

# PL/pgSQL Error Handling



16

## Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

## Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

## Feedback

Please send your feedback, comments and suggestions to:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Error Handling in PL/pgSQL Blocks

Error Names and Codes

Choosing an Error Handler

Error Handling Overhead

# Error Handling in a Block

Error handling is performed if there is an EXCEPTION section  
Changes roll back to the savepoint at the beginning of the block  
an implicit savepoint is set if the block contains an EXCEPTION section

If there is a handler that matches the error

- error handler commands are executed
- the block completes successfully

If there is no suitable handler

- the block completes with an error

If a run-time error occurs within a block, the program (block, function) is usually aborted, and the current transaction enters the failure mode: it cannot be committed and can only be rolled back.

But an error can be caught and processed. It can be done by extending the block with an additional EXCEPTION section, which lists error conditions and provides statements to handle each of them.

In general, EXCEPTION is similar to the try-catch construct available in some programming languages (except for specifics related to transactions, or course).

A savepoint is implicitly set at the start of every block containing an EXCEPTION section. Before an error is processed, all changes are rolled back to the savepoint and all locks are removed.

Because of the savepoint, COMMIT and ROLLBACK commands cannot be used in procedures with EXCEPTION. But although SAVEPOINT and ROLLBACK TO SAVEPOINT commands are not supported by PL/pgSQL, you can still use savepoints and rollbacks to savepoints both in functions and procedures implicitly.

<https://postgrespro.com/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-ERROR-TRAPPING>

## Handling Errors in a Block

Let's take a look at a simple example.

```
=> CREATE TABLE t(id integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO t(id) VALUES (1);
```

```
INSERT 0 1
```

If there are no errors, all statements in a block are executed as usual:

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'The SELECT INTO statement has completed, n = %', n;
END
$$;
```

```
NOTICE: The SELECT INTO statement has completed, n = 1
DO
```

Now let's insert a "redundant" row to trigger an error.

```
=> INSERT INTO t(id) VALUES (2);
```

```
INSERT 0 1
```

If there is no EXCEPTION section in a block, the statement execution is interrupted, and the whole block is considered to be completed with an error condition:

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'The SELECT INTO statement has completed, n = %', n;
END
$$;
```

```
ERROR: query returned more than one row
HINT: Make sure the query returns a single row, or use LIMIT 1.
CONTEXT: PL/pgSQL function inline_code_block line 5 at SQL statement
```

To catch an error, a block must have an EXCEPTION section, which defines one or more error handlers.

This construct works similar to CASE: conditions are scanned from top to bottom, the first suitable code path is selected, and its statements are executed.

What will be displayed?

```
=> DO $$
DECLARE
    n integer;
BEGIN
    n := 3;
    INSERT INTO t(id) VALUES (n);
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'The SELECT INTO statement has completed, n = %', n;
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'No data';
    WHEN too_many_rows THEN
        RAISE NOTICE 'Too much data';
        RAISE NOTICE 'Rows in a table: %, n = %', (SELECT count(*) FROM t), n;
END
$$;
```

```
NOTICE: Too much data
NOTICE: Rows in a table: 2, n = 3
DO
```

The executed handler corresponds to the too\_many\_rows error. Note: if a handler is executed, the table contains two rows because of a rollback to an implicit savepoint at the beginning of the block.

Also note that the local variable keeps the value that was there when the error occurred.

---

Note the following subtlety: if an error occurs in the DECLARE section or within the EXCEPTION section of the handler itself, it will be impossible to catch it in this block.

```
=> DO $$
DECLARE
    n integer := 1 / 0; -- an error is not trapped here
BEGIN
    RAISE NOTICE 'Success';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Division by zero';
END
$$;
```

ERROR: division by zero  
CONTEXT: SQL expression "1 / 0"  
PL/pgSQL function inline\_code\_block line 3 during statement block local variable initialization

# Error Names and Codes

## Error info

error name

five-character error code

additional info: a short message, a detailed message, a hint, names of objects related to this error

## Two-level hierarchy

P0000 – plpgsql\_error

- P0001 – raise\_exception
- P0002 – no\_data\_found
- P0003 – too\_many\_rows
- P0004 – assert\_failure

XX000 – internal\_error

- XX001 – data\_corrupted
- XX002 – index\_corrupted

Each possible error has a name and a code (a five-character string). WHEN clauses accept both error names and error codes.

