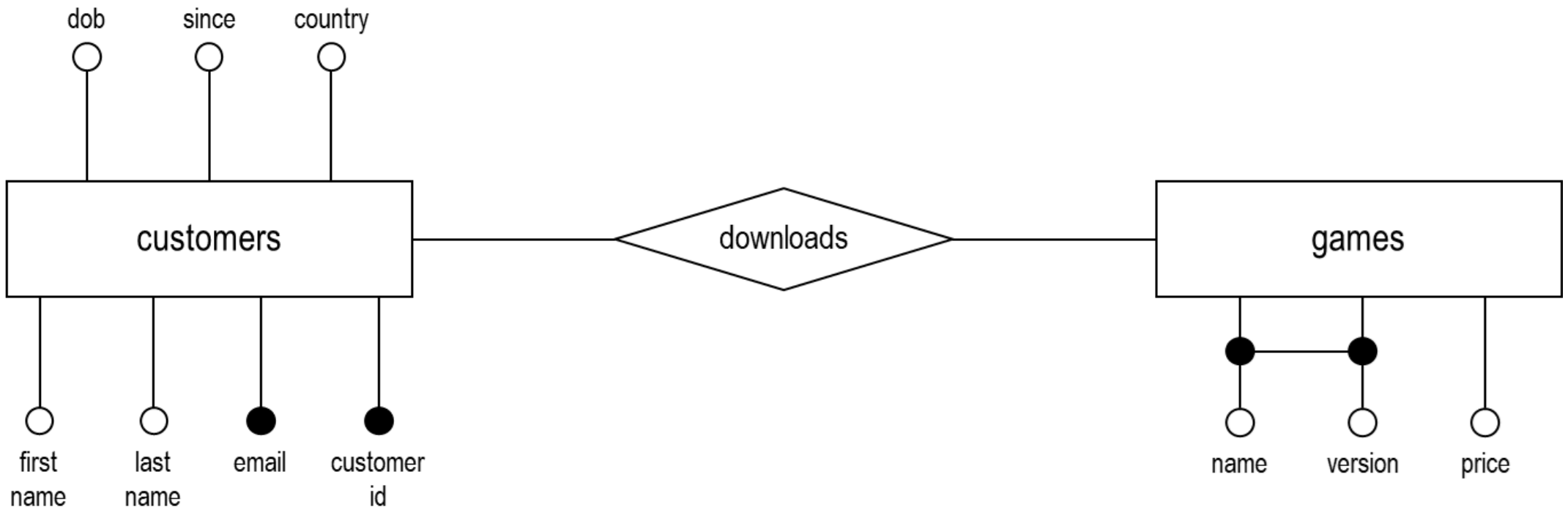
Our company, **Apasaja Pte Ltd**, has been commissioned to develop an application to manage the data of an online app store. We want to store several items of information about our customers such as their first name, last name, date of birth, e-mail, date and country of registration to our online sales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

Case Study

Design

Entity-Relationship Diagram



Case Study

Constraints

Additional

Age Restriction

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 Customer

There are some underage customers.

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

Domainer Download

Several customers downloaded Domainer.

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

Case Study

Constraints

Additional

Age Restriction

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 Customer Who Downloaded Domainer

Some of the customers who downloaded Domainer are underaged.

```
SELECT DISTINCT c.customerid
FROM customers c NATURAL JOIN downloads d
WHERE d.name = 'Domainer'
  AND EXTRACT(year FROM AGE(dob)) < 21;
```

Constraints

SQL

Check

Issue?

PostgreSQL does **NOT** accept subqueries in `CHECK` constraints.

CHECK Constraints

The most natural way to enforce R21 constraint is to write a `CHECK` constraints.

```
CREATE TABLE downloads (
  customerid  VARCHAR(16) REFERENCES customers(customerid)
    ON UPDATE CASCADE ON DELETE CASCADE DEFERRABLE INITIALLY DEFERRED
    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 DEFERRABLE INITIALLY DEFERRED);
```

Constraints

SQL

Assertions

Issue?

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

ASSERTION Constraints

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

```
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
  ))
```

Stored Procedures

Note

Procedures can be thought of as functions without return value.

Preliminary

SQL

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

PostgreSQL, like many modern DBMS, allows users to store, share, and execute code on the server. **PL/pgSQL** procedures and functions can be called from SQL and can call SQL.

