PostgreSQL Developer's Guide

PostgreSQL is an enterprise-level database that competes among proprietary database vendors, owing to its remarkable feature set and reliability without the expensive licensing overhead.

This book is a comprehensive and pragmatic guide to developing databases in PostgreSQL. Beginning with a refresher of basic SQL skills, you will gradually be exposed to advanced concepts, such as learning how to program in native PostgreSQL procedural languages, implementing triggers, custom functions, and views. You will learn database optimization techniques such as query optimization and indexing while exploring extensive code examples. Lastly, you will explore foreign data wrappers, implementing extensibility, and improving maintainability.

Who this book is written for

If you are a database developer who wants to learn how to design and implement databases for application development using PostgreSQL, this is the book for you.

Existing knowledge of basic database concepts and some programming experience is required.

What you will learn from this book

- Refresh your knowledge of SQL with PostgreSQL-supported DDL and DML
- Explore native PostgreSQL procedural languages and learn how to write custom functions
- Leverage the power of indexes for optimal database efficiency
- Unfold tricky trigger concepts and implement them in native procedural languages
- Discover how to implement views, partitioned tables, and window functions in PostgreSQL
- Design efficient queries with query optimization and query analysis
- Communicate with PostgreSQL servers using foreign data wrappers such as LibPQ and ECPG
- Extend database powers using foreign data wrappers and achieve maintainability via PostgreSQL extensions
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 8 ‘Dealing with Large Objects’
- A synopsis of the book’s content
- More information on PostgreSQL Developer’s Guide

About the Authors

Ibrar Ahmed has over 15 years of experience in software development. He is currently working as a software architect for a PostgreSQL-based company. He is an open source evangelist. He has contributed features such as pg_migrator (now called pg_upgrade) and Index Only Scans to PostgreSQL and has contributed to Google Chromium projects such as platform-independent FTP implementations of the Google Chrome browser, Chromium test framework enhancements, and porting of listen_socket and telnet_server to Linux. He has a lot of experience in implementing network protocols and writing Linux kernel modules. Apart from his professional life, he delivers lectures at universities and reads books.

I would like to thank my loving parents for everything they did for me. Personal time always belongs to family, and I did this in my personal time, so thanks to my family for all the support. I would also like to thank Mr. Richard Harvey, who encouraged me to write the book, and my early age mentor, Mr. Mahmood Hussain, who helped me a lot at the start of my professional career. It has been a really great experience to work with Amjad Shahzad and Asif Fayyaz.
Asif Fayyaz is an open source technology lover and practitioner. He is currently working as a senior software quality assurance engineer at a leading PostgreSQL-based company. He has been learning and using PostgreSQL for the last 7 years. His areas of interest are exploring PostgreSQL replication solutions and software test automation solutions for desktop, web, and mobile applications. His future endeavors include joining a sports channel for set-top box automation in Python.

Apart from his professional activities, he, along with his dedicated friends, is keen to find ways that can make life easier for those who are facing the worst living conditions.

His other passions include, but are not limited to, traveling to different places, trying different cuisines, and reading books if somehow permitted by his loving family.

I would like to thank my loving parents, encouraging siblings, friends, colleagues, my caring wife, and adorable daughter for not only bearing with my busy schedule but motivating me as well to get all this done.

I would also like to thank my colleagues, Ibrar Ahmed, for his technical expertise and guidance, and Amjad Shahzad, for his constant moral and technical support.

A thumbs up to Packt Publishing and notably to Richard, Gregory, Azhar, and Samantha. You folks were great mentors and got the best out of us. Undoubtedly a great experience!

Amjad Shahzad has been working in the open source software industry for the last 10 years. He is currently working as a senior quality assurance engineer at a leading PostgreSQL-based company, which is the only worldwide provider of enterprise-class products and services based on PostgreSQL. Amjad's core expertise lies in the areas of pg_upgrade, slony and streaming replication, Cloud database, and database partitioning. His future endeavors include exploring PostgreSQL replication solutions.

Apart from his professional activities, he is also involved in doing social activities that involve helping people stand on their feet. In his free time, he likes to explore nature by doing outdoor activities, including hiking, trekking, and nature photography.