All errors are classified into a two-level hierarchy of sorts. Each error class has a code that ends with three zeros; it corresponds to any error with the same first two characters in its code.

For example, the code 23000 defines the class that includes all errors dealing with violations of integrity constraints (such as 23502, which stands for NOT NULL constraint violation, or 23505, which indicates a UNIQUE constraint violation).

Thus, apart from regular errors, you can specify an error class by its name or code. Besides, you can use a special name OTHERS to catch any errors (except for the fatal ones).

Apart from the name and code, each error can provide additional debug information: a short error message, a detailed message, and a hint.

All errors are described in documentation in Appendix A:

<https://postgrespro.com/docs/postgresql/16/errcodes-appendix>

Errors can be not only trapped, but also raised programmatically.

<https://postgrespro.com/docs/postgresql/16/plpgsql-errors-and-messages>

## Error Names and Codes

We have already seen error names; error codes are specified using SQLSTATE.

An error handler can return an error code and the corresponding message using the predefined variables SQLSTATE and SQLERRM (the variables are undefined outside of the EXCEPTION block).

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
EXCEPTION
    WHEN SQLSTATE 'P0003' OR no_data_found THEN -- there can be several conditions
        RAISE NOTICE '%: %', SQLSTATE, SQLERRM;
END
$$;
```

```
NOTICE: P0003: query returned more than one row
DO
```

Which error handler will be used?

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'No data. %: %', SQLSTATE, SQLERRM;
    WHEN plpgsql_error THEN
        RAISE NOTICE 'Another error. %: %', SQLSTATE, SQLERRM;
    WHEN too_many_rows THEN
        RAISE NOTICE 'Too much data. %: %', SQLSTATE, SQLERRM;
END
$$;
```

```
NOTICE: Another error. P0003: query returned more than one row
DO
```

The first applicable handler is selected, plpgsql\_error in this case (remember: this is not a specific error, but an error category). We will never get to the last error handler.

---

You can force an error using either its code or its name.

Here we use a special name others, which corresponds to any error that can be trapped (except for assertion failures and cases when the execution is aborted by user — you can catch them separately, but you almost never need to).

```
=> DO $$
BEGIN
    RAISE no_data_found;
EXCEPTION
    WHEN others THEN
        RAISE NOTICE '%: %', SQLSTATE, SQLERRM;
END
$$;
```

```
NOTICE: P0002: no_data_found
DO
```

If required, it is also possible to incorporate user-provided error codes that are not predefined, as well as pass some additional information (the example illustrates only some of the supported features):

```
=> DO $$
BEGIN
    RAISE SQLSTATE 'ERR01' USING
        message := 'Matrix failure',
        detail  := 'Irrecoverable matrix failure has occurred during execution',
        hint    := 'Contact your system administrator';
END
$$;
```

```
ERROR: Matrix failure
DETAIL: Irrecoverable matrix failure has occurred during execution
HINT: Contact your system administrator
CONTEXT: PL/pgSQL function inline_code_block line 3 at RAISE
```

Error handlers cannot get this information from variables; there is a special construct for analyzing such data in the code:

```
=> DO $$
DECLARE
    message text;
    detail text;
    hint text;
BEGIN
    RAISE SQLSTATE 'ERR01' USING
        message := 'Matrix failure',
        detail  := 'Irrecoverable matrix failure has occurred during execution',
        hint    := 'Contact your system administrator';
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS
            message := MESSAGE_TEXT,
            detail  := PG_EXCEPTION_DETAIL,
            hint    := PG_EXCEPTION_HINT;
        RAISE NOTICE E'\nmessage = %\ndetail = %\nhint = %',
            message, detail, hint;
END
$$;

NOTICE:
message = Matrix failure
detail = Irrecoverable matrix failure has occurred during execution
hint = Contact your system administrator
DO
```



# Choosing a Handler

An unhandled error is sent one level up

into the enclosing PL/pgSQL block, if available  
into the calling routine, if available

The search path of a handler is determined by the call stack

it is not defined statically, but depends on how the program executes

An unhandled error is passed to the client

the transaction enters the failure mode and has to be rolled back by the client  
the error is registered in the server log

If none of the conditions listed in the EXCEPTION section is triggered, the error goes one level up.

