12 RDBMS and ORM

RDBMSs [1] and ORM [2] are touchy subjects, but there's no way to avoid having to deal with them sooner or later. Many applications have to store data of some kind, and developers often choose to do so using relational databases. And when a developer chooses to use a relational database, the tool they almost always choose to use for it is an ORM library of some kind.

Note

This chapter will be a little less Python-centric than others; bear with me. I'll only be talking about relational databases here, but many of the things we'll cover here can also apply to other kinds of databases.

RDBMSs are about storing relational data using normal form, while SQL is about dealing with relational algebra. Together, they allow you to store data and answer questions about that data. However, there are a number of common difficulties with using ORM in object-oriented programs, known collectively as the object-relational impedance mismatch. The bottom line is, relational databases and object-oriented programs have different representations of data which don't map properly to one another: mapping SQL tables to Python classes won't give you optimal results, no matter what you do.

[1] Relational database management systems
[2] Object-relational mapping

ORM is supposed to make database systems easier to access: these tools abstract the process of creating queries, generating SQL so you don't have to. Unfortunately, more likely sooner than later, you'll want to do something with your database only to discover that the abstraction layer simply won't allow it. To make the most efficient use of your database, you absolutely have to have an understanding of SQL and RDBMSs so that you can write your own queries directly without having to rely on the abstraction layer for everything.

But that's not to say you should avoid ORM entirely. ORM libraries can help with rapid prototyping of your application model, and some even provide useful tools such as schema upgrades/downgrades. The important thing is that you understand that it's not a substitute for a proper grasp of RDBMSs: many developers try to solve problems in the language of their choice rather than using their model API, and the solutions they come up with are inelegant at best.

Imagine a SQL table for keeping track of messages. It has a single column named "id," which is the primary key, and a string containing the message:

```
CREATE TABLE message (
  id serial PRIMARY KEY,
  content text
);
```

We want to avoid duplicates when receiving a message, so a typical developer would write something like this:

```
if message_table.select_by_id(message.id):
    # We already have the message, it's a duplicate, ignore and raise
    raise DuplicateMessage(message)
else:
    # Insert the message
    message_table.insert(message)
```

This would definitely work in most cases, but it has some major drawbacks:

- It implements a constraint already expressed in the SQL schema, so it is a sort of code duplication.
- It execute 2 SQL queries; executing SQL query might be long and requires round-trip to the SQL server, introducing extraneous delay.
- It doesn't take into account the possibility of someone else inserting a duplicate message after we call *select_by_id* but before we call *insert*, which would cause the program to raise an exception.

There's a much better way to write this code, but it requires cooperation with the RDBMS server rather than treating it like dumb storage:

```
try:
    # Insert the message
    message_table.insert(message)
except UniqueViolationError:
    # Duplicate
    raise DuplicateMessage(message)
```

This achieves the exact same effect in a more efficient fashion and without any race condition. This is a very simple pattern, and it doesn't conflict with ORM in any way. The problem is that developers tend to treat SQL databases as dumb storage and duplicate the constraints they wrote (or could write) in SQL in their controller code rather than in their model.

Treating your SQL backend as a model API is good way to make efficient use of it. You can manipulate the data stored in your RDBMS with simple function calls programmed in its own procedural language.

Another point that needs to be raised about ORM is support for multiple database backends. Many ORM libraries tout it as a feature, but it's really a trap waiting to

ensnare unsuspecting developers. No ORM library provides a complete abstraction of all RDBMS features, so you'll have to dumb down your code to the most basic RDBMS available (or that you want to put up with), and you'll be unable to use any advanced RDBMS functions without breaking the abstraction layer.

Simple things that aren't standardized in SQL, such as handling timestamp operations, are a pain to deal with when using an ORM; even more so if your code is written to be RDBMS-agnostic. With this in mind, be sure to choose an RDBMS that suits your application well [3].

