PostgreSQL transition guide: supporting the decision

Socle Interministériel du Logiciel Libre, PGGTIE

PostgreSQL.fr

09/11/2021

Contents

1 Versions **5**

2 Contributors **6**

3 Introduction **7**

4 Satisfying requirements **8**
4.1 Use cases 8
4.2 Data access 8
4.3 Security requirements 8
4.4 Migration cases 9

5 Integration of PostgreSQL on technical platforms **10**
5.1 Technical platforms 10
5.1.1 Processor architectures 10
5.1.2 Operating systems 10
5.1.3 Compatibility with virtualization/containerization 10
5.2 Running PostgreSQL in the cloud 10
5.2.1 Compatibility with storage technologies 11
5.2.2 Compatibility with backup technologies 11
5.3 Security 11
5.3.1 User identification management 11
5.3.2 Confidentiality guarantees and use of encryption 11
5.3.3 Ensure traceability and logging 12
5.3.4 Audit 12

6 Scalability and resilience **13**
6.1 Clustering and replication mechanisms 13
6.2 Resilience driven by the PostgreSQL engine 14
6.3 Resilience driven by the storage infrastructure 14
6.4 Business continuity and disaster recovery plans 14

7 Development with PostgreSQL **15**
7.1 Data types 15
7.2 Database schemas and structures 15
7.3 Development specifics 16
7.4 API and access modes 16
7.4.1 Client interfaces 16

7.4.2 Abstraction layer 17
7.4.3 Connection pooling 17
7.5 Stored procedures, functions, and triggers 17
7.6 Foreign Data Wrappers 18

8 Administration 19
8.1 Monitoring and exploitation tools for PostgreSQL 19
8.1.1 PgAdmin 19
8.1.2 psql 19
8.2 Monitoring tools and logs analysis 19
8.3 Backup 20
8.4 High-Availability 20
8.5 Data migration tools to PostgreSQL 20

9 Decision's Help 22

10 Limits or unsupported features 23

11 Change management 25
11.1 Training 25
11.2 Support 25
11.3 Migration plan 25

12 Return on investment 26
12.1 Database migration cost 26
12.2 Possesion cost 26
12.3 Control of trajectories 26

13 Appendices 27
13.1 Migration from Oracle 28
13.1.1 Technical environment 28
13.1.2 Software stack 28
13.1.3 Database migration 28
13.1.4 Criticality and backups 30
13.1.5 Monitoring, performance improvement and supervision 30
13.2 Migration from Db2 31
13.2.1 Some DDL differences between PostgreSQL and Db2 31
13.2.2 DCL 34
13.2.3 Other considerations 35

13.3 Migration from Informix . . . 37
13.3.1 Structure . . . 37
13.3.2 Plan to follow when migrating databases . . . 37
13.3.3 Integrating the Hibernate framework . . . 38
13.3.4 Risks . . . 38
13.4 Migration from MSSQL . . . 39
13.5 Some references . . . 40
13.6 Extensions and plugins for PostgreSQL . . . 41
13.7 Third-party tools for PostgreSQL . . . 42

14 References **43**

1 Versions

History of this document versions:

Version	Date	Comments
0.1	18/12/2013	First draft
1.0	19/03/2015	Publication of the first version in French
1.0.1	30/09/2015	Russian translation by I. Panchenko and O. Bartunov
2.0	10/12/2019	Update by the PGGTIE
2.0.1	08/06/2020	Removed DINSIC mention
3.0.0	09/11/2021	English version and licence change

2 Contributors

Initial authors:

- Bruce BARDOU - DGFiP
- Barek BOUTGAYOUT - MASS
- Amina CHITOUR - MENMESR
- Alain DELIGNY - MEDDE
- Yohann MARTIN - Maif
- Alain MERLE - MEDDE
- Anthony NOWOCIEN – Société Générale
- Loïc PESSONNIER - MINDEF
- Marie-Claude QUIDOZ - CNRS
- Laurent RAYMONDEAU - MENESR

3 Introduction

This guide was produced within the framework of the « Ayrault » circular of September 19, 2012 on the use of free software in the French administration, which led to the creation of the inter-ministerial « Socle Interministériel du Logiciel Libre » (SILL). This guide presents the implementation of PostgreSQL compared to commercial solutions. Its objective is to answer the questions of project owners and project managers for the implementation of PostgreSQL as a replacement for a commercial solution. This guide aims, without going too deep into the details of the technical implementation, to demonstrate the benefits of PostgreSQL by describing the integration, security and robustness mechanisms.

The initial document was authored collaboratively by Mimprod[1] and PGGTIE[2].

[1]Mimprod: https://www.mim-libre.fr/mimprod-outils-de-production/
[2]PGGTIE: https://www.postgresql.fr/entreprises/english

4 Satisfying requirements

4.1 Use cases

PostgreSQL is an SQL compliant database management system, typically used in transactional applications to ensure data persistence, much like similar commercial products (Oracle, DB2, Informix, MS SQL, Sybase, etc).

There is a PostgreSQL extension for geographic objects (PostGIS[3]) that conforms to the Open Geospatial Consortium (OGC) Standards. It supports JSON format data, consisting of key-value pairs, allowing it to also satisfy the use cases of NoSQL-like solutions. PostgreSQL can also be used in the field of Business Intelligence as a data warehouse, in conjunction with reporting tools (BusinessObjects, Pentaho, etc). It comes with a built-in and feature rich full-text search engine[4].

Many free software packages are natively based on PostgreSQL (document management systems (DMS), rules engines, collaboration software, supervision software, etc). More and more software vendors are supporting PostgreSQL. Open letters, like this one written by the PGGTIE [5] , could also help in adoption.