If an error has occurred in the inner block of a nested structure, the server will search for a handler in the enclosing block. If there is no suitable handler either, the whole outer block will be treated as failed, while the error will be passed to the next nesting level, and so on.

If the error goes through the whole nested structure and does not find an appropriate handler, it goes further up to the level of the routine that has called the outermost block. Therefore, you have to analyze the call stack to determine the order in which different error handlers will be applied.

If none of the available error handlers is triggered:

- The error message usually gets into the server log (the exact behavior depends on the server settings; see PL/pgSQL. Debugging lesson).
- The error is reported to the client that has initiated this operation in the database. The client cannot do anything about the cause of the error at this point: the transaction enters the failure mode, and it can only be rolled back.

It is up to the client to choose how to handle the error from there. For example, psql will display the error message and all the debugging information available. An end-user client may display a generic message like “contact your system administrator”.

## Choosing a Handler

Let's take a look at several examples of choosing a handler in nested blocks. What will be displayed?

```
=> DO $$
BEGIN
  BEGIN
    SELECT 1/0;
    RAISE NOTICE 'The inner block has completed';
  EXCEPTION
    WHEN division_by_zero THEN
      RAISE NOTICE 'Error in the inner block';
  END;
  RAISE NOTICE 'The outer block has completed';
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'Error in the outer block';
END
$$;
```

```
NOTICE: Error in the inner block
NOTICE: The outer block has completed
DO
```

An error is handled in the same block where it has occurred. The outer block is executed as if there has been no error at all.

---

And now?

```
=> DO $$
BEGIN
  BEGIN
    SELECT 1/0;
    RAISE NOTICE 'The inner block has completed';
  EXCEPTION
    WHEN no_data_found THEN
      RAISE NOTICE 'Error in the inner block';
  END;
  RAISE NOTICE 'The outer block has completed';
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'Error in the outer block';
END
$$;
```

```
NOTICE: Error in the outer block
DO
```

The handler in the inner block is not applicable; the block completes with an error that is handled in the outer block.

Remember that the block containing an EXCEPTION section is rolled back to the implicit savepoint at the beginning of this block. In this case, all changes made in both blocks will be rolled back.

---

And now?

```
=> DO $$
BEGIN
  BEGIN
    SELECT 1/0;
    RAISE NOTICE 'The inner block has completed';
  EXCEPTION
    WHEN no_data_found THEN
      RAISE NOTICE 'Error in the inner block';
  END;
  RAISE NOTICE 'The outer block has completed';
EXCEPTION
  WHEN no_data_found THEN
    RAISE NOTICE 'Error in the outer block';
END
$$;
```

```
ERROR: division by zero
CONTEXT: SQL statement "SELECT 1/0"
PL/pgSQL function inline_code_block line 4 at SQL statement
```

None of the handlers is triggered, and the whole transaction is aborted.

There is usually no need to handle all possible errors in the server code. There is nothing wrong in passing an error to the client. In general, an error should be handled at the level where something meaningful can be done about it. So it makes sense to process an error within the database if it can be addressed on the server side (e.g., the operation can be repeated in case of a serialization failure). We'll talk about logging error messages in PL/pgSQL. Debugging lesson.

---

Now let's take a look at an example that uses procedures.

```
=> CREATE PROCEDURE foo()
AS $$
BEGIN
    CALL bar();
END
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

```
=> CREATE PROCEDURE bar()
AS $$
BEGIN
    CALL baz();
END
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

```
=> CREATE PROCEDURE baz()
AS $$
BEGIN
    PERFORM 1 / 0;
END
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

What will happen if we call this procedure?

