Community Experience Distilled

Extend PostgreSQL using PostgreSQL server programming to create, test, debug, and optimize a range of user-defined functions in your favorite programming language.

What you will learn from this book

- Write functions in the built-in PL/pgSQL language or your language of choice
- Define and use user-defined functions (UDF)
- Scale your data with PL/Proxy
- Extend PostgreSQL to overcome the limitations of classical SQL servers
- Debug using NOTIFY and using PL/pgSQL debugging extensions
- Discover the new features in PostgreSQL 9.4 such as event triggers and other improvements relevant to server-side programming
- Fully integrate the database layer into your development

Who this book is written for

This book is for moderate to advanced PostgreSQL database professionals who wish to extend PostgreSQL, utilizing the most updated features of PostgreSQL 9.4. For a better understanding of this book, familiarity with writing SQL, a basic idea of query tuning, and some coding experience in your preferred language is expected.


Free Sample
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 "What Is a PostgreSQL Server?"
- A synopsis of the book’s content

About the Authors

Usama Dar is a seasoned software developer and architect. During his 14 years' career, he has worked extensively with PostgreSQL and other database technologies. He worked on PostgreSQL internals extensively while he was working for EnterpriseDB. Currently, he lives in Munich where he works for Huawei's European Research Center. He designs the next generation of high-performance database systems based on open source technologies, such as PostgreSQL, which are used under high workloads and strict performance requirements.

Hannu Krosing was a PostgreSQL user before it was rewritten to use SQL as its main query language in 1995. Therefore, he has both the historic perspective of its development, as well as almost 20 years of experience in using it to solve various real-life problems.

He was the first database administrator and database architect at Skype, where he invented the sharding language PL/Proxy that allows you to scale the user database in order to work with billions of users.

After he left Skype at the end of 2006—about a year after it was bought by eBay—he has been working as a PostgreSQL consultant with 2ndQuadrant, the premier PostgreSQL consultancy with a global reach and local presence in most parts of the world.

He has coauthored PostgreSQL 9 Administration Cookbook, Packt Publishing, together with one of the main PostgreSQL developers, Simon Riggs.

I want to sincerely thank my wife, Evelyn, for her support while writing this book.
Jim Mlodgenski is the CTO of OpenSCG, a professional services company focused on leveraging open source technologies for strategic advantage. He was formerly the CEO of StormDB, a database cloud company focused on horizontal scalability. Prior to StormDB, he has held highly technical roles at Cirrus Technology, Inc., EnterpriseDB, and Fusion Technologies.

Jim is also a fervent advocate of PostgreSQL. He is on the board of the United States PostgreSQL Association as well as a part of the organizing teams of the New York PostgreSQL User Group and Philadelphia PostgreSQL User Group.

Kirk Roybal has been an active member of the PostgreSQL community since 1998. He has helped organize user groups in Houston, Dallas, and Bloomington, IL. He has mentored many junior database administrators and provided cross-training to senior database engineers. He has provided solutions using PostgreSQL for reporting, business intelligence, data warehousing, applications, and development support.

He saw the scope of PostgreSQL when his first small-scale business customer asked for a web application. At that time, competitive database products were either extremely immature or cost prohibitive.

Kirk has stood by his choice of PostgreSQL for many years now. His expertise is founded on keeping up with features and capabilities as they become available.

Writing a book has been a unique experience for me. Many people fantasize about it, few start one, and even fewer get to publication. I am proud to be part of a team that actually made it to the book shelf (which itself is a diminishing breed). Thanks to Sarah Cullington from Packt Publishing for giving me a chance to participate in the project. I believe that the PostgreSQL community will be better served by this information, and I hope that they receive this as a reward for the time that they have invested in me over the years.

A book only has the value that the readers give it. Thank you to the PostgreSQL community for all the technical, personal, and professional development help you have provided. The PostgreSQL community is a great bunch of people, and I have enjoyed the company of many of them. I hope to contribute more to this project in the future, and I hope you find my contributions as valuable as I find yours.

Thank you to my family for giving me a reason to succeed and for listening to the gobbledygook and nodding appreciatively.

Have you ever had your family ask you what you were doing and answered them with a function? Try it. No, then again, don't try it. They may just have you involuntarily checked in somewhere.
PostgreSQL Server Programming
Second Edition

This fascinating guide to server programming will take your skills of PostgreSQL to a whole new level. A step-by-step approach with illuminating examples will educate you about the full range of possibilities. You will understand the extension framework of PostgreSQL and leverage it in ways you haven't even invented yet. You will learn how to write functions and create your own data types, all in your favorite programming language. It is a step-by-step tutorial, with plenty of tips and tricks to kick-start server programming.

What This Book Covers

Chapter 1, *What Is a PostgreSQL Server?*, introduces you to the PostgreSQL server and will set the tone for the rest of the book. It introduces you to the ways in which a PostgreSQL server is extendible, and shows you that it can be treated as a complete software development framework instead of just a database server.

Chapter 2, *Server Programming Environments*, elaborates that PostgreSQL is built to handle user needs, but more importantly, it is built not to change underneath users in the future. It will touch upon the environments and will highlight some of the important things to be kept in mind when programming on the server in PostgreSQL.

Chapter 3, *Your First PL/pgSQL Function*, builds the foundations by demonstrating how to write simple PL/pgSQL functions.

Chapter 4, *Returning Structured Data*, builds on the knowledge of writing PL/pgSQL functions and demonstrates how to write functions that return a set of values such as rows, arrays, and cursors.

Chapter 5, *PL/pgSQL Trigger Functions*, discusses how to write PL/pgSQL functions that are used to write trigger logic. It also discusses the various types of triggers available in PostgreSQL and the options that a database developer has when writing such functions.

Chapter 6, *PostgreSQL Event Triggers*, discusses PostgreSQL's event trigger functionality. Event triggers are fired when running a DDL operation on a table. This chapter discusses the various possibilities and options of creating event triggers and their limitations in PostgreSQL.