While PL/pgSQL aligns partially with the SQL standard, it shares similarities with other procedural languages like Oracle's *PL/SQL*, SQL Server's *Transact-SQL*, and IBM DB2's *SQL Procedural Language*.

These languages implement variations of the Persistent Stored Modules component of the SQL standard.

Stored Procedures

Preliminary

Properties

Advantage

Stored procedures and functions are shared and reused by all users under access control. The application logic they encode is implemented and maintained in a **single place**.

Since they are executed **on the server**, typically a more powerful machine, they reduce the need for data transfer across network, improving performance. This can minimize **network latency**, especially for complex operations. They can be **pre-compiled** and their code cached.

Disadvantage

However, the SQL code within stored procedures may not always benefit from **adaptive optimization**. The query plans might not adjust dynamically based on the changing data patterns.

Additionally, the **clients' computing resources** remain underutilized, as the heavy lifting is done solely on the server side. The languages and concepts are **complicated**. The code is not portable from one DBMS to the next which may cause problem for migration.

Stored Procedures

Functions

Create

```
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;
```

Stored Procedures

Functions

Create

```
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

Function definition is between $$.

Stored Procedures

Functions

Create

```
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 function name is `calculate_age` and it accepts one parameter. Available types include `NUMERIC`, `VARCHAR()`, *etc.*.

Stored Procedures

Functions

Create

```
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 language is `plpgsql`. Other language are available such as `SQL` or `PL/Python` *(requires extension to be enabled)*.

Stored Procedures

Functions

Invocation

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

Note

We did not need to actually define this function as it is somehow already exists in PostgreSQL.

```
SELECT c.customerid,
       EXTRACT(year FROM AGE(c.dob)) AS age
FROM customers c
ORDER BY age;
```

Stored Procedures

Functions

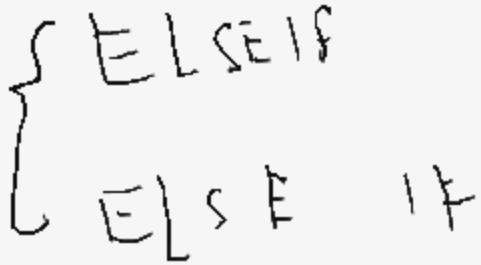
Constructs

Language Constructs

Like any programming language, PL/pgSQL offers several control structures but it also offers built-in constructs to interact with the database.

Selection

```
IF <condition>
  THEN ...
  ELSIF <condition>
  THEN ...
  ELSE ...
  END IF;
```



Repetition

```
FOR <some variable> IN (...)
WHILE <condition>
  LOOP ...
  EXIT
  EXIT WHEN
  END LOOP;
```

Stored Procedures

Functions

Example

Check Illegal Download

Implement a function `is_r21()` that returns `TRUE` if there is an illegal download of 'Domainer' by an underaged customer *(i.e., age less than 21)*.

```
CREATE OR REPLACE FUNCTION is_r21()
RETURNS BOOLEAN AS $$
BEGIN
  IF EXISTS (SELECT * FROM customers c NATURAL JOIN downloads d
             WHERE name = 'Domainer' AND EXTRACT(year FROM AGE(c.dob)) < 21)
  THEN RETURN FALSE;
  ELSE RETURN TRUE;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Usage

```
SELECT is_r21();
-- False
```

Stored Procedures

Functions

Example

Usage

```
CALL clean_r21();
SELECT is_r21();  -- True
```

Remove Illegal Download

Let us **remove** all the illegal downloads and check the result of the function `is_r21()` again. We can remove with a procedure.

```
CREATE OR REPLACE PROCEDURE clean_r21()
AS $$  -- no 'RETURNS' keyword
BEGIN
  IF NOT is_r21() THEN
    DELETE FROM downloads d WHERE d.customerid IN (
      SELECT c.customerid FROM customers c NATURAL JOIN downloads d1
      WHERE name = 'Domainer' AND EXTRACT(year FROM AGE(c.dob)) < 21
    );
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Stored Procedures

Functions

Example

Usage

```
CALL download_game('Tammy1998', 'Domainer', '2.0');
CALL download_game('JohnnyG89', 'Domainer', '2.0');
```

Insert Only Legal Download

We can also create a **procedure** to insert only legal downloads by utilizing the cleanup `clean_r21()`.