A good way to mitigate the problems with ORM libraries is to isolate them as prescribed in Section 2.3. This approach not only allows you to easily swap your ORM library for a different one should the need arise, but it also allows you to optimize your SQL usage by identifying places with inefficient usage of queries, bypassing most of the ORM boilerplate.

An easy way to build such isolation is to for example only use your ORM in a module of your application, for example `myapp.storage`. This module should only exports functions and methods that allow you to manipulate the data at a high level of abstraction. The ORM should be only used from that module. At any point later, you will be able to drop in any module providing the same API to replace `myapp.storage`.

In the end, this section's goal isn't to take a side in the debate over whether to use ORM; there's already plenty of discussion on the Internet arguing over the pros and cons. The point of this section is to help you understand how important it is to know enough about SQL and RDBMS to make use of their full potential in your application.

The most commonly used ORM library in Python (and arguably the *de facto* standard) is SQLAlchemy. It supports a huge number of different backends and provides abstraction for most common operations. Schema upgrades can be handled by third-party packages such as alembic.

[3] When in doubt, pick PostgreSQL.

Some frameworks, such as Django, provide their own ORM libraries. If you choose to use a framework, it's a smart idea to use the built-in library, which will (obviously) have better integration with the framework than an external one.

Warning

The MVC [a] architecture that most frameworks rely on can be easily misused. They implement (or make it easy to implement) ORM in their model directly, but without abstracting enough of it: any code you have in your view and controllers that uses the model will *also* be using ORM directly. This is something that you need to avoid. You should write a data model that *includes* the ORM library rather than *consists* of it: this will provide better testability and better isolation, as well as make it easier to swap out with another storage technology should the need arise.

[a] Model View Controller

12.1 Streaming data with Flask and PostgreSQL

In the previous section, we talked about how important it can be to masterize your data storage system. Here, I'll show you how you can use one of *PostgreSQL*'s advanced features to build an HTTP event streaming system.

The purpose of this micro-application is to store messages in a SQL table and provide access to those messages via an HTTP REST API. Each message consists of a channel number, a source string, and a content string. The code that creates this table is quite simple:

Example 12.1 Creating the message table

```
CREATE TABLE message (
  id SERIAL PRIMARY KEY,
  channel INTEGER NOT NULL,
  source TEXT NOT NULL,
```

```
  content TEXT NOT NULL
);
```

What we also want to do is stream these messages to the client so that it can process them in real time. To do this, we're going to use the LISTEN and NOTIFY features of *PostgreSQL*. These features allow us to listen for messages sent by a function we provide that *PostgreSQL* will execute:

Example 12.2 The *notify_on_insert* function

```
CREATE OR REPLACE FUNCTION notify_on_insert() RETURNS trigger AS $$
BEGIN
  PERFORM pg_notify('channel_' || NEW.channel,
                    CAST(row_to_json(NEW) AS TEXT));
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

This creates a trigger function written in *pl/pgsql*, a language that only *PostgreSQL* understands. Note that we could also write this function in other languages, such as Python itself, as *PostgreSQL* provides a *pl/python* language by embedding the Python interpreter.

This function performs a call to *pg_notify*. This is the function that actually sends the notification. The first argument is a string that represents a *channel*, while the second is a string carrying the actual *payload*. We define the channel dynamically based on the value of the channel column in the row. In this case, the payload will be the entire row in JSON format. Yes, PostgreSQL knows how to convert a row to JSON natively!

We want to send a notification message on each *INSERT* performed in the message table, so we need to trigger this function on such events:

Example 12.3 The trigger for *notify_on_insert*

```
CREATE TRIGGER notify_on_message_insert AFTER INSERT ON message
FOR EACH ROW EXECUTE PROCEDURE notify_on_insert();
```

And we're done: the function is now plugged in and will be executed upon each successful *INSERT* performed in the message table.

We can check that it works by using the *LISTEN* operation in *psql*:

```
$ psql
psql (9.3rc1)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.