I would like to say a special thank you to my parents, who have been my source of inspiration since the start of my career. I would also like to thank my wife for putting up with my late night writing sessions. I would also like to thank Ibrar Ahmed, without whom this book would not have been possible.

Finally, I want to thank the Packt Publishing staff, which includes Richard, Gregory, Azhar, and Samantha for their continuous support.
PostgreSQL Developer's Guide

PostgreSQL is the world's most advanced community-driven open source database. The first open source version of PostgreSQL was released on 1st August 1996, an combined effort between Bruce Momjian and Vadim B. Mikheev. Since then, major releases have come annually, and all releases are available under its free and open source software PostgreSQL license similar to the BSD and MIT licenses. Modern technologies are emerging with new features on a regular basis, and PostgreSQL is one of the fantastic examples of this happening, adding more robust features to cope with the changing trends of technology. Developer and database administrators love to use PostgreSQL because of its reliability, scalability, and continuous support from the open source community.

PostgreSQL Developer's Guide is for database developers fascinated with learning and understanding PostgreSQL from its release. A basic awareness of database concepts is required to understand all of the PostgreSQL technical terms. As a result, by reading this guide, you, as a reader, will be able to understand how applications can be programmed with PostgreSQL, along with the core development concepts. By the end of this book, you will have a solid base in the fundamental development concepts and be able to develop database applications by leveraging the core programming functionality of PostgreSQL.

The main objective of this book is to teach you in programming database applications and custom programmatic functions. It is a practical tutorial book with an emphasis to provide authentic world examples of how applications can be programmed with PostgreSQL and grips on core development concepts and functions. By the end of this book, we will show you how to write custom programming functions, which extends the PostgreSQL database beyond its core capabilities. We wish you the best of luck on your quest of seeking knowledge of PostgreSQL, where we hope that at the end of this book, you will feel like you deserve a pat on the back for your efforts in acquiring some hands-on expertise with PostgreSQL.
**What This Book Covers**

*Chapter 1, Getting Started with PostgreSQL*, explains the birth, present, and future along with the evolution of PostgreSQL in terms of its features, maintainability, and its immensely huge global following. This chapter will explain the concepts of DDL (Data Description Language) and DML (Data Manipulation Language), and explain how to write DDL and DML statements.

*Chapter 2, The Procedural Language*, encompasses the diversified features of PL/pgSQL, native support for four languages, and extensibility for others. We will skim through the description of PL/pgSQL by explaining its structure, declarations, and verbal expressions. This chapter will shed light on using native support and utilization examples of other procedural languages such as PL/Python, PL/Tcl, and PL/Perl.

*Chapter 3, Working with Indexes*, is all about indexes, so expect to see a discussion of the fundamental concepts of indexes, such as the kinds of indexes PostgreSQL supports and the syntax to create them. The main story of this chapter is where to utilize what kind of index and which condition it is best suited for. You can then build different kinds of indexes in the warehouse database to explicate the practical use of indexes.

*Chapter 4, Triggers, Rules, and Views*, consists of three sections: triggers, rules, and views. The first section of this chapter will explain what a trigger is and how to create triggers in PostgreSQL. The second part will deal with PostgreSQL rules. There will be a focus on how the rules work by explaining their call, input, and the results. The final third part will revolve around views and why they are important in database design.

*Chapter 5, Window Functions*, discusses the power and concepts of window functions in conjunction with aggregate functions. We will also cover the scope, structure, and usage of window functions with examples. Another objective will be to acquire a crystal clear understanding of the core of window functions and the data that is processed with the help of frame, OVER, PARTITION BY, and ORDER BY clauses. This chapter will also discuss the available built-in window functions, along with custom ones.