```
=> CALL foo();
```

```
ERROR:  division by zero
CONTEXT:  SQL statement "SELECT 1 / 0"
PL/pgSQL function baz() line 3 at PERFORM
SQL statement "CALL baz()"
PL/pgSQL function bar() line 3 at CALL
SQL statement "CALL bar()"
PL/pgSQL function foo() line 3 at CALL
```

The error message displays the call stack: top to bottom means inside out.

Note that this message (like many others) uses the term “function” instead of “procedure”.

---

An error handler can also provide access to the call stack, but it will be presented as a single string:

```
=> CREATE OR REPLACE PROCEDURE bar()
AS $$
DECLARE
    msg text;
    ctx text;
BEGIN
    CALL baz();
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS
            msg := MESSAGE_TEXT,
            ctx := PG_EXCEPTION_CONTEXT;
        RAISE NOTICE E'\nError: %\nError stack:\n%', msg, ctx;
END
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Let's check the result:

```
=> CALL foo();
```

NOTICE:  
Error: division by zero  
Error stack:  
SQL statement "SELECT 1 / 0"  
PL/pgSQL function baz() line 3 at PERFORM  
SQL statement "CALL baz()"  
PL/pgSQL function bar() line 6 at CALL  
SQL statement "CALL bar()"  
PL/pgSQL function foo() line 3 at CALL

CALL

---

Since a block with an EXCEPTION section creates an implicit savepoint, procedures cannot use COMMIT and ROLLBACK commands both in this block and in all the blocks up the call stack.

```
=> CREATE OR REPLACE PROCEDURE baz()  
AS $$  
BEGIN  
    COMMIT;  
END  
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

```
=> CALL foo();
```

NOTICE:  
Error: invalid transaction termination  
Error stack:  
PL/pgSQL function baz() line 3 at COMMIT  
SQL statement "CALL baz()"  
PL/pgSQL function bar() line 6 at CALL  
SQL statement "CALL bar()"  
PL/pgSQL function foo() line 3 at CALL

CALL

Any block with an EXCEPTION section is executed slower

because of setting an implicit savepoint

Additional costs are incurred in case of an error

because of the rollback to the savepoint

Error handling can and should be used, but not overused

PL/pgSQL is an interpreted language that uses SQL to compute expressions anyway

the speed is more than enough for most tasks

performance issues are usually related to queries, not to PL/pgSQL code

The mere inclusion of an EXCEPTION section already incurs overhead because it requires setting an implicit savepoint at the beginning of the block. If an error occurs, the rollback to the savepoint increases the overhead even more.

So if there is a simple way to avoid exception handling, it's better to do without it; you should not base your application logic on "exception juggling".

However, if error handling is really required, you should use it without doubt: errors can and must be handled regardless of the overhead.

First, the PL/pgSQL language itself is quite slow because of interpreting instructions and constantly calling SQL to compute expressions.

Second, its speed is usually still adequate. Yes, you can create a faster implementation in C, but what's the point?

And third, the main performance issues are usually caused by bad query plans that affect query speed, not by the execution speed of procedural code (for details, see the QPT course that deals with query performance tuning).

But if there is an alternative that is both simpler and faster, it should certainly be preferred.

## Overhead

To estimate the overhead, let's take a look at the following simple example.

Suppose we have a table with a text field that stores arbitrary data inserted by users (although usually a sign of bad design, it may sometimes be required). We need to extract all numbers into a separate column of a numeric type.

```
=> CREATE TABLE data(comment text, n integer);

CREATE TABLE

=> INSERT INTO data(comment)
SELECT CASE
    WHEN random() < 0.01 THEN 'not a number' -- 1%
    ELSE (random()*1000)::integer::text      -- 99%
END
FROM generate_series(1,1_000_000);

INSERT 0 1000000
```

Let's solve this problem by handling errors that come up when converting text to integer:

```
=> CREATE FUNCTION safe_to_integer_ex(s text) RETURNS integer
AS $$
BEGIN
    RETURN s::integer;
EXCEPTION
    WHEN invalid_text_representation THEN
        RETURN NULL;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

Let's check the result:

```
=> \timing on

Timing is on.

=> UPDATE data SET n = safe_to_integer_ex(comment);

UPDATE 1000000
Time: 13175.397 ms (00:13.175)

=> \timing off

Timing is off.

=> SELECT count(*) FROM data WHERE n IS NOT NULL;

 count
-----
 990104
(1 row)
```

The following implementation of our function will check the format using a (slightly simplified) regular expression, without error handling:

```
=> CREATE FUNCTION safe_to_integer_re(s text) RETURNS integer
AS $$
BEGIN
    RETURN CASE
        WHEN s ~ '^\d+$' THEN s::integer
        ELSE NULL
    END;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

Let's check this implementation:

```
=> \timing on

Timing is on.