Chapter 7, *Debugging PL/pgSQL*, elaborates on how to debug PL/pgSQL's stored procedures and functions in PostgreSQL. This chapter explains how to install the debugger plugin and use the pgAdmin debugger console.
Chapter 8, Using Unrestricted Languages, explains the differences between restricted and unrestricted PostgreSQL languages. This chapter uses PL/Python as an example and demonstrates the examples of both restricted and unrestricted functions in PL/Python.

Chapter 9, Writing Advanced Functions in C, explains how to extend PostgreSQL by writing user-defined functions (UDFs) in C.

Chapter 10, Scaling Your Database with PL/Proxy, explains the use of a special programming language in PostgreSQL called PL/Proxy and how to use it in order to partition and shard your database.

Chapter 11, PL/Perl – Perl Procedural Language, discusses a popular PL language in PostgreSQL called PL/Perl. This chapter uses some simple examples to demonstrate how you can use Perl to write database functions.

Chapter 12, PL/Tcl – Tcl Procedural Language, discusses Tcl as a language of choice when writing database functions. It discusses the pros and cons of using Tcl in the database.

Chapter 13, Publishing Your Code as PostgreSQL Extensions, discusses how to package and distribute the PostgreSQL extensions. Well-packaged extensions can be easily distributed and installed by other users. This chapter also introduces you to the PostgreSQL Extension Network (PGXN) and shows you how to use it to get the extensions published by other developers.

Chapter 14, PostgreSQL as an Extensible RDBMS, discusses more extensibility options in PostgreSQL, such as creating new data types, operators, and index methods.
What Is a PostgreSQL Server?

If you think that a PostgreSQL Server is just a storage system and the only way to communicate with it is by executing SQL statements, you are limiting yourself tremendously. That is, you are using just a tiny part of the database's features.

A PostgreSQL Server is a powerful framework that can be used for all kinds of data processing, and even some non-data server tasks. It is a server platform that allows you to easily mix and match functions and libraries from several popular languages.

Consider this complicated, multilanguage sequence of work:

- Call a string parsing function in Perl
- Convert the string to XSLT and process the result using JavaScript
- Ask for a secure stamp from an external timestamping service, such as http://guardtime.com/, using their SDK for C
- Write a Python function to digitally sign the result

This multilanguage sequence of work can be implemented as a series of simple function calls using several of the available server programming languages. The developer who needs to accomplish all this work can just call a single PostgreSQL function without the need to be aware of how the data is being passed between languages and libraries:

```sql
SELECT convert_to_xslt_and_sign(raw_data_string);
```
What Is a PostgreSQL Server?

In this book, we will discuss several facets of PostgreSQL Server programming. PostgreSQL has all of the native server-side programming features available in most larger database systems such as triggers, which are automated actions invoked automatically each time data is changed. However, it has uniquely deep abilities to override the built-in behavior down to very basic operators. This unique PostgreSQL ability comes from its catalog-driven design, which stores information about data types, functions, and access methods. The ability of PostgreSQL to load user-defined functions via dynamic loading makes it rapidly changeable without having to recompile the database itself. There are several things you can do with this flexibility of customization. Some examples of this customization include the following:

- Writing **user-defined functions (UDF)** to carry out complex computations
- Adding complicated constraints to make sure that the data in the server meets guidelines
- Creating triggers in many languages to make related changes to other tables, audit changes, forbid the action from taking place if it does not meet certain criteria, prevent changes to the database, enforce and execute business rules, or replicate data
- Defining new data types and operators in the database
- Using the geography types defined in the PostGIS package
- Adding your own index access methods for either the existing or new data types, making some queries much more efficient

What sort of things can you do with these features? There are limitless possibilities, such as the ones listed here:

- Write data extractor functions to get just the interesting parts from structured data, such as XML or JSON, without needing to ship the whole, possibly huge, document to the client application.
- Process events asynchronously, such as sending mails without slowing down the main application. You can create a mail queue for changes to user information, populated by a trigger. A separate mail-sending process can consume this data whenever it is notified by an application process.
- Implement a new data type to custom hash the passwords.
- Write functions, which provide inside information about the server, for example, cache contents, table-wise lock information, or the SSL certificate information of a client connection for a monitoring dashboard.
The rest of this chapter is presented as a series of descriptions of common data management tasks, showing how they can be solved in a robust and elegant way via server programming.

The samples in this chapter are all tested to work, but they come with minimal commentary. They are used here just to show you various things that server programming can accomplish. The techniques that are described will be explained thoroughly in later chapters.

**Why program in the server?**

Developers program their code in a number of different languages, and it can be designed to run just about anywhere. When writing an application, some people follow the philosophy that as much of the logic as possible for the application should be pushed to the client. We see this in the explosion of applications leveraging JavaScript inside browsers. Others like to push the logic into the middle tier, with an application server handling the business rules. These are all valid ways to design an application, so why will you want to program in the database server?

Let's start with a simple example. Many applications include a list of customers who have a balance in their account. We'll use this sample schema and data:

```sql
CREATE TABLE accounts(owner text, balance numeric, amount numeric);
INSERT INTO accounts VALUES ('Bob',100);
INSERT INTO accounts VALUES ('Mary',200);
```

When using a database, the most common way to interact with it, is to use SQL queries. If you want to move 14 dollars from Bob's account to Mary's account with simple SQL, you can do so using the following:

```sql
UPDATE accounts SET balance = balance - 14.00 WHERE owner = 'Bob';
UPDATE accounts SET balance = balance + 14.00 WHERE owner = 'Mary';
```
However, you also have to make sure that Bob actually has enough money (or credit) in his account. Note that if anything fails, then none of the transactions will happen. In an application program, this is how the preceding code snippet will be modified:

```
BEGIN;
SELECT amount FROM accounts WHERE owner = 'Bob' FOR UPDATE;
-- now in the application check that the amount is actually bigger
-- than 14
UPDATE accounts SET amount = amount - 14.00 WHERE owner = 'Bob';
UPDATE accounts SET amount = amount + 14.00 WHERE owner = 'Mary';
COMMIT;
```

Did Mary actually have an account? If she did not, the last `UPDATE` command will succeed by updating zero rows. If any of the checks fail, you should do `ROLLBACK` instead of `COMMIT`. Once you have done all this for all the clients that transfer money, a new requirement will invariably arrive. Perhaps, the minimum amount that can be transferred is now 5.00. You will need to revisit the code in all your clients again.