*Chapter 6, Partitioning*, deals with table partitioning. Table partitioning in PostgreSQL is implemented through table inheritance. In this section, there will be a brief overview of partition in order to improve the performance of queries before moving on to its implementation in PostgreSQL. The chapter also covers the list (that utilizes key values) and range (that utilizes key columns) partitions. *Chapter 7, Query Optimization*, is about query analysis and optimization. Queries can be optimized utilizing indexes and hints and manipulating planner parameters. As a reader, you will find this chapter very useful in utilizing and optimizing your queries.
Chapter 8, *Dealing with Large Objects*, is about the handling of Large Objects (LO) as there is a need to store large objects such as audio and video files. PostgreSQL has support to manipulate these objects. The handling of sizably huge objects is consummately different from the other objects such as text, varchar, and int. This chapter will explain why we need to store Large Objects (LO) and how PostgreSQL implements LO storage.

Chapter 9, *Communicating with PostgreSQL Using LibPQ*, explains how to write C programs and connect and execute queries in the C language using libpq, which is a PostgreSQL client library to communicate with the PostgreSQL server. In this chapter, we will grip on the different ways of communication by utilizing libpq and the utilization of all libpq functions. To extend our story of the warehouse database, we will write a program to manipulate the data from the database.

Chapter 10, *Embedded SQL in C – ECPG*, covers all the syntax and utilization of Embedded SQL to manipulate the data inside this code. Other than libpq, there is an alternative to communicate in C code to a PostgreSQL server called ECPG. Additionally, there will be coverage of how to compile the ECPG program, and we will discuss the command-line options of the ECPG binary.

Chapter 11, *Foreign Data Wrapper*, covers how to explain the building blocks of the foreign data wrapper and discusses how to utilize postgres_fdw and file_fdw to manipulate foreign data. PostgreSQL introduces an incipient feature called the foreign data wrapper. It's a template to write the module to access foreign data. This is rigorously based in SQL/MED standards (SQL Management of External Data). There are only two community maintained wrappers, postgres_fdw and file_fdw, along with many externally maintained foreign data wrappers.

Chapter 12, *Extensions*, covers how to install and use available extensions in PostgreSQL. PostgreSQL has features to install the loadable modules called extensions. Instead of creating a bunch of objects by running SQL queries, an extension, which is a collection of objects, can be created and dropped using a single command. The main advantage of an extension is maintainability. There are several extensions available.
Dealing with Large Objects

Databases provide data types as suitable containers to store values accordingly. You use the `int` data type to store numeric values and `char` and `varchar` data types for string values. Each data type has its own limitations with respect to the size and type of data it can store. A database solution model will be based on real life problems, therefore these are not the only types of data that you will always confront. We do not live in the ice age anymore; we have to store large-sized images, audio, video files and `varchar` is certainly not the answer to this. Objects that require huge storage size and can't be entertained with simple available data types are usually referred to as **Large Objects (LOs)** or **Binary Large Objects (BLOBs)**. To handle these LOs, you need a LO storage mechanism that can store them easily and access them efficiently.

This chapter is all about large objects, and it is well described under this heading to give you a clear concept of LOs or BLOBs. In this chapter, we will cover the following:

- Why large objects?
- PostgreSQL large objects
- Large objects in action
- Manipulating large objects through the `libpq` client interface library

**Why large objects?**

We will see in a more comparative mode why and where we need large objects.

You can categorize large objects in general, based on the structure or type of data. The types you usually come across are as follows:

- Simple
- Complex
Dealing with Large Objects

- Semi-structured
- Unstructured

Among the first two, \textit{simple-structured} data is related to data that can be organized in simple tables and data types and \textit{complex-structured} data is the one that deals with requirements such as that of user-defined data types.

In the age of the Internet, the types mentioned previously are not the only forms of data that you have to handle; you have XML and JSON as well. It's not interpreted by a relational database in a general way. This type of data can be categorized as \textit{semi-structured}. Again, referring to storage of images, audio, and videos that are used massively today and can't be stored in the same way as the first three types of data because they can't be broken down into smaller logical structures for interpretation by standard means. It is hence \textit{unstructured} data and needs a different mechanism to handle them.

PostgreSQL answers your problems with the feature of large objects that store objects of considerably huge size, and it's been there since the release of the PostgreSQL. Good things happened and over the time it's got even better.

