Migrating to PostgreSQL

Tools and Methodology

Dimitri Fontaine

January 2018

Contents

1 Preface

This white paper describes how to migrate from another relational database server technology to PostgreSQL. The reasons to do so are many, and first among them is often the licensing model.

PostgreSQL open source license model is very simple to understand: the software is free for anyone to use, study, modify and redistribute. As a user, that means that you are allowed to use PostgreSQL in any way you want. You have full flexibility when it comes to design the best production architecture to suits your needs, in contrast to what can happen with other proprietary options.

A migration to PostgreSQL must be approached as a project, and some risks need to be handled properly. This book presents a complete methodology that you can follow in order to migrate to PostgreSQL and minimize any risks while doing so.

The migration method proposed here is called ***Continuous Migration***. Continuous Migration makes it easy to make incremental progress over a period of time, and also to pause and resume the migration work later on, should you need to do that. The method is pretty simple — just follow those steps:

1. Setup your target PostgreSQL architecture
2. Fork a *Continuous Integration* environment that uses PostgreSQL
3. Migrate the data over and over again every night, from your current production RDBMS

4. As soon as the CI is all green using PostgreSQL, schedule the D-day
5. Migrate without any suprises... and enjoy!

2

Migration Projects

On February the 18th, 2015 I received a pretty interesting mention on Twitter:

> *@tapoueh thank you for pgloader, it made our migration from mysql to postgresql really easy (~1Tb)*

This tweet from CommaFeed is still available online, and their story follows:

> *@tapoueh it was almost too easy, I just ran the one-liner and waited for 48 hours. Nothing to change in the app, thanks to hibernate.*

While this is awesome news for this particular project, it is still pretty rare that having to change your connection string is all you need to do to handle a migration!

If you're less lucky than *CommaFeed*, you might want to prepare for a long running project and activate the necessary resources, both servers and people availability. Even if you're using an ORM and never wrote a SQL query yourself, your application is still sending SQL queries to the database, and maybe not all of them can be written in a PostgreSQL compatible way.

That said, it is still worth trying to run the migration just like *CommaFeed* did!

Reasons to Migrate to PostgreSQL

Before starting the migration to PostgreSQL, you need to rehash the reasons why you're initiating such a project. It might take you a lot of effort and time to complete the migration. Also, it being an IT project, things might become stalled at some point, resources might have to be reallocated in the middle of the project, and chances are that the project will take more time than anticipated. It could even run over time and over budget!

If any of this happens, it's important to remember why you're taking on the project in the first place, lest it be cancelled.

Also, the methodology that this white paper introduces allows the project to be suspended and resumed with as little negative impact as possible.

Classic reasons to migrate to PostgreSQL include the following:

- Total cost of ownership

  Migrating to PostgreSQL usually reduces the TCO a lot, because there's no licensing cost, neither for the RDBMS itself nor for some of its most advanced features. This lack of invoicing doesn't depend on the CPUs, vCPUs or the number of concurrent users you want to work with.

- Production architecture freedom

  The lack of licensing cost also means that you may run as many standby servers as you want, or even deploy a full-blown container based architecture in a testing environment just to better understand the trade-offs involved. There's no added cost.

- Advanced features out of the box

  PostgreSQL comes in a single package with all the features enabled, for free. This includes point in time recovery, hot standby, logical replication, table partioning, advanced data types such as JSONB and full text search, six different kinds of indexes to address all your use cases, server programming including *stored procedures* that you can write using many different programming languages — from SQL to Python — and so much more.

Using the same *PostgreSQL licence* you can also add functionality to PostgreSQL using PostgreSQL *extensions*. See the list of contrib extensions on the PostgreSQL documentation page entitled Additional Supplied Modules, and make sure you install the PostgreSQL *contrib* package in your environments.

PostgreSQL *contrib* extensions are maintained by the same development team as the core PostgreSQL code, and other extensions are available that are maintained outside of the main development tree. Externaly maintained extensions include ip4r, an IPv4/v6 and IPv4/v6 range index type for PostgreSQL, postgresql-hll which adds HyperLogLog data structures as a native data type as detailed in the Citus Data article Distributed count(distinct) with HyperLogLog on Postgres and many more. Have a look at the PGXN extension registry!

- SQL standard compliance

  You might want to adhere to the SQL standard as much as possible, and PostgreSQL is one of the most compliant database technologies around when it comes to the SQL standard.

  Examples of SQL standard features included by default in PostgreSQL include common table expression, window functions and grouping sets, hypothetical-set aggregate functions and more.

  PostgreSQL also extends the standard nicely, for instance by adding support for data-modifying statements in WITH or the ability to create your own aggregate and window functions thanks to the CREATE AGGREGATE command.

- Open source licence

  The PostgreSQL Licence is a derivative of the BSD and MIT licenses and allow any usage of the software. So if you want to embed PostgreSQL in your product, you can do so without impacting your own choice of software licensing.