PostgreSQL also supports background processing, such as batch or deferred processing.

4.2 Data access

PostgreSQL conforms[6] to the SQL:2016 standard, the common query language of many RDBMSs. Migration is therefore facilitated between RDBMSs respecting the standard.

4.3 Security requirements

PostgreSQL meets security needs in terms of availability, integrity, confidentiality and traceability (information security[7]).

Encryption is standard and several strong authentication methods (like SCRAM and Kerberos) are possible. Cryptographic requirements are covered by additional modules, in particular *pgcrypto* [8]. At present, it is not possible to encrypt an entire instance, but work is ongoing (see wiki on Transparent Data Encryption [9]).

[3]PostGIS: https://postgis.net/
[4]Full-text search: https://www.postgresql.org/docs/current/textsearch.html
[5]Open letter: https://www.postgresql.fr/entreprises/20171206_open_letter_to_software_vendors
[6]SQL standard compliance: https://www.postgresql.org/docs/current/features.html
[7]Information security: https://en.wikipedia.org/wiki/Information_security
[8]pgcrypto: https://www.postgresql.org/docs/current/pgcrypto.html
[9]Transparent Data Encryption: https://wiki.postgresql.org/wiki/Transparent_Data_Encryption

PostgreSQL has a very responsive community that provides security patches and manages component obsolescence. It is recommended that you apply minor patches as soon as possible. A major version is released every year and supported [10] for 5 years.

PostgreSQL guarantees the integrity of the data handled even in the event of an incident by respecting the properties of atomicity, consistency, isolation and durability (ACID [11]).

PostgreSQL natively offers the mechanisms to meet confidentiality and rights management needs through roles [12]. Very fine granularity can be achieved, even at the row level (Row Security Policies [13]).

4.4 Migration cases

The migration of a system represents a significant cost, consisting mainly of:

- data migration, even if there are a number of tools to perform this migration, depending on the source RDBMS;
- changes needed in the source code of the application. The need for changes depends on the use of any functionalities that are specific to the existing product (like stored procedures, triggers, and non-standard functions) and the use of an abstraction layer (like Hibernate);
- steps to guarantee equivalent functionalities and that no errors were introduced during the migration (regression tests).

As such, migrations to PostgreSQL are often carried out during a functional evolution of the application. The cost of migration is integrated into the project as a whole, particularly in terms of technical and functional acceptance.

However, an ambitious global migration plan can be arranged. PostgreSQL can become the preferred RDBMS for any new application, with projects having to justify specific needs for another RDBMS (exemption request).

[10] Support and EOL: https://www.postgresql.org/support/versioning/
[11] ACID properties: https://en.wikipedia.org/wiki/ACID
[12] Roles: https://www.postgresql.org/docs/current/user-manag.html
[13] Row Security Policies: https://www.postgresql.org/docs/current/ddl-rowsecurity.html

5 Integration of PostgreSQL on technical platforms

5.1 Technical platforms

5.1.1 Processor architectures

PostgreSQL runs on the following processor architectures[14]: x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL and PA-RISC.

5.1.2 Operating systems

PostgreSQL works on most operating systems. Binary versions exist for the Red Hat family (including CentOS / Fedora / Scientific), the Debian GNU / Linux family and its derivatives, the Ubuntu Linux family and its derivatives, SuSE, OpenSuSE, Solaris, Windows, macOS, FreeBSD, OpenBSD etc.

For distributions using .rpm files, it is possible - and recommended - to use the community repository https://yum.postgresql.org/. For those using .deb, please refer to https://wiki.postgresql.org/wiki/Apt. Source code is also available online[15].

5.1.3 Compatibility with virtualization/containerization

PostgreSQL is compatible with VMWare and KVM in particular. Virtualization brings resiliency benefits. It is also possible to make PostgreSQL work with containers like Docker.

5.2 Running PostgreSQL in the cloud

All major cloud vendors have a (or sometimes multiple) managed PostgreSQL offer. Being a managed service means that the vendor will handle the infrastructure and administrative operations and grant you restricted privileges on the instance.

Their offerings will vary vastly according to the SLA they can provide, the flavors of instances available, their use of a fully community version or a proprietary one, the list of available extensions, the hih availability / disaster recovery solutions provided,…

[14] Supported platforms: https://www.postgresql.org/docs/current/supported-platforms.html
[15] Source code: https://www.postgresql.org/ftp/source/

5.2.1 Compatibility with storage technologies

PostgreSQL works on storage arrays (SAN, NAS, etc) but like any DBMS, the attachment must be permanent and at block level. Storage virtualization is completely transparent to the DBMS.

Communities provide recommendations on the types of filesystem to use (ext4, XFS, ZFS, etc). The use of NFS is not recommended.

5.2.2 Compatibility with backup technologies

PostgreSQL supports the usual backup methods:

- cold backup : file system level backup - the database must be offline ;
- hot backup : basic filesystem level backup supported. No possibility to perform an incremental backup natively. Tools like Barman[16] and pgBackRest[17] allow this, though ;
- continuous backup: file system level backup supported, allowing *Point In Time Recovery* (PITR). It is possible to use the tools mentioned above, or *pitrery* ;
- SQL dump : the database can be exported in SQL format, optionally compressed, in a format specific to PostgreSQL.

PostgreSQL is compatible with other backup tools as well (TINA, TSM, Veeam, Netbackup, Bacula, etc).

5.3 Security

5.3.1 User identification management

PostgreSQL provides authentication mechanisms and manages the attribution of privileges (GRANT, etc). It also allows separation through database schemas.

5.3.2 Confidentiality guarantees and use of encryption

Encryption is possible via the *pgcrypto* module, allowing column encryption by public key and private key. Transparent Data Encryption (TDE) is not yet available in community PostgreSQL.