\section*{PostgreSQL large objects}

Interestingly, PostgreSQL provides two ways to store large objects with respect to each requirement you have to meet. They are as follows:

- Implementation of the \texttt{BYTEA} data type
- Implementation of large object storage

Though our area of interest here has been large objects, yet we will skim through some characteristics of \texttt{BYTEA}. It is similar to \texttt{VARCHAR} and \texttt{TEXT} character strings, yet it has a few distinctive features as well. It can store raw or unstructured data, but character strings do not. It also allows storing of null values. \texttt{VARCHAR} does not permit storing zero octets, other octets, or sequences of octet values that are not valid as per database character set encoding. While using \texttt{BYTEA}, you can manipulate actual raw bytes, but in the case of character strings, processing is dependent on locale setting.

\texttt{BYTEA} when compared with large object storage comes with a big difference of storage size; each \texttt{BYTEA} entry permits storage of 1 GB whereas large objects allow up to 4 TB. The large object feature provides functions that help you manipulate external data in a much easier way that could be quite complex when doing the same for \texttt{BYTEA}.
The preceding discussion was a small comparison and analysis to show you the available choices in PostgreSQL to store binary data using BYTEA or large object storage. A requirement is the best judge to opt any of these.

Implementing large objects

Things are well remembered when they are listed and this is how we will remember PostgreSQL large objects implementation in our memory:

- Large objects, unlike BYTEA, are not a data type but an entry in a system table.
- All large objects are stored in the pg_largeobject system table.
- Each large object also has a corresponding entry in the pg_largeobject_metadata system table.
- Large objects are broken up into chunks of default size and further stored as rows in the database.
- These chunks in rows are B-tree indexed; hence, this ensures fast searches during read/write operations.
- From PostgreSQL 9.3 onwards, the maximum size of a large object in a table can be 4 TB.
- Large objects are not stored in user tables; rather, a value of the Object Identifier (OID) type is stored. You will use this OID value to access the large object. So, when you have to access a large object, you will reference the OID value that points to a large object present on the pg_largeobject system table.
- PostgreSQL provides the read/write Application Program Interface (API) that offers client- and server-side functions. Using this API, you can perform operations such as create, modify, and delete on large objects. OIDs are used in this function as a reference to access large objects, for example, to transfer the contents of any file to the database or to extract an object from the database into a file.
- From PostgreSQL 9.0 onwards, large objects now have an associated owner and a set of access permissions. Retrieving data using these functions gives you the same binary data you added. Examples of the functions are lo_create(), lo_unlink(), lo_import(), and lo_export().
- PostgreSQL provides the ALTER LARGE TABLE feature to change the definition of a large object. Remember that its only functionality is to assign a new owner.
Functions for large objects must be called in a transaction block, so when autocommit is off, make sure that you issue the BEGIN command explicitly.

Large objects in action

Now is the time to get some hands-on practice with large objects.

In this section, you will play with large objects using server functions in your SQL statements. In the following section, you will be exposed to access large objects through the libpq client interface library.

Let's search PostgreSQL for the list of functions available to access large objects:

```sql
postgres=# SELECT n.nspname as "Schema", p.proname as "Name",
    pg_catalog.pg_get_function_result(p.oid) as "Result data type",
    pg_catalog.pg_get_function_arguments(p.oid) as "Argument data types"
FROM pg_catalog.pg_proc p LEFT JOIN
    pg_catalog.pg_namespace n ON n.oid = p.pronamespace
WHERE p.proname ~ '^(lo_.*)$' AND
    pg_catalog.pg_function_is_visible(p.oid)
ORDER BY 1, 2, 4;
```

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Result data type</th>
<th>Argument data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>lo_close</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_create</td>
<td>oid</td>
<td>integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_create</td>
<td>oid</td>
<td>oid</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_export</td>
<td>integer</td>
<td>oid, text</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_import</td>
<td>oid</td>
<td>text</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_import</td>
<td>oid</td>
<td>text, oid</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_lseek</td>
<td>integer</td>
<td>integer, integer,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_lseek64</td>
<td>bigint</td>
<td>integer, bigint,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_open</td>
<td>integer</td>
<td>oid, integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_tell</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_tell64</td>
<td>bigint</td>
<td>integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_truncate</td>
<td>integer</td>
<td>integer, integer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_truncate64</td>
<td>integer</td>
<td>integer, bigint</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>lo_unlink</td>
<td>integer</td>
<td>oid</td>
</tr>
</tbody>
</table>

(14 rows)
Suppose we need a table that can store digital pictures of huge sizes.