So, what can you do to make all of this more manageable, secure, and robust? This is where server programming, executing code on the database server itself, can help. You can move the computations, checks, and data manipulations entirely into a UDF on the server. This not only ensures that you have only one copy of operation logic to manage, but also makes things faster by not requiring several round trips between the client and the server. If required, you can also make sure that only the essential information is given out from the database. For example, there is no business for most client applications to know how much money Bob has in his account. Mostly, they only need to know whether there is enough money to make the transfer, or to be more specific, whether the transaction succeeded.

**Using PL/pgSQL for integrity checks**

PostgreSQL includes its own programming language named PL/pgSQL that is aimed to integrate easily with SQL commands. PL stands for procedural language, and this is just one of the many languages available for writing server code. pgSQL is the shorthand for PostgreSQL.

Unlike basic SQL, PL/pgSQL includes procedural elements, such as the ability to use the `if/then/else` statements and loops. You can easily execute SQL statements, or even loop over the result of a SQL statement in the language.
The integrity checks needed for the application can be done in a PL/pgSQL function that takes three arguments: names of the payer and the recipient and the amount to be paid. This sample also returns the status of the payment:

```sql
CREATE OR REPLACE FUNCTION transfer(
    i_payer text,
    i_recipient text,
    i_amount numeric(15,2))
RETURNS text
AS
$$
DECLARE
    payer_bal numeric;
BEGIN
    SELECT balance INTO payer_bal
    FROM accounts
    WHERE owner = i_payer FOR UPDATE;
    IF NOT FOUND THEN
        RETURN 'Payer account not found';
    END IF;
    IF payer_bal < i_amount THEN
        RETURN 'Not enough funds';
    END IF;
    UPDATE accounts
    SET balance = balance + i_amount
    WHERE owner = i_recipient;
    IF NOT FOUND THEN
        RETURN 'Recipient does not exist';
    END IF;
    UPDATE accounts
    SET balance = balance - i_amount
    WHERE owner = i_payer;
    RETURN 'OK';
END;
$$ LANGUAGE plpgsql;
```
What Is a PostgreSQL Server?

Here are a few examples of the usage of this function, assuming that you haven’t executed the previously proposed `UPDATE` statements yet:

```sql
postgres=# SELECT * FROM accounts;
  owner | balance  
-------+---------
   Bob  |    100  
  Mary  |    200  
(2 rows)

postgres=# SELECT transfer('Bob','Mary',14.00);
transfer
----------
   OK
(1 row)

postgres=# SELECT * FROM accounts;
  owner | balance  
-------+---------
  Mary  |  214.00  
   Bob  |   86.00  
(2 rows)
```

Your application will need to check the return code and decide how to handle these errors. As long as it is written to reject any unexpected value, you can extend this function to do more checking, such as the minimum transferrable amount, and you can be sure it will be prevented. The following three errors can be returned:

```sql
postgres=# SELECT transfer('Fred','Mary',14.00);
transfer
-------------------------
Payer account not found
(1 row)

postgres=# SELECT transfer('Bob','Fred',14.00);
transfer
--------------------------
Recipient does not exist
(1 row)

postgres=# SELECT transfer('Bob','Mary',500.00);
transfer
----------------
Not enough funds
(1 row)
```
For these checks to always work, you will need to make all the transfer operations go through the function, rather than manually changing the values with SQL statements. One way to achieve this, is by revoking update privileges from users and from a user with higher privileges that define the transfer function with `SECURITY DEFINER`. This will allow the restricted users to run the function as if they have higher privileges similar to the function's creator.

**About this book's code examples**

The sample output shown here has been created with the `psql` utility of PostgreSQL, usually running on a Linux system. Most of the code will work the same way if you are using a GUI utility such as `pgAdmin3` to access the server instead. Take an example of the following line of code:

```
postgres=# SELECT 1;
```

The `postgres=#` part is the prompt shown by the `psql` command.

The examples in this book have been tested using PostgreSQL 9.3. They will probably work on PostgreSQL Version 8.3 and later. There haven't been many major changes to how server programming happens in the last few versions of PostgreSQL. The syntax has become stricter over time to reduce the possibility of mistakes in the server programming code. Due to the nature of these changes, most code from newer versions will still run on the older ones, unless it uses very new features. However, the older code can easily fail to run due to one of the newly enforced restrictions.

**Switching to the expanded display**

When using the `psql` utility to execute a query, PostgreSQL normally outputs the result using vertically aligned columns:

```
$ psql -c "SELECT 1 AS test"
test
-------
  1
(1 row)
```

```
$ psql
psql (9.3.2)
```
What Is a PostgreSQL Server?

Type "help" for help.

```
postgres=# SELECT 1 AS test;
     test
    ------
       1
(1 row)
```

You can tell when you're seeing a regular output because it will end up showing the number of rows.

This type of output is hard to fit into the text of a book such as this. It's easier to print the output from what the program calls the expanded display, which breaks each column into a separate line. You can switch to the expanded display using either the -x command-line switch or by sending \x to the psql program. Here's an example of using each of these:

$ psql -x -c "SELECT 1 AS test"
- [ RECORD 1 ]
test | 1

$ psql
psql (9.3.2)
Type "help" for help.
postgres=# \x
Expanded display is on.
postgres=# SELECT 1 AS test;
- [ RECORD 1 ]
test | 1

Notice how the expanded output doesn't show the row count and numbers each output row. To save space, not all of the examples in the book will show the expanded output being turned on. You can normally tell which type you can see, by differences such as whether you're seeing rows or RECORD. The expanded mode will normally be preferred when the output of the query is too wide to fit into the available width of the book. It is a good idea to set the expanded mode to auto. This will automatically switch to expanded mode for tables with a lot of columns. You can turn on the expanded mode using \x auto:

```
postgres=# \x auto
Expanded display is used automatically.
```
Moving beyond simple functions

Server programming can mean a lot of different things. Server programming is not just about writing server functions. There are many other things you can do in the server, which can be considered as programming.