[16] Barman: https://www.pgbarman.org/
[17] pgBackRest: https://pgbackrest.org/

5.3.3 Ensure traceability and logging

The traceability of events in PostgreSQL is ensured by transaction logs (called *WAL*, or Write-Ahead Logs):

- logging of engine operations (starting, stopping, etc) ;
- access logging (queries, user access, errors, etc).

5.3.4 Audit

While a significant amount of information can be recorded in the logs[18], it may be necessary in some cases to implement a more extensive audit policy. The most successful tool to do this is *PGAudit*[19]. In particular, it will make it possible to understand the context in which an operation was performed. An example implementation by the United States Defense Information Systems Agency (DISA) is available[20].

[18] Log configuration: https://www.postgresql.org/docs/current/runtime-config-logging.html
[19] PGAudit: https://www.pgaudit.org/
[20] DISA Security Technical Implementation Guide: https://www.crunchydata.com/disa-postgres-security-guide/disa-postgres-security-guide-v1.pdf

6 Scalability and resilience

6.1 Clustering and replication mechanisms

Definitions

Scalability and Elasticity

> Scalability is the ability of PostgreSQL to adapt to an order of magnitude change in demand (up or down).

> The elasticity of a system is the ability of a system to automatically increase or decrease its resources as needed.

Resilience and Robustness (or stability)

> Resilience is the ability of a system or network architecture to continue to function in the event of failure.

> Robustness is the quality of a system that does not crash, that functions well even in a hostile (penetration, DoS, etc) or abnormal (e.g. incorrect inputs) environment.

Cluster

> A cluster is a group of servers (or « compute farm ») sharing common storage. A cluster provides high availability and load balancing functions.

Replication

> Replication is a process of sharing information to ensure data consistency between multiple redundant data sources, to improve reliability, fault tolerance, or availability.

As with the vast majority of RDBMSs, it is easy to increase server resources (CPU, RAM, disk) to vertically scale the Postgres service. Certain solutions also allow you to perform sharding (distribute data over several instances, such as *Citus*, *PostgreSQL-XL*, etc).

Several clustering and replication modes are supported by PostgreSQL, with advantages and disadvantages to each. These considerations are independent of the choice of DBMS though; all products are affected :

- "Active-Passive" mode : a primary database is read / write, another database, the "standby", is synchronized in the background. Replication can be asynchronous (better performance) or synchronous (better security).

- “Partial Active-Active” or “Read/Write - Read” mode: a primary database is read / write, the “standby” databases are read-only.

Like other DBMSs, PostgreSQL does not allow transactions on several servers in “Active-Active” or “Read/Write - Read/Write” mode. You have to go through third-party contributions, such as EDB’s BDR[21] (Bidirectionnal Replication), that offer this functionality. However, this is a proprietary development on a modified version of PostgreSQL.

6.2 Resilience driven by the PostgreSQL engine

PostgreSQL provides replication mechanisms for setting up an “active-passive” or “partial active-active” cluster.

There are two main types of replication, namely physical replication (eg *streaming replication* [22] based on modification of data blocks) and logical replication (based on modification of database objects). For high availability requirements, we will focus on physical replication. Logical replication is intended for other use cases, like the selective supply of another database (e.g. datawarehouse), an upgrade with the aim of minimizing downtime, etc.

6.3 Resilience driven by the storage infrastructure

PostgreSQL is compatible with clustering driven by the platform, independently of the DBMS. Writing to a node is performed by PostgreSQL and replication to a second node is handled by the storage array.

To restart the engine on the second node, there must be an interruption of service.

6.4 Business continuity and disaster recovery plans

Business continuity and disaster recovery plans must be supported by the application. Recommendations for the choice of scenarios are made according to the recovery and continuity needs, independently of the DBMS.

PostgreSQL provides the tools for the integration of recovery and continuity plans. Several solutions are available for switchover and manual/automatic failover.

[21] BDR: https://www.2ndquadrant.com/en/resources/postgres-bdr-2ndquadrant/
[22] Streaming replication https://www.postgresql.org/docs/current/warm-standby.html#STREAMING-REPLICATION

7 Development with PostgreSQL

7.1 Data types

PostgreSQL offers standard data types (alphanumeric, date, time, BLOB, etc…) as well as more complex data types (geospatial, XML, JSON, etc…). We refer the reader to the documentation[23].

It is recommended to use ISO 8601[24] for dates (YYYY-MM-DD). For data including date and time, the *timestamp with time zone* data type should be used.

The *Binary Large OBject* (BLOB) data type allows storage of binary form content in the database (office files, PDFs, photos, audio, video, multimedia, etc).

PostgreSQL provides two ways of storing binary data :

- directly in the table using the *bytea* type;
- in a separate table with a special format that stores binary data and refers to it by a value of type *oid* in the main table using the Large Object function.

The *bytea* data type, which can hold up to 1 GB, is therefore not suitable for storing very large amounts of binary data. The Large Object function is best suited for storing very large volumes. However, it has its own limits :

- deleting a record does not delete the associated binary data. This requires a maintenance action on the database;
- any connected user can access binary data even if they do not have rights on the main database.

The use of BLOB columns is not recommended if there is no need to search the information.

7.2 Database schemas and structures

Table partitioning is implemented as standard[25]. It is worth noting a significant improvement with V10, which allows declarative partitioning. It was previously achieved through to the notion of inheritance and the writing of triggers. Version 11 continues the improvements, in particular on the possibility of having primary/foreign keys on the partitions, partition-pruning, the ability to define a default partition, etc.