We will first create a table named test_large_objects. This can be done in the following manner:

```sql
postgres=# CREATE TABLE test_large_objects
(   picture_id INTEGER,
    name VARCHAR(30),
    picture_loc oid,
    CONSTRAINT pk_picture_id PRIMARY KEY(picture_id)
);
```

We have an image residing in our local storage on a directory, say the /tmp/ directory, named as pg_disney.jpg.

```sql
[ User should have permissions to read the file. ]
```

Of course, we have not inserted a row with an image yet. Let's still query the pg_largeobject system table and the test_large_objects table for their current state. When we insert a row that includes a function call, it will affect the system table as well. Let's check the pg_largeobject system table using the following statement:

```sql
postgres=# SELECT * FROM pg_largeobject;
 loid | pageno | data
-------+--------+------
 (0 rows)
```

Let's check the test_large_object table using the following statement:

```sql
postgres=# SELECT * FROM test_large_objects;
picture_id | name | picture_loc
-------------+------+------------------
 (0 rows)
```
Dealing with Large Objects

Now, insert a row with the full path to the image in the following manner:

```
postgres=# INSERT INTO test_large_objects VALUES
    (1, 'pg_disney_trip', lo_import('/tmp/pg_disney.jpg'));
ERROR:  could not open server file "/tmp/pg_disney.jpg": Permission denied
```

When using dummy data or images to test the large objects, make sure that they are of some size or contain data.

It was intentionally done to remind us to give appropriate permissions to the file we have to store. Giving the appropriate permission to the `postgres` user will let you add the file to the database. This can be done in the following manner:

```
postgres=# INSERT INTO test_large_objects VALUES
    (1, 'pg_disney_trip', lo_import('/tmp/pg_disney.jpg'));
INSERT 0 1
```

What happens here is that the `lo_import()` function has loaded the image inside the `pg_largeobject` table and returns an OID value as a reference to the large object. If you query your table, you will see the OID value and not the actual image.

Let's query the `test_large_objects` user table using the following statement:

```
postgres=# SELECT * FROM test_large_objects;
picture_id |      name      | picture_loc
------------+----------------+-------------
    1       | pg_disney_trip |       74441
(1 row)
```

Now, query the `pg_largeobjects` system table using the following statement:

```
postgres=# SELECT loid FROM pg_largeobject;
loid
-------
    74441
(1 row)
```

We'll now query the `pg_largeobjects_metadata` table using the following statement:

```
postgres=# SELECT oid FROM pg_largeobject_metadata;
oid
-------
    74441
(1 row)
```
You will observe different OIDs when practicing these examples.

It's quite evident now that you have successfully imported an image inside a database and it has been assigned an OID as well.

You learned how to import an object; exporting or retrieving a large object can be simpler as well but with the difference that you will use the `lo_export()` function along with the OID in the following manner:

```sql
postgres=# SELECT lo_export(74441, '/tmp/pg_disney_second.jpg');
```

```
<table>
<thead>
<tr>
<th>lo_export</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```

If you visit the `/tmp` directory, you will find an image with the name `pg_disney_second.jpg`.

This is how you retrieve large objects from the database, and in the end, we will try to delete this large object using the `lo_unlink()` function in the following manner:

```sql
postgres=# SELECT lo_unlink(74441);
```

```
<table>
<thead>
<tr>
<th>lo_unlink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```

You might observe the usage of OID as a reference to export and unlink the large objects. To make sure that it goes well, let's query the `pg_largeobject` and `pg_largeobject_metadata` tables again.