Migrating to PostgreSQL only to reduce the *total cost of ownership* of your RDBMS production setup is not the best idea. Take some time to analyze your situation in light of the other benefits that PostgreSQL is going to bring to the table for you.

Typical Migration Budget

A typical migration budget is divided up into these areas:

1. Cost of migrating the data
2. Cost of migrating the code
3. Cost of migrating the service
4. Opportunity cost

Usually the most costly part of the migration budget is the second part here: migrating the code. This means you should provide enough resources (time, people availability, CI/CD setup, QA slots, etc) to make this part go more smoothly, but you should still consider the other parts of the budget.

Migrating the Data

This part should be quite easy. Put simply, it's all about moving data defined in terms of relations, attributes and attribute domains from one technology to another. The basics of using an RDBMS is that it's easy to push data in, and it's easy to process and get the data out of the system.

So why do we even list *migrating the data* as an item in our migration budget? It turns out that the *attribute domains* are defined in different ways in different RDBMS, but under the same name. Attribute domains are also known as data types, and here are some oddities:

- An empty *text* value might be equivalent to NULL, or something distinct from NULL, depending on the RDBMS.
- A *datetime* attribute — or *timestamp with time zone* in standard SQL and PostgreSQL terms — might accept the year zero, when there's no such date in our calendar.

  Read the Wikipedia entry on Year zero for more details:

  > *. . . the year 1 BC is followed by AD 1.*

- Some systems deal with *unsigned* numeric values, allowing one to extend the range of positive values, and PostgreSQL doesn't handle *unsigned* data types by default.

- PostgreSQL requires full encoding validation before accepting any text, which is not always the case with other systems. This will cause difficulties when migrating.

  We can find many different encodings in the same column in a MySQL database, and I've been told about a case where more than ten different encodings were found in a table with only 100,000 rows.

- Collation rules might differ from one system to another. For instance, MySQL used to default to sorting texts with case insensitive Swedish rules. If your users are used to that and you don't want to change the sorting rules when migrating to PostgreSQL, then you will have to pick the right collation rules. It won't happen automatically.

- Constraint management and error handling differ among RDBMS. Some might accept NULL input for NOT NULL attributes, and then automatically replace the NULL value with some DEFAULT value, either the one from the table definition or an hard-coded one.

As a result, migrating the data from another RDBMS to PostgreSQL might take more effort than anticipated, and this part of the project should get your full attention.

Also, the lack of constraint enforcement means that even if you can migrate your production data set at one point in time, new data might show up later that you won't be able to migrate with the same tooling or processing rules.

Finally, you might have some historical records that you don't need in the new version of your software or product. In that case, isolating those parts and having a special migration process for it is worthwhile, as it can amount to loads of data. Migrating a lot of data you don't need will slow down every part of the process for no good reason, and in some cases the proper way to handle the data is to simply forget about it.

It's still possible to automate the entire data migration process, as we are going to see in more detail in the Tooling: pgloader section later on.

Migrating the Code

Application code embeds SQL queries and relies on RDBMS behavior in both obvious and non obvious ways. This means that when migrating, you need to be able to run all your automated tests against your software or product and have them pass with PostgreSQL.

Migrating from one RDBMS to another one is a rare case in which using an ORM rather than writing SQL might prove to be helpful, as the ORM might be able to compile your queries to another system. Then it's all about editing the ORM configuration.

In many cases though, even when using an ORM, some SQL rewriting is necessary. In all cases, serious testing is required.

The main challenge with migrating the code is to confirm that your software still works exactly the way it used to. If you have a QA team, this might require manual QA work.

Migrating the Service

So your data is under control and you know how to migrate it to PostgreSQL. You've been migrating the whole production data set several times already, and you have a CI/CD environment that uses PostgreSQL. Your code runs in that environment and passes all the tests, and the QA department gives a green light for the migration. It's time to migrate the service.

The usual process looks like this:

1. Setup the new PostgreSQL based architecture
2. Deploy the PostgreSQL compliant code in the new architecture
3. Switch the production into read-only mode, or shut it down
4. Migrate the data to the new PostgreSQL service architecture
5. Run the pre-opening tests (CI, QA, etc)
6. Switch the production on for everyone

Duplicating a whole production setup is easier to consider when *running in the cloud*. Otherwise, it may be the case that only the database servers are being replaced. Maybe you're switching to PostgreSQL on the same

production gear as you had before, which renders the process more complex to handle, and the *rollback* paths is also complex.

Migrating the service usually isn't where you'll spend most of your migration budget. It nonetheless requires proper planning, testing and resources. It's so much easier to migrate to a distinct environment with its own hardware; you might want to consider whether it makes sense in your case to simplify the problem in that way.