Previously one would rather avoid creating more than 500/1000 partitions for a single table. With the latest versions that have brought very appreciable performance gains in this area, this is not a relevant limit anymore.

[23] Data types: https://www.postgresql.org/docs/current/datatype.html
[24] ISO 8601: https://en.wikipedia.org/wiki/ISO_8601
[25] Partitioning: https://www.postgresql.org/docs/current/ddl-partitioning.html

7.3 Development specifics

PostgreSQL complies with the SQL 2016 standard.

The documentation[26] provides links to see which functions are actually covered and which are not yet: overall the coverage of the standard is very good, although it is surprising that the `MERGE` statement is not yet supported. PostgreSQL does, however, have an `INSERT ... ON CONFLICT`[27] statement that meets a similar need.

As such, PostgreSQL does not require significant query syntax training. However, developers' habits regarding the use of specific functions and particular syntaxes for joins in their usual DBMS will require support. For example, the date functions are often specific to each DBMS (see Appendices by DBMS).

It is possible to create custom function libraries to simulate the specific functions of other DBMSs, facilitating both migration and training.

The choice of character set is done according to the applications. UTF-8 is recommended for the whole chain. ISO can also be used if needed.

Be careful with the default options when installing PostgreSQL: if no options are specified, *initdb* and `CREATE DATABASE` use the server's language configuration, either LATIN9 (ISO 8859-15) or, failing that, ASCII (https://www.postgresql.org/docs/current/app-initdb.html). To use UTF-8, you must specify it when creating the database.

PostgreSQL allows the use of materialized views. While they can be refreshed on demand (and with minimal blocking via the `CONCURRENTLY` clause), it is currently not natively possible to continuously update them. This could be possible through the use of triggers but would come with a very important performance penalty.

7.4 API and access modes

7.4.1 Client interfaces

PostgreSQL provides several client interfaces for accessing data.

Name	Language	Comments	Address
JDBC	Java	Type 4 JDBC driver	https://jdbc.postgresql.org/
psqlODBC	ODBC	ODBC driver	https://odbc.postgresql.org/
psycopg	Python	DB API 2.0-compliant	https://initd.org/psycopg/

[26] SQL conformance: https://www.postgresql.org/docs/current/features.html
[27] INSERT ON CONFLICT: https://www.postgresql.org/docs/current/sql-insert.html#SQL-ON-CONFLICT

Name	Language	Comments	Address
Npgsql	.NET	.NET data provider	https://www.npgsql.org/
DBD::Pg	Perl	Perl DBI driver	https://metacpan.org/release/DBD-Pg
libpqxx	C++	New-style C++ interface	http://pqxx.org/development/libpqxx/
pgtclng	Tcl	Tcl interface	https://sourceforge.net/projects/pgtclng/

7.4.2 Abstraction layer

The use of an "ORM" type abstraction layer (object/relation mapping) is supported.

7.4.3 Connection pooling

A connection manager is strongly recommended for several hundred or more simultaneous connections. As PostgreSQL uses a process for each new session, establishing and closing a connection can become slow in relation to query execution time. Overloading your database with active sessions is also a risk. *PgBouncer*[28] is stable and performant – we recommend it, even if it is single–threaded. *Pgpool-II*[29] is more complex to use and is recommended for load-balancing.

7.5 Stored procedures, functions, and triggers

A trigger is an action (stored procedure or SQL query) executed on an event. Triggers have many uses, including implementation of traces related to the application (error management, activity, etc).

A stored procedure is a program stored in the database. This is usually a function, and version 11 allows you to actually define a procedure, invoked via the CALL command.

Stored procedures bring adherence to the chosen RDBMS and might render future hypothetical migrations more difficult. Using PostgreSQL to its full extend and using its specific features will trade portability for improved performance. Deciding which one is more important is an architectural question you will have to answer.

Functions can be used to extend the use of pre-existing data types.

PostgreSQL allows you to write functions and procedures in languages other than SQL and C. These other languages are generically called procedural languages. There are currently four procedural languages in the standard PostgreSQL distribution:

[28] PgBouncer: https://www.pgbouncer.org/
[29] Pgpool-II: https://www.pgpool.net/mediawiki/index.php/Main_Page

- PL/pgSQL: SQL procedural language ;
- PL/Tcl: Tcl procedural language ;
- PL/Perl: Perl procedural language ;
- PL/Python: Python procedural language.

There are other procedural languages that are not included in the main distribution (like PL/Java, PL/PHP, PL/Py, PL/R, PL/Ruby, PL/Scheme, PL/sh, …). Other languages can be defined by users, but the process can be a bit complex.

7.6 Foreign Data Wrappers

These are extensions that allow PostgreSQL to communicate with other data sources. The data sources can be relational databases (PostgreSQL, MySQL, Oracle, etc), NoSQL databases (CouchDB, MongoDB, etc), CSV files, or LDAP directories. The fact that the data comes from other sources is transparent to the end user.

Some FDWs, like Oracle and MySQL, have read/write capabilities; others only read.

For a complete list, see the wiki[30].

[30] FDW: https://wiki.postgresql.org/wiki/Foreign_data_wrappers

8 Administration

8.1 Monitoring and exploitation tools for PostgreSQL

8.1.1 PgAdmin

PgAdmin is an client-server administration tool, open source an released under the PostgreSQL licence. It is available on all plateforms and is included by default during a Windows installation. You can download it there : https://www.pgadmin.org/download/.

As a GUI tool, it has a low entry barrier. Here are some of its most important features : graphical query tool, display customization, addons (for example PostGIS, Shapefile and DBF loader) and a script langage. It can also display the query plan result (`EXPLAIN` command) in a graphical way.

It can be installed in server mode to avoid installation on the client's end. This software is updated frequently.