First, query the `pg_largeobject` table using the following statement:

```sql
postgres=# SELECT loid FROM pg_largeobject;
```

```
<table>
<thead>
<tr>
<th>loid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Then, query the `pg_largeobject_metadata` table using the following statement:

```sql
postgres=# SELECT oid FROM pg_largeobject_metadata;
```

```
<table>
<thead>
<tr>
<th>oid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

---

[163]
Dealing with Large Objects

However, the user table still contains the row and if you try to run the `lo_unlink()` function again, it will prompt you with a meaningful message that your large object does not exist.

Use `SELECT` on the `test_large_objects` table as follows:

```
postgres=# SELECT * from test_large_objects;
picture_id |      name      | picture_loc
-------------+----------------+-------------
            1 | pg_disney_trip |       74441
(1 row)
```

Use the `lo_unlink()` function as follows:

```
postgres=# SELECT lo_unlink(74441);
ERROR:  large object 74441 does not exist
```

### Manipulating large objects through the libpq client interface library

PostgreSQL provides multiple ways to store and access large objects. The PostgreSQL `libpq` client interface library helps you access the large objects with ease and efficiency.

This section will give you an overview of some of the main functions, their usage as per the online PostgreSQL manual, and their usage in code examples crafted for you.

For a detailed reference, you can refer to the online PostgreSQL documentation, available at [http://www.postgresql.org/docs/9.4/static/lo-interfaces.html](http://www.postgresql.org/docs/9.4/static/lo-interfaces.html).

When using these functions in client applications, include the `libpq/libpq-fs.h` header file and link with the `libpq` library.

We have designed two examples in the C programming language for you; the first example will demonstrate importing and exporting using the `libpq` library and the second will focus on accessing via the open, write, read, close, and unlink functionalities using the `libpq` client interface. Let’s go through each functionality one by one:
**lo_create**

The syntax for the `lo_create` function is as follows:

```c
Oid lo_create (PGconn *conn, int mode);
```

The `lo_create` function creates a new large object and returns an OID that is assigned to the newly created large object.

This is how we will use this in our code example (see details of the symbolic constants, that is, `INV_READ`, `INV_WRITE`, or `INV_READ | INV_WRITE` in the `libpq/libpq-fs.h` header file):

```c
lo_oid = lo_create (conn, INV_READ | INV_WRITE);
```

**lo_import**

We have already seen that `lo_import` is used to import a large object inside the database.

The syntax for the `lo_import` function is as follows:

```c
Oid lo_import (PGconn *conn, const char *filename);
```

This function takes a filename as an argument that contains the full path of the object and returns an OID. As the file is read by the client interface library and not the server, so it must exist in the client file system with appropriate permissions.

The following example shows how `lo_import` is used:

```c
lo_oid = lo_import (conn, FILE_NAME);
```

**lo_export**

The `lo_export` function is used to retrieve or export a large object from the system table into the operating system file.

The syntax for the `lo_export` function is as follows:

```c
int lo_export (PGconn *conn, Oid lobjId, const char *filename);
```

The following example shows how `lo_export` is used:

```c
lo_export (conn, lo_oid, FILE_NAME)
```
Dealing with Large Objects

You will now be shown the first part of the code examples we were referring to. In the imp_exp.c example, we first connected to the PostgreSQL database; after a successful connection, we imported the imp.jpg file into the database using lo_import, and then we exported it as exp.jpg from the database using the OID returned in lo_export. How to connect and play with the PostgreSQL database through libpq is explained in detail later in the chapter. So for now, you need not be worried of how things are working here. As said earlier, to grasp it firmly, revisit the code examples after reading the next chapter.

The first part of the code example is as follows:

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"
#define FILE_TO_EXPORT   "exp.jpg"
#define FILE_TO_IMPORT   "imp.jpg"

int main(int argc, char **argv)
{
    PGconn *conn;
    PGresult *res;
    int lo_oid;