On the other hand, if your service has the luxury of having a long enough *maintenance window* to be able to run the fourth step above, *migrate the data*, all within this window, things are going to be much simpler. That's another reason why removing all useless data in the migration process ends up being so useful: it reduces the maintenance window needs when migrating the service.

Opportunity Cost

Finally, as the migration to PostgreSQL is a real project to which you are assigning resources, it means that while those resources are working for the migration, they can't be working for another project.

The opportunity cost is hard to compute. It might be a list of projects that you won't be able to deliver to the marketing, product, sales, and finance departments while working on the migration project rather than a budget expressed in dollars.

Port vs Migration

According to the Wikipedia page on porting, a software port is defined as such:

> *In software engineering, porting is the process of adapting software for the purpose of achieving some form of execution in a computing environment that is different from the one that a given program (meant for such execution) was originally designed for (e.g. different CPU, operating system, or third party*

> *library). The term is also used when software/hardware is changed to make them usable in different environments.*

This follows:

> *Software is portable when the cost of porting it to a new platform is significantly less than the cost of writing it from scratch. The lower the cost of porting software, relative to its implementation cost, the more portable it is said to be.*

In the Wikipedia page on software migration, we also read the following:

> *Legacy modernization, or software modernization, refers to the conversion, rewriting or porting of a legacy system to a modern computer programming language, software libraries, protocols, or hardware platform. Legacy transformation aims to retain and extend the value of the legacy investment through migration to new platforms*

In the context of migrating a system to PostgreSQL, we use the following terminology in this document:

- *Port*

  A port allows an existing system to also run with PostgreSQL, where it didn't before.

  One example of a software port is when you're a software editor and you want to be able to provide more options to your customers. Once the port is done, your customer will have a choice of running your software with PostgreSQL rather than another RDBMS.

  When your aim is to *port* your software to PostgreSQL, one of the goals is to have a minimal set of changes in terms of the software architecture.

- *Migration*

  In all other cases, your software runs with a single RDBMS at a time, and you can specialize your code and software architecture to benefit from PostgreSQL.

  So in a typical migration project, you're free to reconsider your production architecture and software architecture, to the extent that makes sense for you.

> Usually, while migrating some software to PostgreSQL, the architecture is only adapted in a way that simplifies the migration itself, and simplified the timely maintenance of the new system once it's in production.

Whether you're having to deal with a *port* or a *migration* in your project, some of your process will have to change:

- Your current *backup and recovery* tooling might not support PostgreSQL out of the box.
- Differences in how to implement *point in time recovery* and *hot standby* can impact your current high availability solution, and it's better to have a fully optimized and dedicated solution per RDBMS rather than something unreliable supporting many RDBMS at once.
- Some application code paths might need to be specialized depending on the RDBMS *provider* that's being used in production, so in the process of migrating your code and service to PostgreSQL some application processes might have to be rewritten from scratch.

If you're familiar with the notion of *porting* a software to a new platform, then you should be familiar with the idea of having specialized modules that only run in one of the supported platforms. The RDBMS is part of the platform here.

More About PostgreSQL

Developers should know about PostgreSQL before migrating to this amazing piece of technology. Mastering PostgreSQL in Application Development is the book to read if you want to be ready to make the most out of PostgreSQL from day one.

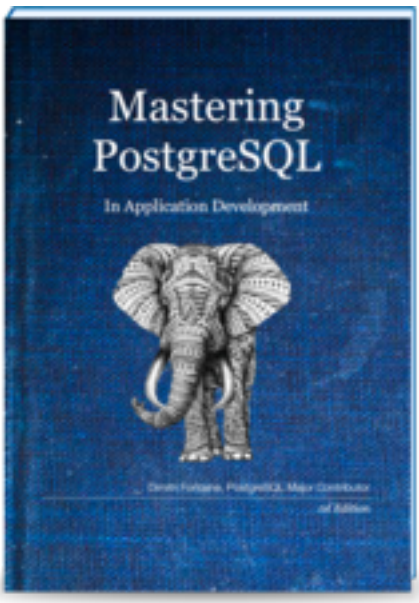


Figure 2.1: Mastering PostgreSQL in Application Development

3

Continuous Migration

Migrating to PostgreSQL is a project in its own right, and requires a good methodology in order to be successful. One important criteria for success is being able to deliver on time and on budget. This doesn't mean that the first budget and timing estimates are going to be right — it means that the project is kept under control.

The best methodology to keep things under control is still a complex research area in our field of computer science, though some ideas seem to keep being considered as the basis for enabling such a goal: split projects in smaller chunks and allow incremental progress towards the final goal.