mydatabase=> LISTEN channel_1;
LISTEN
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'hello world');
INSERT 0 1
Asynchronous notification "channel_1" with payload
"{"id":1,"channel":1,"source":"jd","content":"hello world"}"
received from server process with PID 26393.
```

As soon as the row is inserted, the notification is sent and we're able to receive it through the PostgreSQL client. Now all we have to do is build the Python application that streams this event:

Example 12.4 Receiving notifications in Python

```
import psycopg2
import psycopg2.extensions
import select
```

```
conn = psycopg2.connect(database='mydatabase', user='myuser',
                        password='idkfa', host='localhost')

conn.set_isolation_level(
    psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

curs = conn.cursor()
curs.execute("LISTEN channel_1;")

while True:
    select.select([conn], [], [])
    conn.poll()
    while conn.notifies:
        notify = conn.notifies.pop()
        print("Got NOTIFY:", notify.pid, notify.channel, notify.payload)
```

The above code connects to PostgreSQL using the *psycopg2* library. We could have used a library that provides an abstraction layer, such as *SQLAlchemy*, but none of them provide access to the LISTEN/NOTIFY functionality of PostgreSQL. It's still possible to access the underlying database connection to execute the code, but there would be no point in doing that for this example, since we don't need any of the other features the ORM library would provide.

The program listens on *channel_1*. As soon as it receives a notification, it prints it to the screen. If we run the program and insert a row in the *message* table, we get this output:

```
$ python3 listen.py
Got NOTIFY: 28797 channel_1
{"id":10,"channel":1,"source":"jd","content":"hello world"}
```

Now, we'll use Flask, a simple HTTP micro-framework, to build our application.

We're going to send the data using the **Server-Sent Events** message protocol defined by HTML5 [4].

Example 12.5 Flask streamer application

```
import flask
import psycopg2
import psycopg2.extensions
import select

app = flask.Flask(__name__)

def stream_messages(channel):
    conn = psycopg2.connect(database='mydatabase', user='mydatabase',
                            password='mydatabase', host='localhost')
    conn.set_isolation_level(
        psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

    curs = conn.cursor()
    curs.execute("LISTEN channel_%d;" % int(channel))

    while True:
        select.select([conn], [], [])
        conn.poll()
        while conn.notifies:
            notify = conn.notifies.pop()
            yield "data: " + notify.payload + "\n\n"

@app.route("/message/<channel>", methods=['GET'])
def get_messages(channel):
    return flask.Response(stream_messages(channel),
```

[4] An alternative would be to use *Transfer-Encoding: chunked* defined by HTTP/1.1.

```
                            mimetype='text/event-stream')


if __name__ == "__main__":
    app.run()
```

This application is quite simple and only supports streaming for the sake of the example. We use Flask to route a request to `GET /message/<channel>`; as soon as the code is called, it returns a response with the mimetype *text/event-stream*, sending back a generator function instead of a string. Flask will then call this function and send results each time the generator yields something.

The generator, `stream_messages`, reuses the code we wrote earlier to listen to PostgreSQL notifications. It receives the channel identifier as an argument, listens to that channel, and then yields the payload. Remember that we used PostgreSQL's JSON encoding function in the trigger function, so we're already receiving JSON data from PostgreSQL: there's no need for us to transcode it, since we're fine with sending JSON data to the HTTP client.

Note

For the sake of simplicity, this example application has been written in a single file. It isn't easy to depict examples spanning multiple modules in a book. If this were a real application, it would be a good idea to move the storage handling implementation into its own Python module.

We can now run the server:

```
$ python listen+http.py
 * Running on http://127.0.0.1:5000/
```

On another terminal, we can connect and retrieve the events as they're entered. Upon connection, no data is received and the connection is kept open:

```
$ curl -v http://127.0.0.1:5000/message/1
* About to connect() to 127.0.0.1 port 5000 (#0)
*   Trying 127.0.0.1...
* Adding handle: conn: 0x1d46e90
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x1d46e90) send_pipe: 1, recv_pipe: 0
* Connected to 127.0.0.1 (127.0.0.1) port 5000 (#0)
> GET /message/1 HTTP/1.1
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:5000
> Accept: */*
>
```

But as soon as we insert some rows in the *message* table:

```
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'hello world');
INSERT 0 1
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'it works');
INSERT 0 1
```

Data starts coming in through the terminal where *curl* is running:

```
data: {"id":71,"channel":1,"source":"jd","content":"hello world"}


data: {"id":72,"channel":1,"source":"jd","content":"it works"}
```

A naive and arguably more portable implementation of this application [5] would in-

[5] It would be compatible with other RDBMS servers, such as MySQL

stead loop over a *SELECT* statement over and over to poll for new data inserted in the table. However, there's no need to demonstrate that a push system like this one is much more efficient than constantly polling the database.

12.2 Interview with Dimitri Fontaine

I first met Dimitri a decade ago. He is a skilled PostgreSQL Major Contributor who works at 2ndQuadrant and argues with other database gurus on the *pgsql-hackers* mailing-list. We've shared a lot of open source adventures, and he's been kind enough to answer some questions about what you should do when dealing with databases.

What advice would you give to developers using RDBMS as their storage backends? What should they know about?*

That's a very good question, mainly because it offers more than one opportunity to clarify assumptions that I want to highlight as very wrong here. If you think the question as asked makes sense, you really need to be reading my answer now!

Let's start with something really boring: RDBMS stands for Relational DataBase Management System. Those beasts have been invented in the 70s to answer some common needs that every application developer needed to solve themselves at that time, and the main services RDBMS have been implementing are not data storage, as everyone knew how to implement that already.

The main services offered by a RDBMS are the following:

- Concurrency: access your data for read or write with as many concurrent **threads of execution** as you want to, the RDBMS is there to handle that correctly for you. That's the main feature you want out of a RDBMS.
- Concurrency semantics: the details about the concurrency behaviour when using a RDBMS are proposed with a high-level specification in terms of **Atomicity** and **Isolation**, that are maybe the most crucial parts of **ACID**. **Atomicity** is the property that in between the time you `BEGIN` a transaction and the time you're done with it (either `COMMIT` or `ROLLBACK`), no other concurrent activity on the system is allowed to know what you're doing, whatever that is. When using a proper RDBMS that includes **Data Definition Language** (or `DDL`, e.g. `CREATE TABLE` or `ALTER TABLE`). **Isolation** is all about what you're allowed to notice of the concurrent activity of the system from within your own transaction. The **SQL standard** defines 4 level of isolation, as described in transaction isolation documentation

The RDBMS takes full responsibility for your data. So it allows the developer to **describe** its own rules for consistency and then it will check that those rules are valid at crucial times such as transaction **commit** or statements boundaries, depending on the **deferability** of your constraints declarations.

The first constraint you can place on your data is about its expected input and output formatting, using the proper **data type**. A proper RDBMS will know how to work with much more than **text**, **numbers** and **dates**, and will properly handle dates that actually appear in a calendar in use today (**Julian** is not huge nowadays, you probably want **Gregorian** unless doing history).

Data Types are not just about input and output formats, though. They also allow to implement behaviours and some level of **polymorphism**, as we

all expect the basic equality tests to be data type specific: we don't compare text and numbers, dates and IP addresses, points boxes and lines, booleans and circles, UUIDs and XML, Arrays and Ranges in the same way, to name but a few.

Protecting your data also means that the only choice for a proper RDBMS is to actively refuse data that won't match with your consistency rules, the first of which is the data type you've chosen. If you think it's OK to have to deal with a date such as `0000-00-00` that never existed in the calendar, then you need to rethink.