```
CREATE OR REPLACE PROCEDURE download_game(cid VARCHAR(16), gname VARCHAR(32), gver CHAR(3))
AS $$
BEGIN
  INSERT INTO downloads VALUES (cid, gname, gver);
  CALL clean_r21();
END;
$$ LANGUAGE plpgsql;
```

Issue

User may **forgot** to use this procedure. Instead, they use `INSERT` statement directly.

Stored Procedures

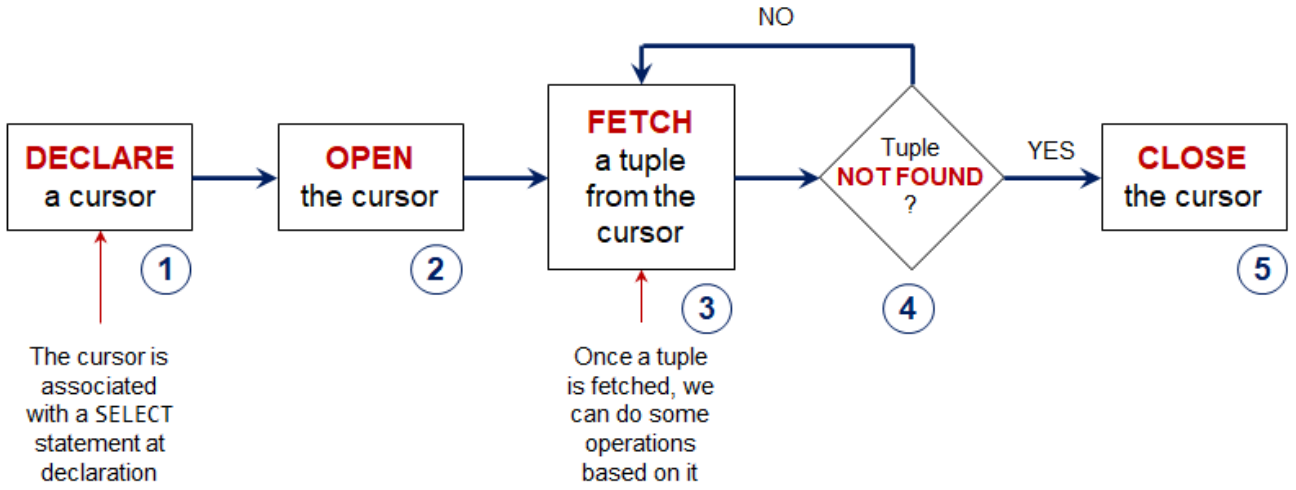
Cursor

Basic

Cursor

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***.



Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Step 1: Declare Cursor

It is associated with a **query**. In this case, a simple SELECT-FROM-WHERE.

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Step 2: Open Cursor

We can pass the argument **gname** to be accepted by the parameter **vname**.

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Step 3: Fetch Tuple

This is **typically** done inside a loop.

By default, we fetch the **NEXT** element.

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;  -- actual fetching
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Step 4: Check if Found

We stop and **exit the loop** if not found.

```
EXIT WHEN NOT FOUND
```

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Step 5: Close Cursor

Do NOT forget to close the cursor to release resources.

If not closed, subsequent call to `avg1` may have error.

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Computation 1: Total + Count

```
sum := sum + price;
count := count + 1;
```

This is done in a loop to compute for all values.

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Computation 2: Check Empty

The average of an empty table is **unknown** *(i.e., NULL)*

A simple `IF-THEN-ELSE`.

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Stored Procedures

Cursor

Example

Average

The following function uses a cursor to calculate the **average price** of the different versions of a game given its name.

Usage

```
SELECT avg1('Aerified');
```

avg1
6.5

Equivalent

```
SELECT ROUND(AVG(g.price), 2) FROM games g
WHERE g.name = 'Aerified';
```

```
CREATE OR REPLACE FUNCTION avg1(gname VARCHAR(32))
  RETURNS NUMERIC AS $$
DECLARE cur SCROLL CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price FROM games g WHERE g.name = vname;
  price NUMERIC;  sumprice NUMERIC;  count NUMERIC;
BEGIN
  OPEN cur(vname := gname);
  price := 0;  sumprice := 0;  count := 0;
  LOOP
    FETCH cur INTO price;
    EXIT WHEN NOT FOUND;
    sumprice := sumprice + price;  count := count + 1;
  END LOOP;
  CLOSE cur;
  IF count < 1 THEN RETURN null;
  ELSE RETURN ROUND(sumprice / count, 2);
  END IF;
END; $$ LANGUAGE plpgsql;
```

Break

Back by 19:43

Triggers

Checking

Function