To make it possible to split a migration project into chunks and allow for incremental progress, we are going to implement ***continuous migration***:

- Continuous migration is comparable to continuous integration and continuous deployments, or *CI/CD*.
- The main idea is to first setup a target PostgreSQL environment and then use it everyday as developers work on *porting* your software to this PostgreSQL platform.
- As soon as a PostgreSQL environment exists, it's possible to fork a *CI/CD* setup using the PostgreSQL branch of your code repository.
- In parallel to porting the code to PostgreSQL, it's then possible for the ops and DBA teams to make the PostgreSQL environment pro-

duction ready by implementing backups, automated recovery, high availability, load balancing, and all the usual ops quality standards.

Setting up a continuous migration environment does more than allow for progress to happen in small chunks and in parallel with other work — it also means that your team members all have an opportunity to familiarize themselves with the new piece of technology that PostgreSQL might represent for them.

PostgreSQL Architecture

In order to be able to implement the *Continuous Migration* methodology, the first step involves setting up a PostgreSQL environment. A classic anti-pattern here is to simply host PostgreSQL on a virtual machine and just use that, putting off production-ready PostgreSQL architecture until later.

Here's some reasons why it's best to begin with production ready PostgreSQL Architecture right from the start of your migration project:

- When PostgreSQL is a new technology for your teams, then ops people and DBA will need training time to get used to how PostgreSQL works and figure out which tools to use to maintain the service in production.

  By setting-up a production ready environment from the start of the migration project, you then have a training opportunity that helps get team members up to speed in a test environment.

- Even if the PostgreSQL service isn't critical for the end users yet, it's already critical for the developers on your team, making the environment more than just a sandbox.

- Some architecture choices for your PostgreSQL services are going to impact how to write the code, and likewise the other way around.

  When implementing load balancing for reporting and other read-only queries, it's easier to debug the application code when it runs on read-only PostgreSQL standby servers.

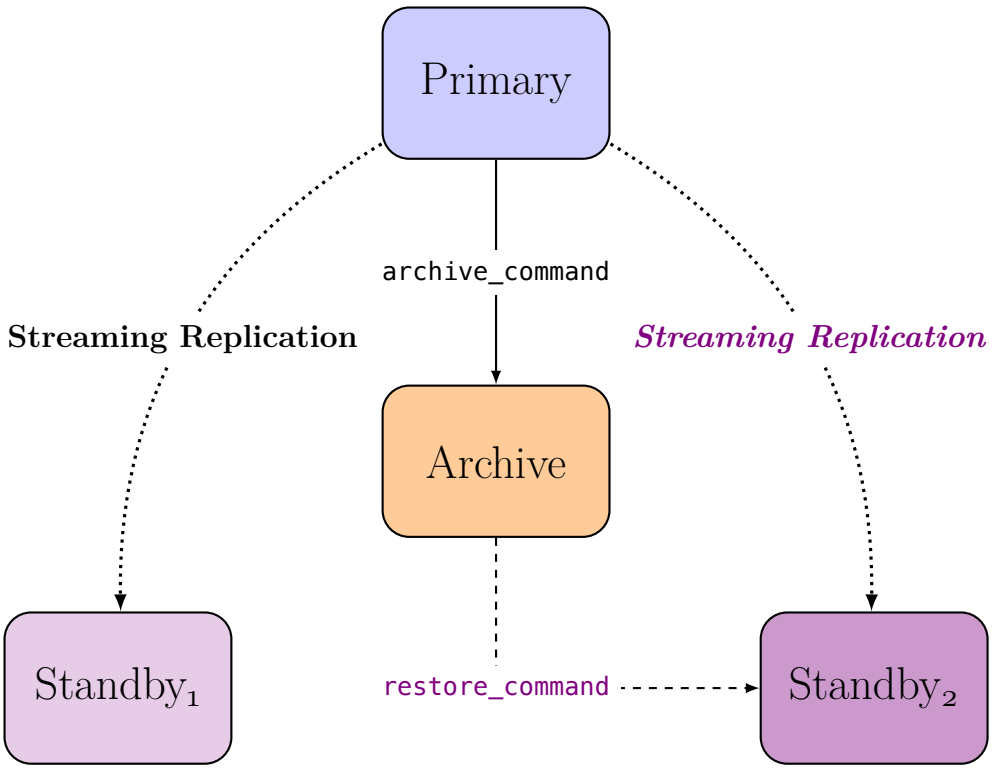
Some functionality that your previous RDBMS provided might not be found in PostgreSQL, such as Oracle scheduling packages.

- Your continuous integration pipeline can now run on-top of a working PostgreSQL architecture too, meaning that you will find bugs earlier and you can fix them before the application users get bitten.

With that in mind, your PostgreSQL migration project now begins with an assessment of your application needs in terms of RDBMS usage, and designing a PostgreSQL architecture that satisfies these needs.

High Availability

A very classic PostgreSQL architecture to begin with involves WAL archiving and a standby server, and it looks like this:



Automated Recovery

In the previous schema you can see the generic terms `archive_command` and `restore_command`. Those PostgreSQL configuration hooks allow one to implement WAL archiving and *point in time recovery* thanks to management of an archive of the *change log* of your production database service.