The other part of the **consistency** guarantees is expressed in terms of **constraints** as in `CHECK` constraints, `NOT NULL` constraints and **constraint triggers**, one of which is known as **foreign key**. All of that can be though as a user level extension of the data type definition and behavior, the main difference being that you can choose to `DEFER` checking those constraints from being enforced at the end of each statement to being enforced at the end of the current transaction.

The **relational** bits of an RDBMS is all about modeling your data and the guarantee that all **tuples** found in a **relation** share a common set of rules: structure and constraints. When enforcing that, we are enforcing the use of a proper explicit schema to handle our data.

Working on a proper schema for your data is a process known as **Normalization** and you can aim for a number of subtly different **Normal Forms** in your design. Sometimes though, you need more flexibility than given by the result of your **Normalization** process. Common wisdom is to first normalize your data schema and only then see about how to **denormalize** it in order to get back the flexibility you think you need. Chances are that you realize you actually don't need any.

When you do need more flexibility, using PostgreSQL you can pick from

a number of **denormalisation** options: composite types, records, arrays, hstore, json or XML, for starters.

There's a very important drawback to **denormalisation** though, which is that the **Query Language** we're going to talk about next is designed to handle rather **normalized** data. With PostgreSQL of course the Query Language has been extended to support as much **denormalisation** as possible when using composite types, arrays or hstore, and even json in recent releases.

The RDBMS knows very much about your data and can help you implement a very fined grain security model, should you need to do so. The access patterns are managed at the relation and column level, and PostgreSQL also implements `SECURITY DEFINER` stored procedure, allowing you to offer access to sensible data in a very controlled way, much the same as with using `suid` programs.

The RDBMS offers you to access your data using a **Structured Query Language** which became a **de-facto** standard in the 80s and is now driven by a commitee. In the case of PostgreSQL, lots of extensions are being added with each and every major release each year allowing you to have access to a very rich **DSL** language. All the work of query planning and optimisation is done for you by the RDBMS so that you can focus on a **declarative** query where you only describe the result you want from the data you have.

And that's also why you need to pay close attention to the NoSQL offerings here, as most of those trendy products are in fact not just removing the **Structured Query Language** out of the offering, but a whole lot of other foundations that you've been trained to expect.

My advice to developers is to remember the differences between a **storage backend** and a RDBMS. Those are very different services, and if all you need actually is a storage backend, maybe consider not using a RDBMS.

Most often though, what you really need is a full blown RDBMS. In that case, the best option you have is PostgreSQL. Go read its documentation, see the list of data types, operators, functions, features and extensions it provides. Read some usage examples on blog posts.

Then consider PostgreSQL as a tool you can leverage in your development, and include it in your application architecture. Parts of the services you need to implement are best offered at the RDBMS layer, and PostgreSQL excels at being that trustworthy part of your whole implementation.

What's the best way to use or not use ORM?

SQL stands for **Structured Query Language** and in the case of PostgreSQL has been proven to be **Turing Complete**. Its implementation and optimizer are far from trivial.

As ORM stands for **Object Relational Mapper**, the idea is that you can deal with a one-to-one mapping of database relations with classes and database tuples with objects, or class instances.

Even when a RDBMS, like PostgreSQL, implements strong static typing, relation definitions are built on the fly: each query result is a new relation. Each subquery result is a new relation that might exists only for the duration of the subquery. Each `JOIN`, either **INNER** or **OUTER**, will result in a new relation dynamically built for solving just that `JOIN`.

As a direct consequence of that, it's easy to understand that where the **ORM** will be able to best work for you is for what's called **CRUD** applications: **Create**, **Read**, **Update** and **Delete**. The **Read** part should then only be limited to a very simple `SELECT` statement targeting a single table. If you compare non-trivial **output lists** you can measure the impact of retrieving more columns than necessary on query performances. Now,