Data comparisons using operators

For more complex tasks, you can define your own types, operators, and casts from one type to another, letting you actually compare apples and oranges.

As shown in the next example, you can define the type `fruit_qty` for fruit-with-quantity and then teach PostgreSQL to compare apples and oranges, say to make one orange to be worth 1.5 apples, in order to convert apples to oranges:

```sql
postgres=# CREATE TYPE FRUIT_QTY as (name text, qty int);

postgres=# SELECT '("APPLE", 3)':''::FRUIT_QTY;
   fruit_qty
  -----------
("APPLE",3)
   (1 row)

CREATE FUNCTION fruit_qty_larger_than(left_fruit FRUIT_QTY,
                                      right_fruit FRUIT_QTY)
RETURNS BOOL
AS $$
BEGIN
  IF (left_fruit.name = 'APPLE' AND right_fruit.name = 'ORANGE')
  THEN
    RETURN left_fruit.qty > (1.5 * right_fruit.qty);
  END IF;
  IF (left_fruit.name = 'ORANGE' AND
      right_fruit.name = 'APPLE')
  THEN
    RETURN (1.5 * left_fruitqty) > right_fruit.qty;
  END IF;
  RETURN left_fruit.qty > right_fruit.qty;
END;
$$
LANGUAGE plpgsql;

postgres=# SELECT fruit_qty_larger_than('("APPLE", 3)::*FRUIT_QTY','("ORANGE", 2)::*FRUIT_QTY);
```
What Is a PostgreSQL Server?

```sql
fruit_qty_larger_than
-----------------------
t
(1 row)
```

```sql
postgres=# SELECT fruit_qty_larger_than('("APPLE", 4)':'FRUIT_QTY','("ORANGE", 2)':'FRUIT_QTY);
fruit_qty_larger_than
-----------------------
t
(1 row)
```

```sql
CREATE OPERATOR > (
    leftarg = FRUIT_QTY,
    rightarg = FRUIT_QTY,
    procedure = fruit_qty_larger_than,
    commutator = >
);
```

```sql
postgres=# SELECT '("ORANGE", 2)':'FRUIT_QTY > '("APPLE", 2)':'FRUIT_QTY;
?column?
--------
t
(1 row)
```

```sql
postgres=# SELECT '("ORANGE", 2)':'FRUIT_QTY > '("APPLE", 3)':'FRUIT_QTY;
?column?
--------	f
(1 row)
```

Managing related data with triggers

Server programming can also mean setting up automated actions (triggers), so that some operations in the database cause some other things to happen as well. For example, you can set up a process where making an offer on some items is automatically reserved to them being in the stock table.

So, let’s create a fruit stock table, as shown here:

```sql
CREATE TABLE fruits_in_stock {
    name text PRIMARY KEY,
```
in_stock integer NOT NULL,
reserved integer NOT NULL DEFAULT 0,
CHECK (in_stock between 0 and 1000 ),
CHECK (reserved <= in_stock)
);

The CHECK constraints make sure that some basic rules are followed: you can't have
more than 1000 fruits in stock (they'll probably go bad), you can't have a negative
stock, and you can't reserve more than what you have. The fruit_offer table
will contain the fruits from stock which are on offer. When we insert a row in the
fruit_offer table. The offered amount will be reserved in the stock table as shown:

CREATE TABLE fruit_offer {
   offer_id serial PRIMARY KEY,
   recipient_name text,
   offer_date timestamp default current_timestamp,
   fruit_name text REFERENCES fruits_in_stock,
   offered_amount integer
};

The offer table has an ID for the offer (so you can distinguish between offers later),
recipient, date, offered fruit name, and offered amount.

In order to automate the reservation management, you first need a TRIGGER function,
which implements the management logic:

CREATE OR REPLACE FUNCTION reserve_stock_on_offer () RETURNS trigger
AS $$
BEGIN
   IF TG_OP = 'INSERT' THEN
      UPDATE fruits_in_stock
      SET reserved = reserved + NEW.offered_amount
      WHERE name = NEW.fruit_name;
   ELSIF TG_OP = 'UPDATE' THEN
      UPDATE fruits_in_stock
      SET reserved = reserved - OLD.offered_amount
                      + NEW.offered_amount
      WHERE name = NEW.fruit_name;
   ELSIF TG_OP = 'DELETE' THEN
      UPDATE fruits_in_stock
      SET reserved = reserved - OLD.offered_amount
      WHERE name = OLD.fruit_name;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

You have to tell PostgreSQL to call this function each and every time the offer row is changed:

```sql
CREATE TRIGGER manage_reserve_stock_on_offer_change
AFTER INSERT OR UPDATE OR DELETE ON fruit_offer
FOR EACH ROW EXECUTE PROCEDURE reserve_stock_on_offer();
```

After this, we are ready to test the functionality. First, we will add some fruits to our stock:

```sql
INSERT INTO fruits_in_stock VALUES('APPLE',500);
INSERT INTO fruits_in_stock VALUES('ORANGE',500);
```

Then, we will check the stock (using the expanded display):

```sql
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM fruits_in_stock;
-[
 RECORD 1 ]-----
  name     | APPLE
  in_stock | 500
  reserved | 0
-[
 RECORD 2 ]-----
  name     | ORANGE
  in_stock | 500
  reserved | 0
```

Next, let's make an offer of 100 apples to Bob:

```sql
postgres=# INSERT INTO fruit_offer(recipient_name,fruit_name,offered_amount)
   VALUES('Bob','APPLE',100);
INSERT 0 1
postgres=# SELECT * FROM fruit_offer;
-[
 RECORD 1 ]---------------------------
   offer_id | 1
   recipient_name | Bob
   offer_date    | 2013-01-25 15:21:15.281579
   fruit_name   | APPLE
   offered_amount | 100
```