Now, rather than implementing those crucial scripts on your own, you can use production ready WAL management applications such as pgbarman or pgbackrest. If you're using cloud services, have a look at WAL-e if you want to use Amazon S3.

Don't roll your own *PITR* script. It's really easy to do it wrong, and what you want is an entirely automated **recovery** solution, where most projects would only implement the backup parts. A backup that you can't restore is useless, so you need to implement a *fully automated recovery solution*. The projects listed above just do that.

PostgreSQL Standby Servers

Once you have an *automated recovery* solution in place, you might want to reduce the possible downtime by having a *standby server* ready to take over.

To understand all the details about the setup, read all of the PostgreSQL documentation about high availability, load balancing, and replication and then read about logical replication.

Note that the PostgreSQL documentation is best in class. Patches that add or modify PostgreSQL features are only accepted when they also update all the affected documentation. This means that the documentation is always up-to-date and reliable. So when using PostgreSQL, get used to reading the official documentation a lot.

If you're not sure about what to do now, setup a PostgreSQL Hot Standby physical replica by following the steps under hot standby. It looks more complex than it is. All you need to do is the following:

1. Check your `postgresql.conf` and allow for replication
2. Open replication privileges on the network in `pg_hba.conf`
3. Use `pg_basebackup` to have a remote copy of your *primary* data

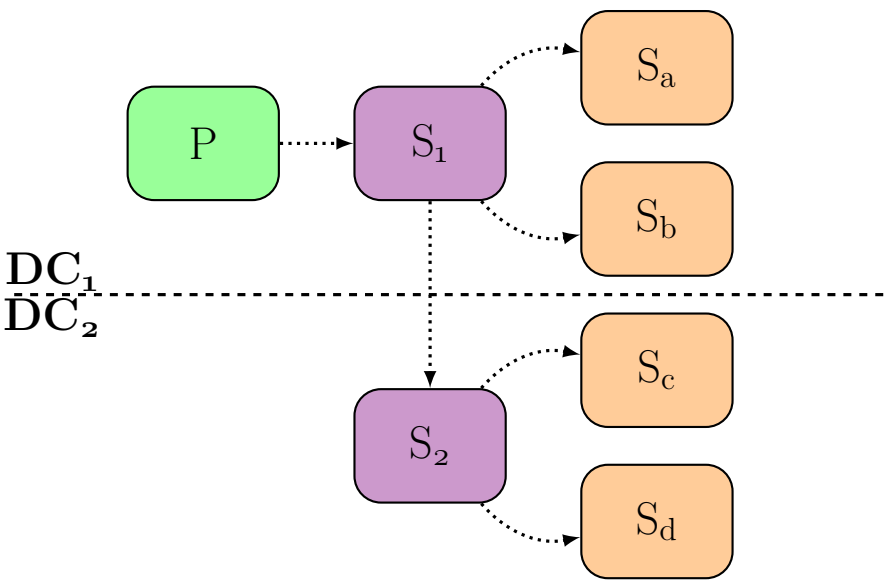
4. Start your *replica* with a setup that connects to the *primary*

A PostgreSQL *hot standby* server replays its primary *write ahead log*, applying the same binary file level modifications as the primary itself, allowing the standby to be a *streaming copy* of the primary server. Also the PostgreSQL *hot standby* server is open for read-only SQL traffic.

> It's called a *hot standby* because not only is the server open for read-only SQL queries — this read-only traffic also doesn't get interrupted in case of a standby promotion!

Load Balancing

PostgreSQL standby servers are available for read-only SQL queries. This means that it's possible to use them for simple load balancing needs, as in the following architecture diagram:



You can also see that in this diagram, we have two levels of cascading standby servers, which is supported out of the box when using PostgreSQL.

The load balancing replicas and the high availability replicas must implement different trade-offs in terms of replaying the *write ahead log*, so it's best to cleanly separate the servers having different roles.

Fancy Architectures

Of course it's possible to setup more complex PostgreSQL architectures. Starting with PostgreSQL 10 you can use *logical replication*. It's easy to setup, as seen in the logical replication quick setup part of the PostgreSQL documentation.

With *logical replication* it is possible to implement *federated servers* and some kinds of *sharding* architectures too.

Do you need such a setup to get started migrating your current application to PostgreSQL though? Maybe not.. it's your call.

Nightly Migration of the Production Data

Chances are that once your data migration script is tweaked for all the data you've gone through, some new data is going to show up in production that will defeat your script.

To avoid data related surprises on D-day, just run the whole data migration script from the production data every night, for the whole duration of the project. You will have such a good track record in terms of dealing with new data that you will fear no surprises. In a migration project, surprises are seldom good.