=> UPDATE data SET n = safe_to_integer_re(comment);
```

```
UPDATE 1000000
Time: 8843.259 ms (00:08.843)
```

```
=> \timing off
```

Timing is off.

```
=> SELECT count(*) FROM data WHERE n IS NOT NULL;
```

```
count
-----
990104
(1 row)
```

This implementation is significantly faster. In this example, the exception has occurred in 1% of cases only. The more often it occurs, the more overhead will be incurred by rollbacks to the savepoint.

```
=> UPDATE data SET comment = 'not a number'; -- 100%
```

```
UPDATE 1000000
```

```
=> \timing on
```

Timing is on.

```
=> UPDATE data SET n = safe_to_integer_ex(comment);
```

```
UPDATE 1000000
Time: 19428.566 ms (00:19.429)
```

```
=> \timing off
```

Timing is off.

---

In some cases (which are not infrequent), you can do without error handling if you choose other suitable means.

Problem: return a row from a table or NULL, if there is no such row.

```
=> CREATE TABLE categories(code text UNIQUE, description text);
```

```
CREATE TABLE
```

```
=> INSERT INTO categories VALUES ('books', 'Books'), ('discs', 'CDs');
```

```
INSERT 0 2
```

A function with error handling:

```
=> CREATE FUNCTION get_cat_desc(code text) RETURNS text
AS $$
DECLARE
    desc text;
BEGIN
    SELECT c.description INTO STRICT desc
    FROM categories c
    WHERE c.code = get_cat_desc.code;

    RETURN desc;
EXCEPTION
    WHEN no_data_found THEN
        RETURN NULL;
END
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Let's check that the function works as expected:

```
=> SELECT get_cat_desc('books');
```

```
get_cat_desc
-----
Books
(1 row)
```

```
=> SELECT get_cat_desc('movies');
```

```
get_cat_desc
-----
(1 row)
```

Can we make it simpler?

---

Yes, we just need to remove STRICT or use a subquery:

```
=> CREATE OR REPLACE FUNCTION get_cat_desc(code text) RETURNS text
AS $$
BEGIN
    RETURN (SELECT c.description
            FROM categories c
            WHERE c.code = get_cat_desc.code);
END
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

It's a good illustration that we do not need PL/pgSQL here at all: we can simply use SQL.

Let's check the result:

```
=> SELECT get_cat_desc('books');
```

```
get_cat_desc
-----
Books
(1 row)
```

```
=> SELECT get_cat_desc('movies');
```

```
get_cat_desc
-----
(1 row)
```

---

Problem: update a table row with the specified ID; if there is no such row, insert it.

Here is the first approach. What is wrong with it?

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
DECLARE
    cnt integer;
BEGIN
    SELECT count(*) INTO cnt
    FROM categories c WHERE c.code = change.code;

    IF cnt = 0 THEN
        INSERT INTO categories VALUES (code, description);
    ELSE
        UPDATE categories c
        SET description = change.description
        WHERE c.code = change.code;
    END IF;
END
$$ VOLATILE LANGUAGE plpgsql;
```

CREATE FUNCTION

Almost everything is bad here, starting from the fact that such a function will not work correctly at the Read Committed isolation level if there are several concurrent sessions. That's because the data in the database can change between the executed SELECT statement and the next operation.