```
CREATE OR REPLACE FUNCTION is_r21()
RETURNS BOOLEAN AS $$
BEGIN
  IF EXISTS (SELECT * FROM customers c NATURAL JOIN downloads d
             WHERE name = 'Domainer' AND EXTRACT(year FROM AGE(c.dob)) < 21)
  THEN RETURN FALSE;
  ELSE RETURN TRUE;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Note

Let us try to reset our database. Then, we will try to add this constraint as a check constraint.

Triggers

Checking

CHECK

```
CREATE OR REPLACE FUNCTION is_r21()
RETURNS BOOLEAN AS $$
BEGIN
  IF EXISTS (SELECT * FROM customers c NATURAL JOIN downloads d
             WHERE name = 'Domainer' AND EXTRACT(year FROM AGE(c.dob)) < 21)
  THEN RETURN FALSE;
  ELSE RETURN TRUE;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

```
ALTER TABLE downloads
ADD CONSTRAINT is_r21 CHECK (is_r21());
```

Error

PostgreSQL checks the new constraints against the **existsing data** in the table. If any rows violate the newly added constraint, the ALTER TABLE command will fail.

Triggers

Checking

CHECK

Note

Let us **delete** these unlawful downloads first. Then, once the database is **consistent**, we add the constraint again. This time, the `CHECK` is added, but is it correct?

```
DELETE FROM downloads d
WHERE d.name = 'Domainer'
  AND d.customerid IN (
    SELECT c.customerid
    FROM customers c
         NATURAL JOIN
         downloads d1
    WHERE d1.name = 'Domainer'
      AND EXTRACT(year FROM AGE(c.dob)) < 21
  );
```

```
ALTER TABLE downloads
ADD CONSTRAINT is_r21 CHECK (is_r21());
```

Triggers

Checking

CHECK

Testing

The customer with identifier `Jonathan2000` is 18 years old.

```
SELECT EXTRACT(year FROM AGE(dob)) AS age
FROM customers WHERE customerid = 'Jonathan2000';
```

Insert

Let us allow `Jonathan2000` to download Domainer 1.0.

```
INSERT INTO downloads VALUES ('Jonathan2000', 'Domainer', '1.0');
```

This unlawful download is still **inserted**.

Triggers

Checking

CHECK

Testing

The customer with identifier `Jonathan2000` is 18 years old.

```
SELECT EXTRACT(year FROM AGE(dob)) AS age
FROM customers WHERE customerid = 'Jonathan2000';
```

Insert

Let us allow `Jonathan2000` to download Aerified 1.0.

```
INSERT INTO downloads VALUES ('Jonathan2000', 'Aerified', '1.0');
```

Now the database is in an **inconsistent state**. No `INSERT`/`UPDATE`, but allow `DELETE`.

Triggers

Checking

CHECK

Limitation

The `CHECK` constraints require that functions be **immutable** or **stable**. In other words, they must **always** return the same result for the same inputs or queries.

Our function `is_r21()` depends on data from multiple rows and tables. It is neither immutable nor stable.

This attempt goes beyond the row-level scope of `CHECK` constraints defined in PostgreSQL *(also defined in other systems and the standard)*.

In addition, `CHECK` constraints cannot be deferred.

Remove Unlawful Download

```
CALL clean_r21();
-- Do not forget to add the procedure
```

Remove Constraints