    /* Connect to Database testdb */
    conn = PQsetdb(NULL, NULL, NULL, NULL, "testdb");
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "connection to database failed\n");
        fprintf(stderr, "%s", PQerrorMessage(conn));
        return -1;
    }
```

[166]
/* Execute the command BEGIN */
    res = PQexec(conn, "BEGIN");
    PQclear(res);

/* Import file to the database */
    lo_oid = lo_import(conn, FILE_TO_IMPORT);
    if (lo_oid < 0)
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        PQfinish(conn);
        return -1;
    }
    res = PQexec(conn, "END");
    PQclear(res);

    res = PQexec(conn, "BEGIN");
    PQclear(res);

    if (lo_export(conn, lo_oid, FILE_TO_EXPORT) < 0)
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        PQfinish(conn);
        return -1;
    }
    res = PQexec(conn, "END");
    PQclear(res);

    PQfinish(conn);
    fprintf(stdout, "file '%s' successful imported using 'lo_import'
and then exported to '%s' using 'imp_export'\n", FILE_TO_IMPORT, FILE_TO_EXPORT);
    return lo_oid;
}

**lo_open**

The `lo_open` function is used to open a large object for reading or writing.

The syntax for the `lo_open` function is as follows:

```c
int lo_open(PGconn *conn, Oid lobjId, int mode);
```
Dealing with Large Objects

The `lo_open` function returns a large object descriptor to be used in `lo_read`, `lo_write`, `lo_lseek`, `lo_lseek64`, `lo_tell`, `lo_tell64`, `lo_truncate`, `lo_truncate64`, and `lo_close`.

The following example shows how `lo_open` is used:

```c
lo_fd = lo_open(conn, lo_oid, INV_READ);
```

**lo_write**

The syntax for the `lo_write` function is as follows:

```c
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

This function writes `len` bytes from `buf` (which must be at least of the size `len`) to the large object descriptor `fd`. The number of bytes actually written is returned and always equal to `len` unless there is an error where the return value is `i1-1`.

The following example shows how `lo_write` is used:

```c
r = lo_write(conn, lo_fd, buf, n);
```

**lo_read**

The syntax for the `lo_read` function is as follows:

```c
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

Explanation is almost the same as `lo_write` with the difference of operation; here, it reads up to `len` bytes from the large object descriptor `fd` into `buf` of the size `len`. The `fd` argument should be returned by the previously mentioned `lo_open` function. The number of bytes actually read is returned, and it will be less than `len` if the end of the large object is reached first. In the case of an error, the return value is `-1`.

The following example shows how `lo_read` is used:

```c
n = lo_read(conn, lo_fd, buf, 512);
```

Though similar in behavior to `lo_lseek`, `lo_read` can accept an offset larger than 2 GB and can deliver a result larger than 2 GB.

**lo_close**

You can close a large object descriptor by calling `lo_close`.

The syntax for the `lo_close` function is as follows:

```c
int lo_close(PGconn *conn, int fd);
```
Here `fd` is a large object descriptor returned by `lo_open`. On success, `lo_close` returns 0.

The following example shows how `lo_close` is used:

```c
lo_close(conn, lo_fd);
```

In case of an error, the return value is -1.

**lo_unlink**

To remove a large object from the database, call the `lo_unlink` function.

The syntax for the `lo_unlink` function is as follows:

```c
int lo_unlink(PGconn *conn, Oid lobjId);
```

The `lobjId` argument specifies the OID of the large object to remove. It will return 1 if it is successful and -1 if failed.

Now we jump to the second example we mentioned previously. In this example, a different approach is used to access large objects. Instead of import and export functions, we used the `lo_write` and `lo_read` functions to achieve the same.

The PostgreSQL manual states the following:

"Any large object descriptors that remain open at the end of a transaction will be closed automatically."

The second example is as follows:

```c
/*-----------------------------------------------
 * read_write.c
 * Accessing large objects using lo_read and lo_write
 *-----------------------------------------------
 */
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>
#include <fcntl.h>
#include <unistd.h>
```
Dealing with Large Objects

```c
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"
#define FILE_TO_EXPORT   "exp.jpg"
#define FILE_TO_IMPORT   "imp.jpg"

int main(int argc, char **argv)
{
    PGconn *conn;
    PGresult *res;
    int lo_oid;
    int fd, lo_fd;
    int n;
    char buf[1024];
    int r;