On checking the stock, we see that indeed 100 apples are reserved, as shown in the following code snippet:

```sql
postgres=# SELECT * FROM fruits_in_stock;
-[
 RECORD 1 ]-----
  name     | ORANGE
```
If we change the offered amount, the reserved amount also changes:

```
postgres=# UPDATE fruit_offer SET offered_amount = 115
WHERE offer_id = 1;
UPDATE 1
postgres=# SELECT * FROM fruits_in_stock;
-[ RECORD 1 ]----
name     | ORANGE
in_stock | 500
reserved | 0
-[ RECORD 2 ]----
name     | APPLE
in_stock | 500
reserved | 115
```

We also get some extra benefits. First, because of the constraint on the stock table, you can't sell the reserved apples:

```
postgres=# UPDATE fruits_in_stock SET in_stock = 100
WHERE name = 'APPLE';
ERROR:  new row for relation "fruits_in_stock" violates check constraint "fruits_in_stock_check"
DETAIL:  Failing row contains (APPLE, 100, 115).
```

More interestingly, you also can't reserve more than you have, even though the constraints are on another table:

```
postgres=# UPDATE fruit_offer SET offered_amount = 1100
WHERE offer_id = 1;
ERROR:  new row for relation "fruits_in_stock" violates check constraint "fruits_in_stock_check"
DETAIL:  Failing row contains (APPLE, 500, 1100).
```

CONTEXT:  SQL statement "UPDATE fruits_in_stock
    SET reserved = reserved - OLD.offered_amount
    + NEW.offered_amount
    WHERE name = NEW.fruit_name"
PL/pgSQL function reserve_stock_on_offer() line 8 at SQL statement
When you finally delete the offer, the reservation is released:

```sql
postgres=# DELETE FROM fruit_offer WHERE offer_id = 1;
DELETE 1
postgres=# SELECT * FROM fruits_in_stock;
- [ RECORD 1 ]----
  name   | ORANGE
 in_stock | 500
 reserved | 0
- [ RECORD 2 ]----
  name   | APPLE
 in_stock | 500
 reserved | 0
```

In a real system, you probably will archive the old offer before deleting it.

**Auditing changes**

If you need to know who did what to the data and when it was done, one way to find out is to log every action that is performed in an important table. In PostgreSQL 9.3, you can also audit the data definition language (DDL) changes to the database using event triggers. We will learn more about this in the later chapters.

There are at least two equally valid ways to perform data auditing:

- Using auditing triggers
- Allowing tables to be accessed only through functions and auditing inside these functions

Here, we will take a look at a minimal number of examples for both the approaches.

First, let's create the tables:

```sql
CREATE TABLE salaries(
  emp_name text PRIMARY KEY,
  salary integer NOT NULL
);

CREATE TABLE salary_change_log(
  changed_by text DEFAULT CURRENT_USER,
  changed_at timestamp DEFAULT CURRENT_TIMESTAMP,
  salary_op text,
  emp_name text,
  old_salary integer,
  new_salary integer
);```
new_salary integer
);
REVOKE ALL ON salary_change_log FROM PUBLIC;
GRANT ALL ON salary_change_log TO managers;

You don't generally want your users to be able to change audit logs, so only grant
the managers the right to access these. If you plan to let users access the salary table
directly, you should put a trigger on it for auditing:

CREATE OR REPLACE FUNCTION log_salary_change () RETURNS trigger AS $$
BEGIN
  IF TG_OP = 'INSERT' THEN
    INSERT INTO salary_change_log(salary_op,emp_name,new_salary)
    VALUES (TG_OP,NEW.emp_name,NEW.salary);
  ELSIF TG_OP = 'UPDATE' THEN
    INSERT INTO salary_change_log(salary_op,emp_name,old_salary,new_salary)
    VALUES (TG_OP,NEW.emp_name,OLD.salary,NEW.salary);
  ELSIF TG_OP = 'DELETE' THEN
    INSERT INTO salary_change_log(salary_op,emp_name,old_salary)
    VALUES (TG_OP,NEW.emp_name,OLD.salary);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

CREATE TRIGGER audit_salary_change
AFTER INSERT OR UPDATE OR DELETE ON salaries
  FOR EACH ROW EXECUTE PROCEDURE log_salary_change ();

Now, let's test out some salary management:

postgres=# INSERT INTO salaries values('Bob',1000);
INSERT 0 1
postgres=# UPDATE salaries SET salary = 1100
  WHERE emp_name = 'Bob';
UPDATE 1
postgres=# INSERT INTO salaries VALUES('Mary',1000);
INSERT 0 1
postgres=# UPDATE salaries SET salary = salary + 200;
UPDATE 2
postgres=# SELECT * FROM salaries;
- [ RECORD 1 ] --
  emp_name | Bob
  salary   | 1300
What Is a PostgreSQL Server?

- [ RECORD 2 ]--
  emp_name | Mary
  salary   | 1200

Each one of these changes is saved into the salary change log table for auditing purposes:

postgres=# SELECT * FROM salary_change_log;
- [ RECORD 1 ]--------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.311299
salary_op  | INSERT
emp_name   | Bob
old_salary |
new_salary  | 1000
- [ RECORD 2 ]--------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.313405
salary_op  | UPDATE
emp_name   | Bob
old_salary | 1000
new_salary | 1100
- [ RECORD 3 ]--------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.314208
salary_op  | INSERT
emp_name   | Mary
old_salary |
new_salary  | 1000
- [ RECORD 4 ]--------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.314903
salary_op  | UPDATE
emp_name   | Bob
old_salary | 1100
new_salary | 1300
- [ RECORD 5 ]--------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.314903
salary_op  | UPDATE
emp_name   | Mary
old_salary | 1000
new_salary | 1200
On the other hand, you may not want anybody to have direct access to the salary table, in which case you can perform the `REVOKE` command. The following command will revoke all privileges from `PUBLIC`:

```
REVOKE ALL ON salaries FROM PUBLIC;
```

Also, give users access to only two functions: the first function is for any user taking a look at salaries and the other function can be used to change salaries, which is available only to managers.

The functions will have all the access to the underlying tables because they are declared as `SECURITY DEFINER`, which means that they run with the privileges of the user who created them.

This is how the salary lookup function will look:

```
CREATE OR REPLACE FUNCTION get_salary(text)
RETURNS integer
AS $$
-- if you look at other people's salaries, it gets logged
INSERT INTO salary_change_log(salary_op,emp_name,new_salary)
SELECT 'SELECT',emp_name,salary
FROM salaries
WHERE upper(emp_name) = upper($1)
    AND upper(emp_name) != upper(CURRENT_USER);
-- don't log select of own salary
-- return the requested salary
SELECT salary FROM salaries WHERE upper(emp_name) = upper($1);
$$ LANGUAGE SQL SECURITY DEFINER;
```

Notice that we implemented a soft-security approach, where you can look up other people's salaries, but you have to do it responsibly, that is, only when you need to, as your manager will know that you have checked.

The `set_salary()` function abstracts away the need to check whether the user exists; if the user does not exist, it is created. Setting someone's salary to 0 will remove him or her from the salary table. Thus, the interface is simplified to a large extent, and the client application of these functions needs to know, and do, less:

```
CREATE OR REPLACE FUNCTION set_salary(i_emp_name text, i_salary int)
RETURNS TEXT AS $$
DECLARE
    old_salary integer;
BEGIN
    SELECT salary INTO old_salary
    FROM salaries
    UPDATE salaries
    SET salary = i_salary
    WHERE emp_name = i_emp_name;
RETURN $old_salary$;
$$ LANGUAGE SQL SECURITY DEFINER;
```

What Is a PostgreSQL Server?

```sql
WHERE upper(emp_name) = upper(i_emp_name);
IF NOT FOUND THEN
  INSERT INTO salaries VALUES(i_emp_name, i_salary);
INSERT INTO salary_change_log(salary_op,emp_name,new_salary)
  VALUES ('INSERT',i_emp_name,i_salary);
  RETURN 'INSERTED USER ' || i_emp_name;
ELSIF i_salary > 0 THEN
  UPDATE salaries
  SET salary = i_salary
WHERE upper(emp_name) = upper(i_emp_name);
INSERT INTO salary_change_log
  (salary_op,emp_name,old_salary,new_salary)
  VALUES ('UPDATE',i_emp_name,old_salary,i_salary);
  RETURN 'UPDATED USER ' || i_emp_name;
ELSE -- salary set to 0
  DELETE FROM salaries WHERE upper(emp_name) = upper(i_emp_name);
  INSERT INTO salary_change_log(salary_op,emp_name,old_salary)
    VALUES ('DELETE',i_emp_name,old_salary);
  RETURN 'DELETED USER ' || i_emp_name;
END IF;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

Now, drop the audit trigger (otherwise the changes will be logged twice) and test the new functionality:

```sql
postgres=# DROP TRIGGER audit_salary_change ON salaries;
DROP TRIGGER
postgres=#
postgres=# SELECT set_salary('Fred',750);
- [ RECORD 1 ]------------------
  set_salary | INSERTED USER Fred

postgres=# SELECT set_salary('frank',100);
- [ RECORD 1 ]-------------------
  set_salary | INSERTED USER frank

postgres=# SELECT * FROM salaries ;
- [ RECORD 1 ]---
  emp_name | Bob
  salary   | 1300
- [ RECORD 2 ]---
  emp_name | Mary
  salary   | 1200
```
- [ RECORD 3 ]---
  emp_name | Fred
  salary   | 750
- [ RECORD 4 ]---
  emp_name | frank
  salary   | 100

postgres=# SELECT set_salary('mary',0);
- [ RECORD 1 ]-----------------
  set_salary | DELETED USER mary

postgres=# SELECT * FROM salaries ;
- [ RECORD 1 ]---
  emp_name | Bob
  salary   | 1300
- [ RECORD 2 ]---
  emp_name | Fred
  salary   | 750
- [ RECORD 3 ]---
  emp_name | frank
  salary   | 100

postgres=# SELECT * FROM salary_change_log ;
...  
- [ RECORD 6 ]------------------------
  changed_by | gsmith
  changed_at | 2013-01-25 15:57:49.057592
  salary_op  | INSERT
  emp_name   | Fred
  old_salary | 
  new_salary | 750

- [ RECORD 7 ]------------------------
  changed_by | gsmith
  changed_at | 2013-01-25 15:57:49.062456
  salary_op  | INSERT
  emp_name   | frank
  old_salary | 
  new_salary | 100

- [ RECORD 8 ]------------------------
  changed_by | gsmith
  changed_at | 2013-01-25 15:57:49.064337
  salary_op  | DELETE
  emp_name   | mary
  old_salary | 1200
  new_salary |
Data cleaning

In the preceding code, we notice that employee names don't have consistent cases. It will be easy to enforce consistency by adding a constraint, as shown here:

```sql
CHECK (emp_name = upper(emp_name))
```

However, it is even better to just make sure that the name is stored as uppercase, and the simplest way to do this is by using trigger:

```sql
CREATE OR REPLACE FUNCTION uppercase_name ()
    RETURNS trigger AS $$
BEGIN
    NEW.emp_name = upper(NEW.emp_name);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER uppercase_emp_name
    BEFORE INSERT OR UPDATE OR DELETE ON salaries
    FOR EACH ROW EXECUTE PROCEDURE uppercase_name ();
```

The next `set_salary()` call for a new employee will now insert `emp_name` in uppercase:

```
postgres=# SELECT set_salary('arnold',80);
-[ RECORD 1 ]-------------------
  set_salary | INSERTED USER arnold
```

As the upcasing happens inside a trigger, the function's response still shows a lowercase name, but in the database, it is upcased:

```
postgres=# SELECT * FROM salaries;
-[ RECORD 1 ]---
  emp_name | Bob
  salary   | 1300
-[ RECORD 2 ]---
  emp_name | Fred
  salary   | 750
-[ RECORD 3 ]---
  emp_name | Frank
  salary   | 100
-[ RECORD 4 ]---
  emp_name | ARNOLD
  salary   | 80
```
After fixing the existing mixed-case employee names, we can make sure that all employee names will be uppercased in the future by adding a constraint:

    postgres=# update salaries set emp_name = upper(emp_name)
    where not emp_name = upper(emp_name);

    UPDATE 3

    postgres=# alter table salaries add constraint
    emp_name_must_be_uppercasepostgres CHECK (emp_name = upper(emp_name));

    ALTER TABLE

If this behavior is needed in more places, it will make sense to define a new type – say u_text, which is always stored as uppercase. You will learn more about this approach in Chapter 14, *PostgreSQL as Extensible RDBMS*.