```
ALTER TABLE downloads
DROP CONSTRAINT is_r21;
```

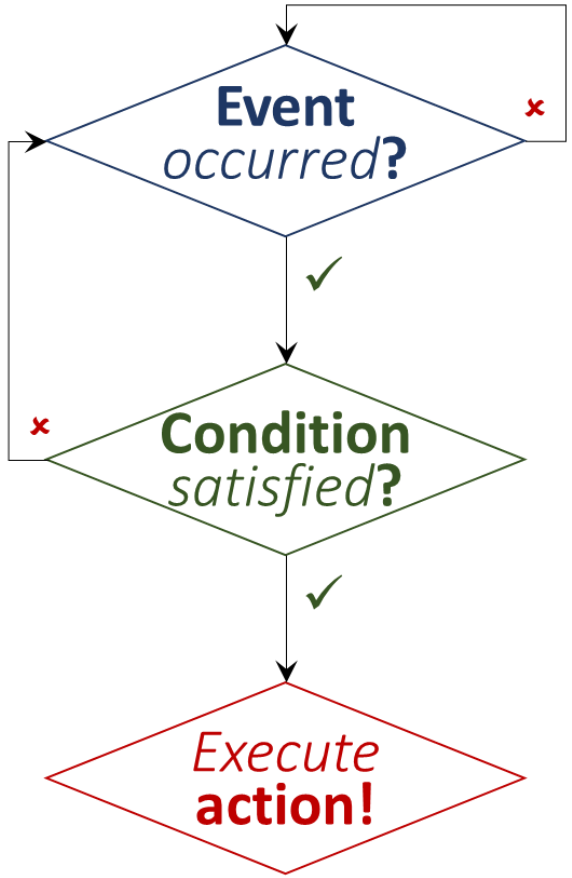
Triggers

Triggers

Basic

Event-Condition-Action

A **trigger** is a procedure or function executed when a database **event** occurs on a table.



Triggers

Triggers

Basic

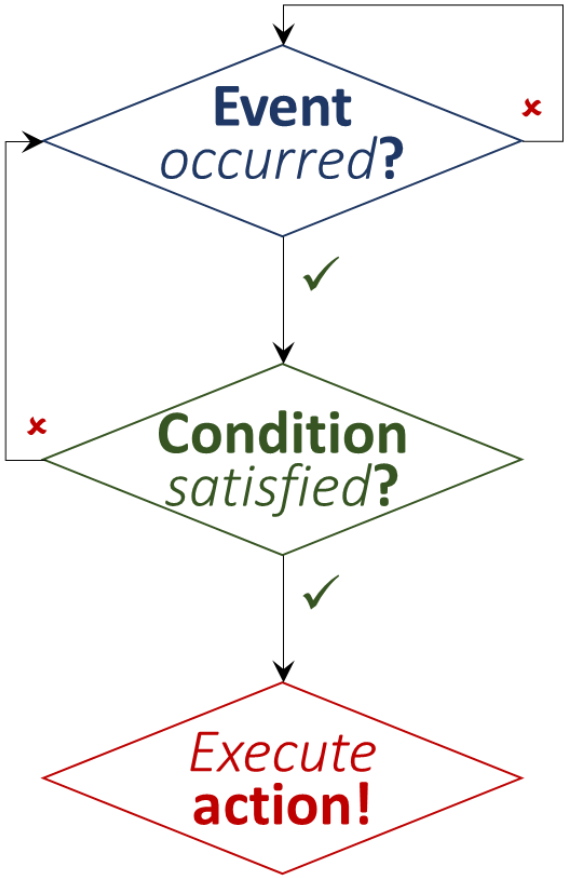
Event-Condition-Action

A **trigger** is a procedure or function executed when a database **event** occurs on a table.

Note

Triggers help maintain **data integrity**, propagate updates, and repair the database. It is a generalization of `ON UPDATE/DELETE`. Code is executed on the **server side**.

Syntax and **semantics** *varies* across DBMS, reducing **portability**. Interactions between triggers, constraints, and transactions are **difficult to control** *(i.e., chain reactions)*.



Triggers

Triggers

Basic

Event-Condition-Action

A **trigger** is a procedure or function executed when a database **event** occurs on a table.

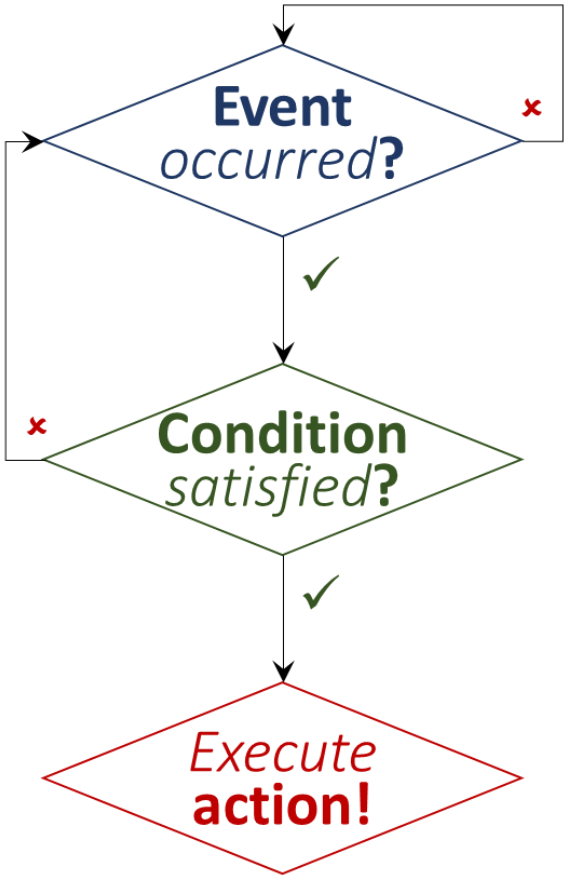
Component

Checking if **event** occurred: ***Trigger***

Binds the trigger function to operations on a **table** or a **view**.

Action executed: ***Trigger Function***

A function invoked when a **statement** *(e.g., insert, update, delete)* is executed.



Triggers

Properties

Trigger Timing

A trigger can be specified to fire

- **before the operation** is attempted on a row
  (i.e., before constraints are checked)
- **after the operation** has completed
  (i.e., after constraints are checked, and `INSERT`, `UPDATE`, or `DELETE` completed)
- **instead of the operation**
  (i.e., only in the case of operations on a `VIEW`)

Trigger Granularity

A trigger can be specified to fire

- **once for every row** using
  `FOR EACH ROW`
  A single statement can affect multiple rows
- **once for each statement** using
  `FOR EACH STATEMENT`
  Regardless of how many rows are affected

Triggers

Triggers

Example

Trigger Function