It can be easily demonstrated by introducing a delay between the commands. For variety, let's take a slightly different (but also incorrect) option:



```

=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
DECLARE
    cnt integer;
BEGIN
    SELECT count(*) INTO cnt
    FROM categories c WHERE c.code = change.code;

    PERFORM pg_sleep(1); -- anything can happen here

    IF cnt = 0 THEN
        INSERT INTO categories VALUES (code, description);
    ELSE
        UPDATE categories c
        SET description = change.description
        WHERE c.code = change.code;
    END IF;
END
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION

```

---

Now let's run this function in two different sessions, almost simultaneously:

```

=> SELECT change('games', 'Games');

```

```

=> SELECT change('games', 'Games');

```

```

ERROR: duplicate key value violates unique constraint "categories_code_key"
DETAIL: Key (code)=(games) already exists.
CONTEXT: SQL statement "INSERT INTO categories VALUES (code, description)"
PL/pgSQL function change(text,text) line 11 at SQL statement

```

```

change
-----

```

```

(1 row)

```

---

A correct solution can be implemented using error handling:

```

=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
BEGIN
    LOOP
        UPDATE categories c
        SET description = change.description
        WHERE c.code = change.code;

        EXIT WHEN FOUND;
        PERFORM pg_sleep(1); -- for the demo

        BEGIN
            INSERT INTO categories VALUES (code, description);
            EXIT;
        EXCEPTION
            WHEN unique_violation THEN NULL;
        END;
    END LOOP;
END
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION

```

Let's check the result.

```

=> SELECT change('vynil', 'Vynil records');

```

```

=> SELECT change('vynil', 'Vynil records');

```

```

change
-----

```

```

(1 row)

```

change

(1 row)

Yes, now everything is correct.

But there is an easier way: you can use a special flavor of the INSERT command that attempts to insert a row and performs an update if a conflict occurs. Again, all you need is pure SQL.

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
VOLATILE LANGUAGE sql
BEGIN ATOMIC
    INSERT INTO categories VALUES (code, description)
    ON CONFLICT(code)
        DO UPDATE SET description = change.description;
END;

CREATE FUNCTION
```

Problem: make sure that the data is processed by one transaction at a time (at the Read Committed isolation level).

Using the same table, let's assume that the category sometimes requires a special single-threaded processing. The function can be declared as follows:

```
=> CREATE OR REPLACE FUNCTION process_cat(code text) RETURNS text
AS $$
BEGIN
    PERFORM c.code FROM categories c WHERE c.code = process_cat.code
    FOR UPDATE NOWAIT; -- try to lock the row immediately
    PERFORM pg_sleep(1); -- the processing itself
    RETURN 'The category has been processed';
EXCEPTION
    WHEN lock_not_available THEN
        RETURN 'Another process is already processing this category';
END
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION
```

Let's check that everything is correct:

```
=> SELECT process_cat('books');

| => SELECT process_cat('books');
|
|          process_cat
|-----
| Another process is already processing this category
| (1 row)
|
|          process_cat
|-----
| The category has been processed
| (1 row)
```

But this problem can also be solved without error handling if we use advisory locks:

```
=> CREATE OR REPLACE FUNCTION process_cat(code text) RETURNS text
AS $$
BEGIN
    IF pg_try_advisory_xact_lock(hashtext(code)) THEN
        PERFORM pg_sleep(1); -- the processing itself
        RETURN 'The category has been processed';
    ELSE
        RETURN 'Another transaction is already processing this category';
    END IF;
END
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION
```

Let's check the result:

```
=> SELECT process_cat('books');
```

```

=> SELECT process_cat('books');

           process_cat
-----
Another transaction is already processing this category
(1 row)

           process_cat
-----
The category has been processed
(1 row)

```

Let's see an example where we cannot do without error handling.

Problem: process a set of documents; a processing error of a particular document should not result in a general failure.

```

=> CREATE TYPE doc_status AS ENUM -- enumeration type
    ('READY', 'ERROR', 'PROCESSED');

```

CREATE TYPE

```

=> CREATE TABLE documents(
    id integer,
    version integer,
    status doc_status,
    message text
);

```

CREATE TABLE

```

=> INSERT INTO documents(id, version, status)
SELECT id, 1, 'READY' FROM generate_series(1,100) id;

```

INSERT 0 100

A procedure that processes a single document can sometimes result in an error:

```

=> CREATE PROCEDURE process_one_doc(id integer)
AS $$
BEGIN
    UPDATE documents d
    SET version = version + 1
    WHERE d.id = process_one_doc.id;
    -- processing can take a while
    IF random() < 0.05 THEN
        RAISE EXCEPTION 'Catastrophic failure';
    END IF;
END
$$ LANGUAGE plpgsql;

```

CREATE PROCEDURE

Now let's create a procedure that processes all documents. It loops through the documents to process them one by one and catches an error if required.

Note that transactions are committed outside of the block that contains the EXCEPTION section.