### Custom sort orders

The last example in this chapter, is about using functions for different ways of sorting.

Say we are given a task to sort words by their vowels only, and in addition to this, to make the last vowel the most significant one when sorting. While this task may seem really complicated at first, it can be easily solved with functions:

```sql
CREATE OR REPLACE FUNCTION reversed_vowels(word text)
RETURNS text AS $$
    vowels = [c for c in word.lower() if c in 'aeiou']
    vowels.reverse()
    return ''.join(vowels)
$$ LANGUAGE plpythonu IMMUTABLE;
```

    postgres=# select word,reversed_vowels(word) from words order by reversed_vowels(word);

<table>
<thead>
<tr>
<th>word</th>
<th>reversed_vowels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abracadabra</td>
<td>aaaaa</td>
</tr>
<tr>
<td>Great</td>
<td>ae</td>
</tr>
<tr>
<td>Barter</td>
<td>ea</td>
</tr>
<tr>
<td>Revolver</td>
<td>eoe</td>
</tr>
</tbody>
</table>

(4 rows)

Before performing this code, please make sure you have Python 2.x installed. We will discuss PL/Python in much detail in the later chapters of this book.

[27]
The best part is that you can use your new function in an index definition:

```
postgres=# CREATE INDEX reversed_vowels_index ON words (reversed_vowels(word));
CREATE INDEX
```

The system will automatically use this index whenever the `reversed_vowels(word)` function is used in the `WHERE` or `ORDER BY` clause.

## Programming best practices
Developing application software is complicated. Some of the approaches that help manage this complexity are so popular that they have been given simple acronyms that can be remembered. Next, we'll introduce some of these principles and show you how server programming helps make them easier to follow.

### KISS – keep it simple stupid
One of the main techniques to successful programming is writing simple code. That is, writing code that you can easily understand 3 years from now and that others can understand as well. It is not always achievable, but it almost always makes sense to write your code in the simplest way possible. You can rewrite parts of it later for various reasons such as speed, code compactness, to show off how clever you are, and so on. However, always write the code in a simple way first, so that you can be absolutely sure that it does what you want. Not only do you get working on the code quickly, but you also have something to compare to when you try more advanced ways to do the same thing.

Remember, debugging is harder than writing code; so, if you write the code in the most complex way you can, you will have a really hard time debugging it.

It is often easier to write a set returning function instead of a complex query. Yes, it will probably run slower than the same thing implemented as a single complex query, due to the fact that the optimizer can do very little to the code written as functions, but the speed may be sufficient for your needs. If more speed is required, it's very likely to refactor the code piece by piece, joining parts of the function into larger queries where the optimizer has a better chance of discovering better query plans until the performance is acceptable again.

Remember that most of the time, you don't need the absolutely fastest code. For your clients or bosses, the best code is the one that does the job well and arrives on time.
DRY – don't repeat yourself

This principle means you should implement any piece of business logic just once and put the code for doing it in the right place.

This may be hard sometimes; for example, you want to do some checks on your web forms in the browser, but still do the final checks in the database. However, as a general guideline, it is very much valid.

Server programming helps a lot here. If your data manipulation code is in the database near the data, all the data users have easy access to it, and you will not need to manage a similar code in a C++ Windows program, two PHP websites, and a bunch of Python scripts doing nightly management tasks. If any of them need to do this thing to a customer's table, they just call:

```
SELECT * FROM do_thisThingToCustomers(arg1, arg2, arg3);
```

That's it!

If the logic behind the function needs to be changed, you just change the function with no downtime and no complicated orchestration of pushing database query updates to several clients. Once the function is changed in the database, it is changed for all the users.

YAGNI – you ain't gonna need it

In other words, don't do more than you absolutely need to.

If you have a creepy feeling that your client is not yet well aware of how the final database will look or what it will do, it's helpful to resist the urge to design everything into the database. A much better way is to do a minimal implementation that satisfies the current specifications, but do it with extensibility in mind. It is very easy to "paint yourself into a corner" when implementing a big specification with large imaginary parts.

If you organize your access to the database through functions, it is often possible to do even large rewrites of business logic without touching the frontend application code. Your application still performs `SELECT * FROM do_this_thing_to_customers(arg1, arg2, arg3)`, even after you have rewritten the function five times and changed the whole table structure twice.
What Is a PostgreSQL Server?

SOA – service-oriented architecture

Usually, when you hear the acronym SOA, it will be from enterprise software people trying to sell you a complex set of SOAP services. But the essence of SOA is to organize your software platform as a set of services that clients, and other services, call in order to perform certain well-defined atomic tasks, as follows:

- Checking a user’s password and credentials
- Presenting him/her with a list of his/her favorite websites
- Selling him/her a new red dog collar with a complementary membership in the red-collar dog club

These services can be implemented as SOAP calls with corresponding WSDL definitions and Java servers with servlet containers, as well as a complex management infrastructure. They can also be a set of PostgreSQL functions, taking a set of arguments and returning a set of values. If the arguments or return values are complex, they can be passed as XML or JSON, but a simple set of standard PostgreSQL data types is often enough. In Chapter 10, Scaling Your Database with PL/Proxy, you will learn how to make such a PostgreSQL-based SOA service infinitely scalable.