if your **ORM** is including all the known fields in its **projections** (or output list), then it will force your RDBMS to fetch external data (and decompress) it before sending it, maybe only to compress it again if you're using **SSL** in between the RDBMS and your application. Also, just think about network bandwidth usage and remember than we're measuring simple **primary key** based lookup queries in fractions of a **millisecond**.

So any column you retrieve from the RDBMS and that you end-up not using is pure waste of precious resources, a first scalability killer.

Even when your **ORM** of choice is well able to only fetch the data you're asking for, then you have to somehow manage the exact list of columns you want in each situation, and avoid using a simple abstract magic method that will automatically compute the fields list for you.

The next part of the **CRUD** queries are simple `INSERT`, `UPDATE` and `DELETE` statements. First, all those commands accept joins and sub-select when you're using an advanced RDBMS, such as PostgreSQL. Then again, for example PostgreSQL implements the `RETURNING` clause, allowing you to return to the client any data that's just been edited, such as **default** (typically sequence numbers for surrogate keys) and other values **computed** automatically on the RDBMS (typically with `BEFORE <action>` triggers).

Is your **ORM** aware of that? What's the syntax there to benefit from that?

In the general case, a relation is either a table, the result of calling a **Set REturning Function**, or the result of any query. It's common practice when using an **ORM** to build a **relational mapping** in between defined tables and some **model classes**, or some other helper stubs.

If you consider the whole SQL semantics in their generalities, then the **relational mapper** should really be able to map any query against a class. You would then presumably have to build a new class for each query you

want to run.

The legend of the Sufficiently Smart Compiler applies to ORMs too. For more details about what that legend is, read On Being Sufficiently Smart by James Hague.

The idea when applied to our very case is that you trust your **ORM** to do a better job than you at writing efficient SQL queries, even when you're not giving it enough information to even work out the exact set of data you are interested into.

It's true that at times, SQL can get quite complex. You're not going to get anywhere near simpler by using an API to SQL generator that you can't control, though.

After having said all that against the typical **ORM**, something needs to be said against the alternatives.

Building **SQL** queries as a string is not scalable. You want to be able to compose several **restrictions** (the `WHERE` clauses) and dynamically add some joins right into a subquery just so that you can optionally fetch some more detailed data, etc.

My current thinking is that the tool you really want to have is not an **ORM**, it's a nice way to compose a **SQL** query from a programmatic interface.

There's a PostgreSQL driver proposing exactly the right abstraction to that problem, it's the **Common Lisp** library Postmodern with the S-SQL solution. Of course, **Lisp** lends itself really well to allow for easy to program **composable** components.

Actually in two cases you can relax and use your ORM, provided that you're willing to accept the following compromise: as soon as possible you will need to edit your ORM usage out of your code base.

- **Time To Market**; When you're really in a hurry and want to gain market

share as soon as possible, the only way to get there is to release a first version of your application and idea. If your team is more proficient at using an ORM when compared to hand crafting SQL queries, then by all means just do that. You have to realize, though, that as soon as you're successful with your application, one of the first scalability problems you will have to solve is going to be related to your ORM producing really bad queries, and your usage of the ORM having painted you into a corner and bad code design decisions. But if you're there, you're successful enough to spend some refactoring money and remove any dependency toward the ORM, right?

- **CRUD Application**; the real thing, where you are only editing a single tuple at a time, and you don't really care about performances. Like for the basic admin application interface.

Are there any pros or cons to choosing PostgreSQL over other databases when working with Python?

Here are my top reasons for choosing PostgreSQL as a developer:

- Community support: the PostgreSQL community really is welcoming to new users, and will typically spend the time it takes to fully understand your question before to answer the best possible answer. The mailing lists are still the best way to communicate with the community. See PostgreSQL Mailing Lists for details.
- Data integrity and durability: any data you send to PostgreSQL is **safe** in its definition and your ability to fetch it again later.