> *"If it wasn't for bad luck, I wouldn't have no luck at all."*
>
> **Albert King**, Born Under a Bad Sign

We'll see how to implement this step in the Tooling: pgloader section of this document.

Migrating Code and SQL Queries

Now that you have a fresh CI/CD environment with yesterday's production data every morning, it's time to rewrite those SQL queries for PostgreSQL. Depending on the RDBMS your are migrating from, the differences in the

SQL engines are either going to be mainly syntactic sugar, or sometimes there will be completely missing features.

Take your time and try to understand how to do things in PostgreSQL if you want to *migrate* your application rather than just *porting* it, that is making it run the same feature set on top of PostgreSQL rather than your previous choice.

Again, the book Mastering PostgreSQL in Application Development is a good companion when learning how to best use PostgreSQL and its advanced SQL feature set.

Data Model and Normalization

When optimizing the query planner and optimizer, the PostgreSQL development team mainly consider normalized data models. This means that in a lot of cases, PostgreSQL is quite strong at implementing *join* operation in a fast way.

When porting code from another RDBMS to PostgreSQL, watch out for manual implementation of join operations in the client code. When you see a loop over a result set, and more processing happening inside the loop, consider these questions:

- Should the processing really be part of the SQL query?
- Is the processing another SQL query sent to the server for each row retrieved in the outer query? If yes, it's better to implement a join operation in the outer query and just be done with it.

PostgreSQL also implements composite data types. That means that it knows how to encode several bits of information in a structured way within a single attribute, or column. Composite data types include Arrays, JSON, Composite Types, and Range Types.

In addition to those, by default PostgreSQL also includes some powerful data types that you might not be used to. Those data types can help a lot when it comes to simplifying your application code, or improve SQL queries. For instance, PostgreSQL includes UUID, Network Address Types, Boolean, and Geometric Types.

It's not just text and numbers, and dates. Also, when mentioning dates, default to always using *timestamp with time zone* in PostgreSQL.

With that in mind, you can audit your application schema and port it over to PostgreSQL. It's also possible to port your schema to be as close as possible to your previous version, and improve it as you go.

It's important to realize what PostgreSQL has to offer to help developers write less application code though, and it's easy enough to review the schema before it goes into production. It's still possible once in production of course, but then it's not as easy.

Standard SQL and Implementations

PostgreSQL implements the SQL standard and extends it. To get a quick overview of how different RDBMS implement the standard, you can check out the comparison of different sql implementations page from Troels Arvin.

Raw SQL

Your application code is sending SQL statements to PostgreSQL, and some of them are good, and some of them are failing and need some attention, debugging and rewriting. It's not always that easy to find where a SQL statement comes from.

First, you can parse PostgreSQL logs for errors in SQL queries. You can use the pgbadger tool to help you in that task.

In order to know where the query comes from in your code, you can also use PostgreSQL setting application_name, and then have it show up in the logs. The `application_name` may be set dynamically in your SQL sessions with the SET SQL command. It's a good idea to have each logical part of your application code use its own `application_name` setting.

Stored Procedures

The SQL standard includes the SQL/PSM part, which stands for *persistent stored modules*. It doesn't seem like any RDBMS is implementing this standard though, even if both Oracle PL/SQL and PostgreSQL PL/pgSQL are inspired from it.

It means that porting stored procedures from another RDBMS to PostgreSQL is going to mostly be a manual task, and it's going to take some time. It's possible in theory to write a PL compiler that translates from one language to another, and some non open source tooling does exist.

You can also contribute to pgloader and help us implement such a feature. Have a look at the pgloader Moral Licence, also introduced later in this paper.

Triggers

SQL triggers are implemented as stored procedures with a specific calling protocol. So porting triggers shares the characteristics of porting stored procedures, as seen above.

Views and Materialized Views

In SQL, a view is a query to which we assign a name, and that is registered and maintained server-side. As a result, a SQL view embeds the RDBMS specific SQL dialect.

This means that porting SQL views from another RDBMS to PostgreSQL would again require an SQL compiler that is able to automatically compile from one SQL dialect to another, and no such tool is available as open source.

Again, consider contributing to the pgloader Moral Licence if you want to see such a compiler developed with an open source license.

Test Coverage

Running your *CI/CD* environment on top of PostgreSQL also means that your test coverage and tests success ratio are good on-going indicators of your migration project. It also allows your team to work in smaller chunks, see progress, and switch to other urgent tasks should there be something else that comes up.

4

Tooling: pgloader

Migrating the data should be easy. After all, your source database is already a relational database server and implements the same SQL basics as any other one: relations, tables, views, and the expected column data types, or attribute and attribute domains.

It is actually quite impressive to realize how many differences exist between different technologies when it comes to implementing data types, default values, and other behaviors.

When using pgloader, all the knowledge when it comes to those differences is integrated into an open source product.