8.1.2 psql

psql tool is a command line utility, with which we can write SQL query, display database schemas, import/export data… It is included in all PostgreSQL distributions and we recommand it as the default client.

To have help for SQL commands, we can use `\help`. Starting from version 12, this command will also give the documentation link of the instruction. For meta-command's help, we have to use `\?`.

8.2 Monitoring tools and logs analysis

Monitoring tools (Nagios, Munin, …) have plugins for PostgreSQL and offer monitoring of log and system tables . The Bucardo's Perl script *check_postgres* is the most important plugin for this tools. Many dedicated tools are available, like for example *temBoard* and *pgwatch2*.

Other modules can store database statistics evolution (like the highly recommended *pg_stat_statements*[31], *pgtop*, …) or analyze logs (*pgBadger*[32]).

It is important to check the health and update activity of the modules used. For example, some modules that used to be popular like *pgFouine*, *pgstatpack*, … are not maintained anymore.

[31] pg_stat_statements: https://www.postgresql.org/docs/current/static/pgstatstatements.html
[32] pgBadger: https://pgbadger.darold.net/

8.3 Backup

Even if PostgreSQL can natively handle hot backup, Point In Time Recovery, … it might be necessary to industrialize the backup process, either with a in-house development (very difficult) or through the use of a dedicated backup tool.

Among the most advanced tools, we will highlight *Barman* and *pgBackRest*. They provide:

- easier PITR ;
- incremental backup ;
- a centralized backup catalog, which can come from different servers ;
- backup retention management;
- …

8.4 High-Availability

Several tools exists to improve resiliency and to make administration of an group of instances easier We will note for example :

- *repmgr*: facilitate the setting up of a complex architecture and ease some administrative actions like failover/switchover/adding an instance…
- *Patroni*: A template to set up a high availability solution. It is usually used with containers and with a distributed configuration tool like *etcd* or *Consul*.
- *PAF*: PostgreSQL Automatic Failover is based on Pacemaker/Corosync to manage high availiblity and the possibilty to switch roles in an automated way.

8.5 Data migration tools to PostgreSQL

The tool *ora2pg*[33] can analyze an Oracle database. Evaluating migration costs was an additional feature required by France's DGFIP. The tool can also realize the migration.

For data transfer, we can use an ETL (Talend Open Studio, *pgloader*[34], …) that can apply transformations to the data during transfert (data type adaptation …). For an initial load and in case of important volume, it is possible to use a tool like *pg_bulkload* that will showcase better performance then the COPY command.

The application code must be adapted and needs to be translated in SQL instructions compatible with PostgreSQL. The tool *code2pg*[35] can help by giving an estimation and ways to transform the

[33] ora2pg: https://ora2pg.darold.net/
[34] pgLoader: https://pgloader.io/
[35] code2pg: https://github.com/societe-generale/code2pg

instructions.

It is highly recommended to test the migration in the same conditions as the production’s environnment (production database, application session logs,…).

9 Decision's Help

Even if PostgreSQL can answer a lot of different use cases, some projects might be better served by other technologies. This chapter will attempt to describe a decision process as it being used in several companies.

PostgreSQL has a very liberal licence in its community version and there is no extra software cost when deploying it. If a warranty is needed, it is possible to subscribe to support on either the community version or a proprietary one that will include some extra features. An often used strategy within enterprises is to have a "PostgreSQL first" policy for all new projects, baring any lack of functionality or vendor's requirements.

Concerning migrations, their ROI might be difficult to justify and can at times be inexistant. It will depend on the project complexity, potential licence cost savings, … It may be simplier to have a application upgrade and a database migration at the same time. Even if PostgreSQL is recognized as a very capable RDBMS on small to medium sized databases, it can also handle databases of dozens or hundreds of terabytes.

The table below will be list :

- features;
- native support (starting from which version) or with which extension (and its version);
- optional comments and limitations to be aware of.

Features	Availability	Comments - Limits
Connection pooler	with pg-pool/pgbouncer …	see § 7.4.3
Master-standby replication without load-balancer	9.0 +	see § 6.1
High availibility with automatic failover and connection routing	with Patroni/repmgr modules…	see § 8.4
Encrypting data	with pgcrypto extension (8.3 +)	see § 4.3
Actvity audit	with plugin pgaudit (9.5 +)	see § 5.2.4

Features	Availability	Comments - Limits
NoSQL	json (9.2+)	see § 4.1
Materialized views	9.3 +	see § 7.3
Declarative table partioning	10 +	see § 7.2/12.1.4
LOBs management	7.2+	
Spatial data	with Postgis	see § 4.1/7.1/12.5
Monitoring and supervision	with tem-board/pgwatch2/…	
UPSERT	9.5 +	
Autonomous transactions	with plugin pg_background	
Indexes		
Maintenance and data loading tools…	7.2+ (Vacuum) 7.1+ (Copy)	
auto-explain	with auto_explain extension	
Full-text search	8.3 +	
Storage, backup externalisation	pg_basebackup or with Bar-man/pgBackRest/…	see § 5.1.5/6.3/8.3

10 Limits or unsupported features

This paragraph lists some unsupported features in the community version. This is of course not a comprenhensive list.

- Instance level encryptiin;
- Master-master clusters ;
- Advanced compression ;
- History snapshot of statistics;

Regarding SQL compliance, a list is available on https://www.postgresql.org/docs/current/unsupported-features-sql-standard.html.

11 Change management

Switching to another DBMS is a project on its own and depending on the scope, can become an enterprise level project. It might therefore require managemenent's involvement to be successful.

11.1 Training

Training sessions can focus on these audiences :

- developers : getting started and PostgreSQL migrations;
- architect and operators : installation, maintenance, backup, monitoring …;
- advanced administrators (DBA) : ops training.