- Data Types, function, operators, arrays and ranges: PostgreSQL has a very rich set of data types that are really useful and come with a host of operators and functions to process them. It's even possible to denormalize using **arrays** or **JSON** data types, and still be able to write

advanced queries including joins against those. For example, did you know about the ~ regular expression operator? and the `regexp_split_to_array` and `regexp_split_to_table` functions?

- The planner and optimizer: you have to try to push the limits you know about those to really understand how complex and powerful they are. I've repeatedly seen 2 to 3 pages long queries run to complement in a small number of milliseconds.
- Transactional DDL: it's possible to `ROLLBACK` almost any command. Try it now, just open your `psql` shell against a database you have and type in `BEGIN;DROP TABLE foo;ROLLBACK;` where you replace **foo** with the name of a table that exists in your local instance. Amazing, right?
- `INSERT INTO ...RETURNING`: you can return anything from the `INSERT` statement directly, like for example the `id` value that got derived from a sequence. You win a network round-trip and get the result with the same protocol and tools as when issuing a `SELECT` statement.
- `WITH (DELETE FROM ...RETURNING *) INSERT INTO ...SELECT`: PostgreSQL support **Common Table Expression** in queries, which are known as **WITH queries**, and thanks to its support for the `RETURNING` clause, it also supports **DML** commands there. That's just awesome, rith?
- Window Functions, `CREATE AGGREGATE`: if you don't know what a window function is, go read about it in the PostgreSQL Manual or in my blog at Understanding Window Functions. Then you have to realise that PostgreSQL allows you to use any existing **aggregate** as a window function, and allows you to dynamically define new aggregates online in SQL.
- PL/Python (and others such as C, SQL, Javascript or Lua): you can run your own code on the server, right where the data is, so that you don't have to fetch it over the network just to process it then send it back in a query to do the next level of JOIN. Whatever it is, you can do it all on the

server.

- Specific Indexing (GiST, GIN, SP-GiST, partial & functional): did you know that you can create Python functions to process your data from within PostgreSQL, then index the result of calling that function? So that when you issue a query with a `WHERE` clause calling that function, it's called only once with the data from the query, then it's matched directly with the contents of the index? PostgreSQL implements index frameworks for non sortable data types, like 2 dimensional types (ranges, geometry, etc); and for container data types. Lots of cases are already supported out of the box, and a host more thanks to the **Extension** system. Have a look at the Additional Supplied Modules and the PostgreSQL Extension Network.
- Extensions: such extensions include **hstore**, a full blown key value store with flexible indexing, **ltree** for indexing nested tags, **pg_trgm** as a poor man's full text search solution, that supports indexing regular expression searches and unanchored **LIKE** queries, **ip4r** for quick searches of an IP address in a range, and a lot more.
- Foreign Data Wrappers: the **foreign data wrappers** are a whole class of extensions, implementing the SQL/MED standard (Management of External Data). The idea is to embed a connection driver right into the PostgreSQL server then expose it through the `CREATE SERVER` command. PostgreSQL provides an API to **foreign data wrapper** authors that allows them to implement read and write access to the remote data, and also `where` clauses push-down for efficient joining capabilities. You can even use the advanced SQL capabilities of PostgreSQL against data that you maintain with another piece of technology!
- `LISTEN/NOTIFY`: PostgreSQL implements an asynchronous server-to-client protocol called `LISTEN/NOTIFY`. The application may receive unsolicited

messages from the server when something interesting happens, for example an UPDATE of some data. The `NOTIFY` command accepts a data payload so that you can e.g. notify your cache application the object id's to purge when the object just has been removed or updated. Of course, the notification only happens if the transaction actually did a successful `COMMIT`.

- `COPY` Streaming protocol: PostgreSQL implements a **streaming** protocol and uses it to implement its fully integrated replication solution. Now, that protocol is quite easy to use from an application and allows impressive performance boosts. As soon as you're working on more than a dozen row at a time, sometimes before, thing about using `COPY` against a **temporary table** then issuing a single statement joining to that temporary table: PostgreSQL knows how to join against other tables in all data modifying statements (**insert**, **update**, **delete**), and batch operation usually are way faster.