```

=> CREATE PROCEDURE process_docs()
AS $$
DECLARE
    doc record;
BEGIN
    FOR doc IN (SELECT id FROM documents WHERE status = 'READY')
    LOOP
        BEGIN
            CALL process_one_doc(doc.id);

            UPDATE documents d
            SET status = 'PROCESSED'
            WHERE d.id = doc.id;
        EXCEPTION
            WHEN others THEN
                UPDATE documents d
                SET status = 'ERROR', message = sqlerrm
                WHERE d.id = doc.id;
        END;
        COMMIT; -- there is a separate transaction for each document
    END LOOP;
END

$$ LANGUAGE plpgsql;

```

CREATE PROCEDURE

You can set up a similar processing using a function, but then all documents will be handled within a single common transaction, which can be a problem if processing takes a long time. This question is discussed at length in the DEV2 course.

Let's check the result:

```

=> CALL process_docs();

CALL

=> SELECT d.status, d.version, count(*)::integer
FROM documents d
GROUP BY d.status, d.version;

```

status	version	count
ERROR	1	3
PROCESSED	2	97

(2 rows)

As you can see, some of the documents have not been processed, but it has not affected the processing of the others.

It is convenient that the information about the occurred errors is stored in the table itself:

```

=> SELECT * FROM documents d WHERE d.status = 'ERROR';

```

id	version	status	message
18	1	ERROR	Catastrophic failure
47	1	ERROR	Catastrophic failure
53	1	ERROR	Catastrophic failure

(3 rows)

Please note once again that if an error occurs, the changes are rolled back to the savepoint at the beginning of the block; that's why documents with the ERROR status still have version 1.

The search for an error handler is performed “inside out” according to block nesting and routine call stack

An implicit savepoint is set at the beginning of the block that contains EXCEPTION; if an error occurs, a rollback to this savepoint is performed

An unhandled error aborts the transaction; the error message is passed to the client and registered in the server log

Error handling incurs overhead



1. An attempt to put in the same author several times when adding a book causes an error.  
Modify the `add_book` function: catch the unique constraint violation error and produce an error with a meaningful message instead.  
Try it out in the application.

1. To determine the name of the error that has to be caught, catch all errors (WHEN OTHERS) and display the required information (by raising another error with the corresponding text).

Then remember to replace WITH OTHERS with a specific error: let all other error types be handled at a higher level if there is no opportunity to do anything useful in this particular position in the code.

(In a real environment, unique constraint violations should not be handled either: it is better to forbid entering the same author twice at the application level.)

## 1. Handling Duplicate Author Names When Adding Books

```
=> CREATE OR REPLACE FUNCTION add_book(title text, authors integer[])
RETURNS integer
AS $$
DECLARE
    book_id integer;
    id integer;
    seq_num integer := 1;
BEGIN
    INSERT INTO books(title)
        VALUES(title)
        RETURNING books.book_id INTO book_id;
    FOREACH id IN ARRAY authors LOOP
        INSERT INTO authorship(book_id, author_id, seq_num)
            VALUES (book_id, id, seq_num);
        seq_num := seq_num + 1;
    END LOOP;
    RETURN book_id;
EXCEPTION
    WHEN unique_violation THEN
        RAISE EXCEPTION 'The same author cannot be specified twice';
END
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION
```

1. Some languages use the construct `try ... catch ... finally ...`, where `try` corresponds to `BEGIN`, `catch` corresponds to `EXCEPTION`, and the statements located in the `finally` block are always triggered regardless of whether an exception has occurred and whether it has been processed by the `catch` block. Find a way to achieve a similar behavior in PL/pgSQL.
2. Compare the call stacks returned by `GET STACKED DIAGNOSTICS` with `pg_exception_context` and `GET [CURRENT] DIAGNOSTICS` with `pg_context`.
3. Create a function `getstack` that returns the current call stack as a string array. The `getstack` function itself must not be a part of the stack.

1. The easiest way to do it is to simply repeat the `finally` statements in several places. But you should try to come up with a solution that avoids code duplication.

2. First, refer to the documentation:

<https://postgrespro.com/docs/postgresql/16/plpgsql-statements#PLPGSQL-STATEMENTS-DIAGNOSTICS>

<https://postgrespro.com/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-EXCEPTION-DIAGNOSTICS>



## 1. Try-catch-finally