```
CREATE OR REPLACE FUNCTION fr21()
RETURNS TRIGGER AS $$ BEGIN
  IF EXISTS (
    SELECT c.customerid
    FROM customers c NATURAL JOIN downloads d
    WHERE d.name = 'Domainer'
      AND EXTRACT (year FROM AGE(c.dob)) < 21
  )
  THEN
    RAISE EXCEPTION 'Underaged!';    -- STOP!
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Trigger

```
CREATE CONSTRAINT TRIGGER tr21
AFTER INSERT OR UPDATE ON downloads
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE PROCEDURE fr21();
```

Note

The trigger is activated whenever there is an `INSERT` or `UPDATE` statement on table `downloads`.

The trigger is done `AFTER` constraint checks are done.

Triggers

Example

Trigger Function

```
CREATE OR REPLACE FUNCTION fr21()
RETURNS TRIGGER AS $$ BEGIN
  IF EXISTS (
    SELECT c.customerid
    FROM customers c NATURAL JOIN downloads d
    WHERE d.name = 'Domainer'
      AND EXTRACT (year FROM AGE(c.dob)) < 21
  )
  THEN
    RAISE EXCEPTION 'Underaged!';    -- STOP!
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Trigger

```
CREATE CONSTRAINT TRIGGER tr21
AFTER INSERT OR UPDATE ON downloads
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE PROCEDURE fr21();
```

Note

The query that performs the consistency check. In this case, we check if there is an **inconsistent** row.

This technique is often called a **global check** as it checks all rows.

Triggers

Example

Trigger Function

```
CREATE OR REPLACE FUNCTION fr21()
RETURNS TRIGGER AS $$ BEGIN
  IF EXISTS (
    SELECT c.customerid
    FROM customers c NATURAL JOIN downloads d
    WHERE d.name = 'Domainer'
      AND EXTRACT (year FROM AGE(c.dob)) < 21
  )
  THEN
    RAISE EXCEPTION 'Underaged!';    -- STOP!
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Trigger

```
CREATE CONSTRAINT TRIGGER tr21
AFTER INSERT OR UPDATE ON downloads
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE PROCEDURE fr21();
```

Note

If there is an inconsistent row, we want to **stop** the insertion. But insertion has been done since this is an `AFTER` trigger.

To stop the insertion, we need to `RAISE EXCEPTION`. The message can be anything.