Type extensibility

Some of the preceding techniques are available in other databases, but PostgreSQL’s extensibility does not stop here. In PostgreSQL, you can just write UDFs in any of the most popular scripting languages. You can also define your own types, not just domains, which are standard types with some extra constraints attached, and new full-fledged types too.

For example, a Dutch company, MGRID, has developed a value with unit set of data types, so that you can divide 10 km by 0.2 hours and get the result in 50 km/h. Of course, you can also cast the same result to meters per second or any other unit of speed. And yes, you can get this as a fraction of c—the speed of light.

This kind of functionality needs both the types and overloaded operands, which know that if you divide distance by time, then the result is speed. You will also need user-defined casts, which are automatically or manually-invoked conversion functions between types.
MGRID developed this for use in medical applications, where the cost of an error can be high—the difference between 10 ml and 10 cc can be vital. However, using a similar system might also have averted many other disasters, where wrong units ended up producing bad computation results. If the amount is always accompanied by the unit, the possibility for these kinds of errors is diminished. You can also add your own index method if you have some programming skills and your problem domain is not well served by the existing indexes. There is already a respectable set of index types included in the core PostgreSQL, as well as several others that are developed outside the core.

The latest index method that became officially included in PostgreSQL is k nearest neighbor (KNN)—a clever index, which can return K rows ordered by their distance from the desired search target. One use of KNN is in fuzzy text search, where this can be used to rank full-text search results by how well they match the search terms. Before KNN, this kind of thing was done by querying all the rows which matched even slightly, then sorting all these by the distance function, and returning K top rows as the final step.

If done using the KNN index, the index access can start returning the rows in the desired order; so, a simple LIMIT K function will return the K top matches.

The KNN index can also be used for real distances, for example, answering the request "Give me the 10 nearest pizza places to Central Station."

As you saw, index types are different from the data types they index. Another example, is the same General Inverted Index (GIN) can be used for full-text searches (together with stemmers, thesauri, and other text-processing stuff), as well as for indexing elements of integer arrays.

**Caching**

Yet another place where server-side programming can be used is to cache values, which are expensive to compute. The following is the basic pattern here:

1. Check whether the value is cached.
2. If it isn't, or the value is too old, compute and cache it.
3. Return the cached value.
For example, calculating the sales for a company is the perfect item to cache. Perhaps, a large retail company has 1,000 stores with potentially millions of individual sales' transactions per day. If the corporate headquarters is looking for sales' trends, it is much more efficient if the daily sales numbers are precalculated at the store level instead of summing up millions of daily transactions.

If the value is simple, such as looking up a user's information from a single table based on the user ID, you don't need to do anything. The value gets cached in PostgreSQL's internal page cache, and all lookups to it are so fast that even on a very fast network, most of the time is spent doing the lookups in the network and not in the actual lookup. In such a case, getting data from a PostgreSQL database is as fast as getting it from any other in-memory cache (such as memcached) but without any extra overhead in managing the cache.

Another use case of caching is to implement materialized views. These are views that are precomputed only when required, not every time one selects data from the view. Some SQL databases have materialized views as separate database objects, but in the PostgreSQL versions prior to 9.3, you have to do it yourself using other database features to automate the whole process.

**Wrapping up – why program in the server?**

The main advantages of doing most data manipulation code on the server-side are stated in the following sections.

**Performance**

Doing the computation near the data is almost always a performance win, as the latencies to get the data are minimal. In a typical data-intensive computation, most of the time is spent in getting the data. Therefore, making data access inside the computation faster is the best way to make the whole thing fast. On my laptop, it takes 2.2 ms to query one random row from a 1,000,000-row database into the client, but it takes only 0.12 ms to get the data inside the database. This is 20 times faster and inside the same machine over Unix sockets. The difference can be bigger if there is a network connection between the client and the server.
A small real-word story:

A friend of mine was called to help a large company (I'm sure all of you know it, but I can't tell you which one) in order to make its e-mail sending application faster. They had implemented their e-mail generation system with all the latest Java EE technologies: first, getting the data from the database, passing the data around between services, and serializing and deserializing it several times before finally doing XSLT transformation on the data to produce the e-mail text. The end result being that it produced only a few hundred e-mails per second, and they were falling behind with their responses.

When he rewrote the process to use a PL/Perl function inside the database to format the data and the query returned already fully-formatted e-mails, it suddenly started spewing out tens of thousands of e-mails per second and they had to add a second copy of the sent mail to actually be able to send them out.

**Ease of maintenance**

If all the data manipulation code is in a database, either as database functions or views, the actual upgrade process becomes very easy. All that is needed is to run a DDL script that redefines the functions; all the clients automatically use the new code with no downtime and no complicated coordination between several frontend systems and teams.

**Improved productivity**

Server-side functions are perhaps the best way to achieve code reuse. Any client application written in any language or framework can make use of the server-side functions, ensuring maximum reuse in all environments.

**Simple ways to tighten security**

If all the access for some possibly insecure servers goes through functions, the database user of these servers can only be granted access to the needed functions and nothing else. They can't see the table data or even the fact that these tables exist. So, even if the server is compromised, all it can do is continue to call the same functions. Also, there is no possibility of stealing passwords, e-mails, or other sensitive information by issuing its own queries such as `SELECT * FROM users;` and getting all the data there is in the database.

Also, the most important thing is that programming in a server is fun!
What Is a PostgreSQL Server?

Summary

Programming inside the database server is not always the first thing that comes to mind to many developers, but its unique placement inside the application stack gives it some powerful advantages. Your application can be faster, more secure, and more maintainable by pushing logic into the database. With server-side programming in PostgreSQL, you can secure your data using functions, audit access to your data and structural changes using triggers, and improve productivity by achieving code reuse. Also, you can enrich your data using custom data types, analyze your data using custom operators, and extend the capabilities of the database by dynamically loading new functions.

This is just the start of what you can do inside PostgreSQL. Throughout the rest of this book, you will learn many other ways to write powerful applications by programming inside PostgreSQL.
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.