11.2 Support

There are many ways to get support within the PostgreSQL community:

- documentation;
- mailing lists / forums / IRC / Slack;
- by contracting commercial support with a vendor;
- within user/company groups (like PGGTIE).

Professional support is also possible, we will refer the reader to the support page[36].

11.3 Migration plan

A migration will be a more or less important adaptation of the application. Regression and performance testing will have to be conducted and the migration cost might be significant.

Whenever the migration can be conducted at the same time as a major functional evolution, this cost can be lowered. Functional testing and technical tests would have had to be conducted, no matter the underlying RDBMS. This has to be balanced by the fact that there are risks at managing two migrations simultaneously.

Project management (both on the business and technical part) and ops support for skill improvement will have to be planned.

[36] Support: https://www.postgresql.org/support/professional_support/europe/

12 Return on investment

12.1 Database migration cost

A migration to a new DBMS will have to address the following points :

- entry cost (training, raising the level of competencies,…);
- application ajustement including test costs (technical, functional and non regression test);
- any renewal of the support market;
- the upgrade of the operating procedures.

12.2 Possesion cost

PostgreSQL is under an open source license[37] similar to the BSD or MIT ones. So there is no pricing policy from an editor that the project must undergo.

12.3 Control of trajectories

PostgreSQL's roadmap is known and controlled.

The PostgreSQL's ecosystem keeps on growing with new plugins and new tools. Commercial software offers more and more interfaces and api to use PostgreSQL.

Its community is both important and active. To take a French example, the reader can have a look at the mailing list[38] and the forum[39]. The inter enterprise work group (PGGTIE) was created in 2016 to answer this exchange need.

[37] Licence: https://www.postgresql.org/about/licence/
[38] Mailing lists: https://www.postgresql.org/list/
[39] Forum: https://forum.postgresql.fr/

13 Appendices

13.1 Migration from Oracle

You must be aware of the elements of the source servers to be able to define the appropriate target servers in terms of technical environment, software stack, storage system, data, criticality of the application and performance monitoring.

13.1.1 Technical environment

Characteristics of the Oracle servers :

Check the type of servers : dedicated, shared or virtualized?

Check the following technical elements :

- Types of processors ;
- Number of processors ;
- Processor clock speeds ;
- Register size ;
- RAM size.

13.1.2 Software stack

Identify the different software that make up the application stack.

The current OS, including which distribution of Linux, and which version?

The Oracle version, to which PostgreSQL version?

What are the source application servers?

The JVM implementation.

The virtualization system.

13.1.3 Database migration

Migration tools.

There are various options: *Ora2Pg*, *ora_migrator*, *orafce*, *code2pg*, ETL or ELT, the development of custom programs, etc.

Process the metadata first before loading the data.

MIGRATION OF ORACLE STRUCTURES to POSTGRESQL: including the structures of tables and views with their appendices, the various procedures, functions, and triggers.

Check for the presence of partitioning and materialized views.

Partitioning is possible via triggers from PostgreSQL 9.1 onwards, and declaratively since version 10. Many performance and feature improvements have been made to declarative partitioning since its initial release and except for very special cases, trigger based partitioning should no longer be used.

Materialized views

Since PostgreSQL 9.3, materialized views are a built-in feature.

The language of SQL procedures

Oracle uses PL/SQL and PostgreSQL uses PL/pgSQL ; they are quite similar but some adaptation is required.

Model for data type mapping

Strings can be replaced by VARCHAR or TEXT.

Oracle 7-byte dates by 8-byte TIMESTAMPTZ (don't use TIMESTAMP[40]).

CLOB strings are replaced by TEXT.

Integer values are replaced in ascending precision by SMALLINT, INTEGER, BIGINT, NUMERIC.

Decimal numeric values by NUMERIC.

BLOB data by BYTEA (be careful with the escape character when using Ora2Pg).

Migration of normal and stored procedures

List all PL/SQL procedures and functions and transform them into PL/pgSQL.

Generation of database migration scripts

It is usually a good idea to separate DDL instructions and the data loading instructions. This is also the approach taken by tools like *ora2pg*. First the database will be initialized with the structure script, then data will be loaded at a later time. This first script (or set of scripts) will include the data structures as well as the procedures and functions that have been re-adapted for PostgreSQL.

Please keep in mind that to speedup the loading part, it might be preferable to create (some) indexes / constraints after the inital load.

Data migration

It can be achieved via a SQL script to be inserted into the new PostgreSQL database.

[40] TIMESTAMP: https://wiki.postgresql.org/wiki/Don't_Do_This#Date.2FTime_storage

It is also possible to load the data directly from the Oracle database to PostgreSQL. Tools like *oracle_fdw* might be useful there.

Application code

Code modification is necessary to integrate the PostgreSQL driver for managing transactions, opening and closing connections, replacing Oracle keywords with PostgreSQL keywords, external joins, pagination, etc. Depending on where the application logic is, this step can represent a significant part of the migration.

13.1.4 Criticality and backups

Consider the concepts of Disaster Recovery, Busines Continuity, RTO, RPO, and replication.

Which technical solution is used to ensure high availability: Oracle RAC, Oracle DATAGUARD, or something else?

There is no Oracle RAC equivalent in PostgreSQL; however, the DATAGUARD function is provided in PostgreSQL by replication.

The DRP (Disaster Recovery Plan) imposes on us a time limit for the recovery of the application. It takes into account the RTO (Recovery Time Objective) and the RPO (Recovery Point Objective). This will also inform, depending on the volume, the technology, the frequency and the type of backup to be implemented.