General Migration Behavior

pgloader follows those steps when migrating a whole database to PostgreSQL:

1. Fetch meta data and catalogs

   pgloader fetches the source database meta data by querying the live database catalogs. Retrieving structured data makes it easier to process the complex level of information. Catalog queries that are used can be seen in the pgloader source code at GitHub.

In cases in which a database schema *merge* is required — that's called the ORM compatibility mode, where the schema is created by a tool other than pgloader — then pgloader also fetches the target PostgreSQL metadata and compares it to the source catalogs.

The information retrieved by pgloader is handled internally in its own in-memory catalog format, and then used throughout the migration tasks. It is possible to implement some *catalog mapping* in these internal pgloader catalogs, as seen later in this chapter.

2. Prepare the target PostgreSQL database

   Unless the database has already been created by another tool, pgloader creates the target PostgreSQL database schema by translating the source catalogs into PostgreSQL specific catalogs.

   This steps involves *casting rules* and the advanced command language included in pgloader allows the user to define custom casting rules.

   The casting rules are applied to the schema definition (column types, or attribute domains) and default values.

3. Copy table data using PostgreSQL copy protocol

   For each table fetched in the first step, pgloader starts two threads:

   - The reader thread issues a `select` statement that retrieves all the data from the source database, sometimes already with some data processing.
   - The writer thread receives the data read from the other thread thanks to a shared queue, and batches the data to be sent over to PostgreSQL.

     While arranging a batch of data, pgloader also applies transformation functions to the data. The transformations applied depend on the casting rules. Examples include transforming a *tinyint* value into a *Boolean* value, or replacing date time entries `0000-00-00 00:00:00` with NULL values.

   It's possible to setup how many threads are allowed to be active at the same time on a pgloader run, and the data migration timing is influenced by that setup.

4. Create the indexes in parallel

   For each table, as soon as the entire data set has made it to PostgreSQL, all the indexes defined on the table are created in parallel. The reason why pgloader creates all the indexes in parallel is PostgreSQL's implementation of synchronize_seqscans:

   > This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload. When this is enabled, a scan might start in the middle of the table and then "wrap around" the end to cover all rows, so as to synchronize with the activity of scans already in progress. This can result in unpredictable changes in the row ordering returned by queries that have no ORDER BY clause. Setting this parameter to off ensures the pre-8.3 behavior in which a sequential scan always starts from the beginning of the table. The default is on.

   For a large table, having to scan it sequentially only once to create many indexes is a big win of course. Actualy, it's still a win with smaller tables.

   A *primary key* index is created with the `ALTER TABLE` command though, and PostgreSQL issues an *exclusive lock*, which prevents those indexes from being created in parallel with the other ones. That's why pgloader creates primary key indexes in two steps. In this step, pgloader creates a unique index over not-null attributes.

5. Complete the target PostgreSQL database

   Once all the data is transfered and all indexes are created, pgloader completes the PostgreSQL database schema by installing the constraints and comments.

   Constraints include turning the candidate unique not-null indexes into primary keys, then installing foreign keys as defined in the source database schema.

Once all those steps are done, your database has been converted to PostgresQL and it is ready to use!

As explained in the previous part about ***Continuous Migration***, the main idea behind pgloader is to implement fully automated database migrations so that you can run this as a nightly task for the whole duration of your migration project.

Sample Output

It's possible to run a whole database migration with a single command line, using only pgloader defaults, like this:

```
$ createdb -U user dbname
$ pgloader mysql://user@host/dbname \
           pgsql://user@host/dbname
```

Here's a sample output of that command:

```
                table name     errors       rows      bytes      total time
-------------------------  ---------  ---------  ---------  --------------
          fetch meta data          0         33                     0.325s
           Create Schemas          0          0                     0.001s
         Create SQL Types          0          0                     0.008s
            Create tables          0         26                     0.202s
           Set Table OIDs          0         13                     0.008s
-------------------------  ---------  ---------  ---------  --------------
             f1db.circuits          0         73    8.5 kB           0.039s
  f1db.constructorresults          0      11052   184.6 kB           0.252s
       f1db.constructors          0        208    15.0 kB           0.054s
            f1db.drivers          0        840    79.6 kB           0.094s
          f1db.laptimes          0     417743    10.9 MB           6.320s
                    ...
            f1db.results          0      23597     1.3 MB           0.987s
             f1db.status          0        134     1.7 kB           0.068s
-------------------------  ---------  ---------  ---------  --------------
  COPY Threads Completion          0          4                     6.468s
          Create Indexes          0         20                     2.347s
   Index Build Completion          0         20                     1.458s
         Reset Sequences          0         10                     0.127s
            Primary Keys          0         13                     0.021s
     Create Foreign Keys          0          0                     0.000s
         Create Triggers          0          0                     0.001s
        Install Comments          0          0                     0.000s
-------------------------  ---------  ---------  ---------  --------------
        Total import time          ✓     511270    14.0 MB         10.422s
```

The output clearly shows three sections: prepare the target database schema, transfer the data and complete the PostgreSQL database schema. Performance and timning will vary depending on hardware used. This sample is using a 2012 laptop optimized for traveling light.