    /* Connect to Database testdb */
    conn = PQsetdb(NULL, NULL, NULL, NULL, "testdb");
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "connection to database failed\n");
        fprintf(stderr, "%s", PQerrorMessage(conn));
        return -1;
    }

    /* Execute the command BEGIN */
    res = PQexec(conn, "BEGIN");
    PQclear(res);

    fd = open(FILE_TO_IMPORT, O_RDONLY, 0666);
    if (fd < 0)
    {
        fprintf(stderr, "open: failed to open file %s\n", FILE_TO_IMPORT);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        return 0;
    }

    lo_oid = lo_creat(conn, INV_READ | INV_WRITE);
    if (lo_oid < 0)
    {
        fprintf(stderr, "lo_create: failed to create object\n");
        fprintf(stderr, "%s", PQerrorMessage(conn));
        return 0;
    }
```

lo_fd = lo_open(conn, lo_oid, INV_WRITE);
for(;;)
{
    n = read(fd, buf, 1024);
    if (n <= 0)
        break;
    r = lo_write(conn, lo_fd, buf, n);
    if (r < n)
        fprintf(stderr, "write: failed to write object\n");
}
lo_close(conn, lo_fd);
close(fd);

res = PQexec(conn, "END");
PQclear(res);
res = PQexec(conn, "BEGIN");
PQclear(res);

fd = open(FILE_TO_EXPORT, O_CREATE | O_WRONLY, 0666);
if (fd < 0)
{
    fprintf(stderr, "open: failed to open file %s\n", FILE_TO_EXPORT);
    return 0;
}

lo_fd = lo_open(conn, lo_oid, INV_READ);
if (lo_fd < 0)
{
    fprintf(stderr, "lo_open: failed to create object\n");
    fprintf(stderr, "%s", PQerrorMessage(conn));
    return 0;
}

fd = open(FILE_TO_EXPORT, O_CREATE | O_WRONLY, 0666);
if (fd < 0)
{
    fprintf(stderr, "open: failed to open file %s\n", FILE_TO_EXPORT);
    return 0;
}
Dealing with Large Objects

for (;;)
{
    n = lo_read(conn, lo_fd, buf, 1024);
    if (n <= 0)
        break;

    r = write(fd, buf, n);
    if (r < n)
    {  
        fprintf(stderr, "write: failed to write object\n");
        break;
    }
}

lo_close(conn, lo_fd);
close(fd);

res = PQexec(conn, "END");
PQclear(res);
PQfinish(conn);

fprintf(stdout, "file '%s' successfully imported using 'lo_write' and then exported to '%s' using 'lo_read'\n", FILE_TO_IMPORT, FILE_TO_EXPORT);
return lo_oid;

Usage of both approaches is clear from the fact that you can use import/export when you have to access the complete file/binary data but read/write allows processing on selective parts of the binary data stored as a large object.

For readers, here are the steps to build the code and content of the make file for a quick reference.

Use the make command to build the code in the following manner:

    make imp_exp
    make read_write
    make clean

[172]
The following is the content of Makefile:

```bash
CC=gcc
POSTGRESQL_INCLUDE=/usr/local/pgsql.master/include
POSTGRESQL_LIB=/usr/local/pgsql.master/lib
CFLAGS=-I$(POSTGRESQL_INCLUDE)
LD_FLAGS=-L$(POSTGRESQL_LIB)

imp_exp:
  $(CC) imp_exp.c -o imp_exp $(CFLAGS) $(LD_FLAGS) -lpq
read_write:
  $(CC) read_write.c -o read_write $(CFLAGS) $(LD_FLAGS) -lpq
clean:
  rm -rf *.o imp_exp read_write
```

**Summary**

This chapter has tried to build your clear concepts about PostgreSQL large objects, their usage, and implementation. You not only learned why and when to use large objects, but got hands-on practice on how to use them as well. Accessing them through **libpq** (C-based) client interface library has widened your scope for large object access and strengths of **libpq**. In fact, a whole chapter is designed to explain **libpq** in much detail, and the good news is, this is the next chapter!
Where to buy this book

You can buy PostgreSQL Developer's Guide from the Packt Publishing website.

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

Click here for ordering and shipping details.