```
=> CREATE DATABASE plpgsql_exceptions;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_exceptions
```

You are now connected to database "plpgsql\_exceptions" as user "student".

The tricky part is the 'finally' statements. They must always execute, even if there is an exception catch (in the EXCEPTION block).

The solution can use two nested blocks and a dummy exception that is thrown when the inner block is executed as expected. This makes it possible to put finally statements in one place: the external block error handler.

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'try statements';
        --
        RAISE NOTICE '...no exception';
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'catch statements';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'finally statements';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END
$$;
```

```
NOTICE: try statements
NOTICE: ...no exception
NOTICE: finally statements
DO
```

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'try statements';
        --
        RAISE NOTICE '...exception that is processed';
        RAISE no_data_found;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'catch statements';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'finally statements';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END
$$;
```

```
NOTICE: try statements
NOTICE: ...exception that is processed
NOTICE: catch statements
NOTICE: finally statements
DO
```

```

=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'try statements';
        --
        RAISE NOTICE '...exception not handled';
        RAISE division_by_zero;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'catch statements';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'finally statements';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END
$$;

NOTICE: try statements
NOTICE: ...exception not handled
NOTICE: finally statements
ERROR: division_by_zero
CONTEXT: PL/pgSQL function inline_code_block line 7 at RAISE

```

But in the proposed solution, all changes made in the block are always rolled back, so it is not suitable for commands that change the database state. Also, don't forget about the exception handling overhead. This task is just an exercise.

## 2. GET DIAGNOSTICS

```

=> DO $$
DECLARE
    ctx text;
BEGIN
    RAISE division_by_zero; -- line 5
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS ctx := PG_EXCEPTION_CONTEXT;
        RAISE NOTICE E'stacked =\n%', ctx;
        GET CURRENT DIAGNOSTICS ctx := PG_CONTEXT; -- line 10
        RAISE NOTICE E'current =\n%', ctx;
END
$$;

NOTICE: stacked =
PL/pgSQL function inline_code_block line 5 at RAISE
NOTICE: current =
PL/pgSQL function inline_code_block line 10 at GET DIAGNOSTICS
DO

```

GET STACKED DIAGNOSTICS provides the stack of calls that resulted in the error.

GET CURRENT DIAGNOSTICS provides the current stack of calls.

## 3. Call Stack as an Array

The function itself:

```

=> CREATE FUNCTION getstack() RETURNS text[]
AS $$
DECLARE
    ctx text;
BEGIN
    GET DIAGNOSTICS ctx := PG_CONTEXT;
    RETURN (regexp_split_to_array(ctx, E'\n'))[2:];
END
$$ VOLATILE LANGUAGE plpgsql;

```

CREATE FUNCTION

To test it, create several functions that call each other:

```

=> CREATE FUNCTION foo() RETURNS integer
AS $$
BEGIN
    RETURN bar();
END
$$ VOLATILE LANGUAGE plpgsql;

```

CREATE FUNCTION

```
=> CREATE FUNCTION bar() RETURNS integer
AS $$
BEGIN
    RETURN baz();
END
$$ VOLATILE LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> CREATE FUNCTION baz() RETURNS integer
AS $$
BEGIN
    RAISE NOTICE '%', getstack();
    RETURN 0;
END
$$ VOLATILE LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> SELECT foo();
```

```
NOTICE: {"PL/pgSQL function baz() line 3 at RAISE","PL/pgSQL function bar() line 3 at
RETURN","PL/pgSQL function foo() line 3 at RETURN"}
foo
-----
  0
(1 row)
```

```
=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
=> DROP DATABASE plpgsql_exceptions;
```

DROP DATABASE