As a result, don't read too much into the numbers shown here. Best

practices for doing the database migration include having different source hardware and target machines in order to maximize the disk IO capacity, and also a high bandwitdth network in between the servers. Using the same disk both as a source for reading and a target for writing is quite bad in term of performance.

Catalog Mapping

It is possible to change some catalog entries on the fly when doing a database migration. Supported changes include:

- Limiting tables to take care of,
- Migrating from one schema name to another one,
- Renaming tables.

It is also possible (only with MySQL as a source at the moment) to migrate data from a view definition rather than a table, allowing schema redesign on the fly.

Data Type Casting Rules

In order for the database migration to be fully automated, pgloader allows users to define their own casting rules. Here's a `sample.load` command file implementing this idea:

```
load database
     from mysql://root@unix:/tmp/mysql.sock:3306/pgloader
     into postgresql://dim@localhost/pgloader

 alter schema 'pgloader' rename to 'mysql'

 CAST column base64.id to uuid drop typemod drop not null,
      column base64.data to jsonb using base64-decode,

      type decimal when (and (= 18 precision) (= 6 scale))
        to "double precision" drop typemod

 before load do $$ create schema if not exists mysql; $$;
```

Custom casting rules may target either specific columns in your source database schema, or a whole type definition.

pgloader Features

Many users already use pgloader happily in their production environment and are quite satisfied with it. The software comes with a long list of features and is well documented at http://pgloader.readthedocs.io/en/latest/.

Supported RDBMS

pgloader already has support for migrating the following RDBMS in a fully automated way:

- DBF
- SQLite
- MySQL
- MS SQL Server

On the Road Map

The following projects are part of the pgloader road map:

- New database sources
  - Oracle support
  - Sybase support
  - IBM DB2 support, via an ODBC driver
- New SQL objects support
  - View definitions
  - Stored procedures
- Quality and delivery
  - More frequent releases
  - Binaries for all platforms, including Windows
  - Continuous Integration for database sources

pgloader Moral License

The pgloader project is fully open source and released under the *PostgreSQL license* so anyone can easily contribute to the project. All of the project management (issue tracking, feature proposals, etc) happens on the pgloader github page, which is public.

There are two ways to contribute to pgloader and its road map. As it's a fully open source project, you can of course implement the feature yourself and contribute it to the project. To do that, fork the project on github and get started, then submit a pull request, as is usually done.

If you don't want to do that — maybe because you don't have enough time to both hack pgloader and migrate your database — then consider contributing financially to the project and buy a pgloader Moral Licence online. It's easy to do and it's a big help!

pgloader Customers

Here's what our customer *Andrea Crotti* from *Iwoca* has to say about using and sponsoring pgloader:

> *Thanks to pgloader we were able to migrate our main database from MySQL to Postgres, which involved moving hundreds of tables used by our complex Django project. Dimitri was very helpful. He implemented a new feature for us quickly and smoothly.*

Here's what our customer *Alexander Groth* from *FusionBox* has to say about using and sponsoring pgloader:

> *Fusionbox used pgloader on a project for a large government agency. We needed to migrate a large set of data from an existing SQL Server cluster to a new PostgreSQL solution. Pgloader greatly reduced the time required to accomplish this complex migration.*

And when Redpill Linpro was asked about the most efficient way to migrate from Microsoft SQL Server to our beloved PostgreSQL by one of their customers, they asked me to add support for SQL Server in pgloader.

Thanks to Redpill Linpro now everyone can do the same, as the result is open source and fully integrated into pgloader!

5 Closing Thoughts

Migrating from one database technology to PostgreSQL requires solid project methodology. In this document we have shown a simple and effective database migration method, named ***Continuous Migration***:

1. Setup your target PostgreSQL architecture
2. Fork a continuous integration environment that uses PostgreSQL
3. Migrate the data over and over again every night, from your current production RDBMS
4. As soon as the CI is all green using PostgreSQL, schedule the D-day
5. Migrate without any unpleasant suprises... and enjoy!

This method makes it possible to break down a huge migration effort into smaller chunks, and also to pause and resume the project if need be. It also ensures that your migration process is well understood and handled by your team, drastically limiting the number of surprises you may otherwise encounter on migration D-day.

The third step isn't always as easy to implement as it should be, and that's why the pgloader open source project exists: it implements fully automated database migrations!

For your developers to be ready when it comes to making the most out of PostgreSQL, consider reading Mastering PostgreSQL in Application Development which comes with an Enterprise Edition that allows a team of up to 50 developers to share this great resource.