On Oracle, RMAN manages backups and restorations; there is no database equivalent under PostgreSQL. Several third-party tools exist, however, including *pgBackRest* and *Barman*. Archive logging is well implemented, almost identically, in the two DBMSs.

Physical restores do not have a partial granularity in PostgreSQL (the whole instance is restored).

13.1.5 Monitoring, performance improvement and supervision

There is no global monitoring like Grid Control in Oracle. It is however possible to use Nagios with the plugin *check_postgres*[41]. Other tools are also available (*PGObserver*, *pgwatch2*, *temBoard*, *PoWa*, etc…).

[41] check_postgres: https://bucardo.org/check_postgres/

13.2 Migration from Db2

A migration leads to a review of the DDL from Db2: it is recommended to provide a tool for transforming (scripting) the DDL, taking into account the elements described below. Regarding the DCL, it is preferable not to try to transpose the Db2 grants to PostgreSQL, as the authorisation levels are very different.

The maintenance procedures (backups, history, defragmentation, statistics collection , obsolete files cleaning, etc.) are naturally different and will have to be adapted.

13.2.1 Some DDL differences between PostgreSQL and Db2

- Character strings:
  PostgreSQL: a character string is defined by its number of characters, with the TEXT type being an exception and a convenient way to store most varchar. Db2: the fixed or variable character strings are expressed in bytes with the consequence of a different number of possible characters depending on the encoding of the database: in UTF-8 some characters are coded on several bytes, in iso8859-15 each character is worth one byte. A varchar(5) string can therefore contain fewer than 5 characters in Db2 if the database is in UTF-8 and includes accented characters.

  Take this into account if you have to migrate a Db2 UTF-8 database to a PostgreSQL database.
- Creating a table in a tablespace:

PostgreSQL	Db2
CREATE TABLE… TABLESPACE <name_ts>	CREATE TABLE…IN <name_ts>

Kindly note that the use of tablespaces in PostgreSQL is not as common/relevant as in Db2.

- Creating a table via the LIKE keyword: in PostgreSQL, the use of ( ) is mandatory around the LIKE clause, in Db2 it is not necessary.

PostgreSQL	Db2
CREATE TABLE eqos.statssov (LIKE eqos.stats INCLUDING ALL)	CREATE TABLE eqos.statssov LIKE eqos.stats

- Default value of a table column:

PostgreSQL	Db2
CREATE TABLE…DEFAULT <value>	CREATE TABLE …WITH DEFAULT <value>

- Creating a table: Some keywords are not recognized by PostgreSQL, so remove them from the Db2 DDL before migration. Example: *APPEND*, *WITH RESTRICT ON DROP*, and *LONG IN*, all to do with table partioning, are defined differently in PostgreSQL.
- Automatic increment of a column counter on table creation: Db2 autoincrement (*[GENERATED ALWAYS | GENERATED BY DEFAULT] AS IDENTITY*) is now supported as of PostgreSQL v12. In earlier versions, you have to use sequences.
- The serial type in PostgreSQL makes it possible to create a sequence easily, but you must not put the integer type (already carried by serial) nor `NOT NULL` as these will lead to syntax errors.
- In Db2, the creation of a sequence requires an obligatory `CREATE SEQUENCE`.

Therefore, if the PostgreSQL version is lower than 12, all Db2 autoincrements must be converted. If the version is at least equal to 12, the keywords used must be checked because Db2 allows more options.

- Unlogged table: A Db2 table declared as `_NOT LOGGED INITIALLY_` must be `_UNLOGGED_` in PostgreSQL.
- Temporary tables: The syntax and functionality of temporary tables differ.

Example:

- Db2 : `DECLARE GLOBAL TEMPORARY TABLE tabtemp LIKE eqos.stats NOT LOGGED`
- PostgreSQL : `CREATE GLOBAL TEMPORARY TABLE tabtemp (LIKE eqos.stats)`

Db2 temporary tables with inter-session data sharing are done via a `CREATE GLOBAL TEMPORARY TABLE`, they have no equivalence in PostgreSQL because although syntactically the keywords `LOCAL` and `GLOBAL` are accepted, the behavior of the temporary table remains local in both cases (no sharing of temporary data).

- Timestamp column on row updates:

These columns, frequently used in Db2, do not exist in PostgreSQL.

Example in Db2: a column with name TS_UPDATE is updated automatically as soon as the row is modified by UPDATE/INSERT sql:

```
CREATE TABLE...
TS_UPDATE TIMESTAMP NOT NULL GENERATED ALWAYS FOR EACH ROW ON UPDATE
AS ROW CHANGE TIMESTAMP
```

When inserting or updating a row, Db2 automatically fills in the TS_UPDATE column. These columns are systematically deployed in certain Db2 databases, **when using PostgreSQL, this action must be carried out by the application, so modify your programs accordingly… or do it with triggers.**.

- ISO timestamp formats:

Not only is the precision of timestamp formats different, but the separator between date and time is different too. **Be careful when migrating timestamps from Db2 to PostgreSQL!**

PostgreSQL	Db2
2014-01-31 00:00:00	2014-01-31__-__00.00.00.**000000**

Note that the `NOT NULL DEFAULT CURRENT TIMESTAMP` format tolerated in Db2 is not allowed in PostgreSQL. You must use `CURRENT\_TIMESTAMP` (also recognized by Db2).