Triggers

Example

Trigger Function

```
CREATE OR REPLACE FUNCTION fr21()
RETURNS TRIGGER AS $$ BEGIN
  IF EXISTS (
    SELECT c.customerid
    FROM customers c NATURAL JOIN downloads d
    WHERE d.name = 'Domainer'
      AND EXTRACT (year FROM AGE(c.dob)) < 21
  )
  THEN
    RAISE EXCEPTION 'Underaged!';    -- STOP!
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Trigger

```
CREATE CONSTRAINT TRIGGER tr21
AFTER INSERT OR UPDATE ON downloads
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE PROCEDURE fr21();
```

Note

This line has **no effect**. The keyword `NEW` refers to the new line inserted/updated. The keyword `OLD` refers to the new line inserted/deleted.

`RETURN NULL` in `BEFORE` trigger stops the operation.

Triggers

Example

Trigger Function

```
CREATE OR REPLACE FUNCTION fr21()
RETURNS TRIGGER AS $$ BEGIN
  IF EXISTS (
    SELECT c.customerid
    FROM customers c NATURAL JOIN downloads d
    WHERE d.name = 'Domainer'
      AND EXTRACT (year FROM AGE(c.dob)) < 21
  )
  THEN
    RAISE EXCEPTION 'Underaged!';    -- STOP!
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Trigger

```
CREATE CONSTRAINT TRIGGER tr21
AFTER INSERT OR UPDATE ON downloads
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE PROCEDURE fr21();
```

Issue

This is **insufficient**. We also need to check the updates to the table customers in case they can change their date of birth.

The same **trigger function** can be associated with **different triggers**.

Triggers

Example

Lawful Download

We can insert lawful download.

```
INSERT INTO downloads
VALUES ('Deborah84', 'Domainer', '1.0');
```

Unlawful Download

We can NOT insert unlawful download.

```
INSERT INTO downloads
VALUES ('Jonathan2000', 'Domainer', '1.0');
```

No Age Modification

We should not be able to modify age of customers who has downloaded Domainer.

```
UPDATE customers SET dob = '2010-08-01'
WHERE customerid = 'Deborah84';
```

Triggers

Alternative

Trigger Function

```
CREATE OR REPLACE FUNCTION fr21()
RETURNS TRIGGER AS $$ BEGIN
  IF NEW.name = 'Domainer' AND EXISTS (
    SELECT c.customerid
    FROM customers c
    WHERE c.customerid = NEW.customerid
      AND EXTRACT (year FROM AGE(c.dob)) < 21
  )
  THEN
    RETURN NULL;  -- Stop! for Before trigger
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Trigger

```
CREATE TRIGGER tr21
BEFORE INSERT OR UPDATE ON downloads
-- Cannot be deferred
FOR EACH ROW
EXECUTE PROCEDURE fr21();
```

Issue

The code assumes the inserted/updated row has the column `name`. This does NOT work for `UPDATE` on `customers`.

Need a new **trigger function** for **each table**. You might **forgot** to do some check.

Triggers

Triggers

Alternative

Trigger Function

```
CREATE OR REPLACE FUNCTION fr21()
RETURNS TRIGGER AS $$ BEGIN
  IF                          EXISTS (
    SELECT c.customerid
    FROM customers c
    WHERE c.customerid = NEW.customerid
      AND EXTRACT (year FROM AGE(c.dob)) < 21
  )
  THEN
    RETURN NULL;  -- Stop! for Before trigger
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Cleanup

```
DROP TRIGGER tr21 ON downloads;                    -- drop
DROP SCHEMA public CASCADE; CREATE SCHEMA public;  -- nuke!
```

Trigger

```
CREATE TRIGGER tr21
BEFORE INSERT OR UPDATE ON downloads
FOR EACH ROW
WHEN NEW.name = 'Domainer'
EXECUTE PROCEDURE fr21();
```

Condition

We can specify a simple **condition** for the trigger to occur. This way, the trigger is not executed on all operations.

Only simple check, cannot have queries inside **WHEN** condition.

Conclusion

Remark

Usage

Stored procedures, functions, and triggers are very **powerful** mechanism.

They can be used to move some of the application's logic closer to the data, lighten the burden on the application programmers, and make the application safer.

Paradigm

Their programming paradigm significantly departs from the **declarative programming** style of SQL queries.

Tactics

There are **two** general tactics.

1. **Global Check** — Perform operation and check if there are **any inconsistent row** at the end.
2. **Local Check** — Check if each operation will **violate constraints** before execution.

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