- Materialized views:
  PostgreSQL: creation via `CREATE MATERIALIZED VIEW`.
  Db2: creation via `CREATE TABLE` followed by options specific to a materialized view (called MQT in Db2). **Therefore, the DDL from Db2 needs to be corrected**.
- Drop table and integrity constraints: PostgreSQL : a `DROP TABLE... CASCADE` removes the integrity constraints.
  Db2 : a `DROP TABLE` is sufficient to remove the integrity constraints.
- Creating an index, schema: PostgreSQL : an index name must not be preceded by a schema name, otherwise there is a syntax error. Db2 : it is recommended to include the schema name, and the Db2 tool for generating DDL generates this in front of the index name. If it is omitted in Db2, a schema with the same name as the current schema or connection authority will be used.
- Creating an index, options: Some keywords are not recognized by PostgreSQL, and should therefore be removed from the Db2 DDL before migration. For example: cluster, allow reverse scan, pctfree, etc.
- Assigning indexes to a dedicated tablespace:
  PostgreSQL : the assignment is made during table creation for the primary key and unique constraints. You can put an index in a dedicated tablespace when creating the index. Db2 : You can direct all indexes related to a table into a tablespace with the INDEX IN clause, for example:

```
CREATE TABLE.. IN <name_ts_table> INDEX IN <name_ts_index>.
```

Or when creating the index, like in PostgreSQL.

- Page size:
  PostgreSQL: only 8kB pages are allowed once compiled with the default options.
  Db2: works with page sizes of 4, 8, 16 or 32kB. Modify any Db2 DDL which explicitly specifies page sizes.
- Bufferpool:

  The Db2 notion of bufferpool does not exist in PostgreSQL, we do not create these resources so the Db2 DDL must be adapted accordingly by removing all creation instructions and references to bufferpools in tablespaces.
- Tablespaces

Tablespace creation options differ between Db2 and PostgreSQL, so need to be fixed.

13.2.2 DCL

- Roles vs Unix groups:
  PostgreSQL : it is mandatory to use grants given to roles.
  Db2 : grants are given either to roles or directly to Unix groups.
- PUBLIC rights:

  The basic concept of RESTRICTIVE Db2, which eliminates so-called PUBLIC grants, does not exist in PostgreSQL.
- GRANT ALTER TABLE:

  Used in Db2, but it does not exist as such in PostgreSQL. In PostgreSQL, it is necessary to make the account wishing to modify the table belong to a role which is the OWNER of the table.
- TRUNCATE TABLE:
  PostgreSQL : a `GRANT TRUNCATE` must be done to be allowed to truncate.
  Db2 : a `GRANT DELETE` (or control) gives this privilege.
- GRANT CONNECT:

  To be authorized to connect to a database, you must:
  PostgreSQL: `GRANT CONNECT ON DATABASE \<name\_database\>...` Db2: `GRANT CONNECT ON DATABASE` – do not specify the name of the database, the current database is used by default. If we specify the name of the database, we get a syntax error.

  Other syntax differences may also emerge.

- GRANTS without Db2/PostgreSQL equivalence:

  Some Db2 grants, those linked to functions not present or different in PostgreSQL, do not exist in PostgreSQL. This is the case with grant load, grant use of tablespace, grant usage on workflow, etc.

- System catalog

  This information is stored in upper case in Db2, but lower case in PostgreSQL. Take this into account when searching for information, especially in scripts that use the catalog to collect information.

  Example: search for information in tables belonging to a schema with the name IDENTITY

  PostgreSQL: `SELECT * FROM pg_tables WHERE schemaname = 'identity'`
  Db2: `SELECT * FROM syscat.tables WHERE tabschema = 'IDENTITY'`

13.2.3 Other considerations

- Loading utility:
  A Db2 table can be loaded by *import*, *load*, *ingest* ou *db2move*.
  PostgreSQL exclusively uses the *copy* utility which is much less feature rich than the Db2 utilities.
  Partial loading of a table is only possible since V12, with the introduction of a WHERE clause.
- Other Db2/PostgreSQL utilities:
  reorg becomes *vacuum*
  runstats becomes *analyze*
  backup becomes *pg_dump*, *pg_dumpall* or *pg_basebackup*
- Double quote character:

During the migration, beware of the double quote character ": it is the default string delimiter in Db2 when exporting data so it is found in the unloaded (export) file, surrounding each string. By default, when loading to PostgreSQL these double quote characters will be loaded into the table if they are present in the data file. To avoid this, files exported from Db2 should be in DEL (ASCII) mode and imported into PostgreSQL as csv via the WITH CSV delimiter ',' QUOTE '"' parameters of copy.

- Transaction logs:
  Db2: each database has its own logs.
  PostgreSQL: this is not the case, the logs are common to databases within the same instance, meaning that log issues will have an instance wide impact and will not be limited to a single database.

- LOBs:
  Db2 BLOBs can be replaced by columns in BYTEA format.
  Db2 CLOBs can be replaced by columns in TEXT format. Unlike Db2, which allows embedding LOBs in backups, PostgreSQL does not backup LOBs via *pg_dumpall* (pg backs up LOBs via pg_dump). Db2 options linked to LOBs no longer apply in PostgreSQL (logged/not logged, compact/not compact, inline length, etc.).
- Stored procedures: Written in Db2 in SQL/PL, they must be adapted to PostgreSQL (PL/pgSQL).
- Source code (programs): Modify the call methods of the driver and the use of its properties. Change the SQL code which is specific to Db2/PostgreSQL.

13.3 Migration from Informix

13.3.1 Structure

Database schema (tables, indexes, constraints, etc.).

Scripts should allow the creation of different databases in PostgreSQL.

It will be necessary to ensure that all objects are created respecting the PostgreSQL syntax, as well as the recommended standards:

- Tables;
- Views;
- Triggers;
- Constraints;
- Indexes.

The following table describes some differences for the above objects between the Informix and PostgreSQL DBMSs:

Objects / DBMS	Informix	PostgreSQL
Data type	BLOB	BYTEA
Data type	DATETIME	TIMESTAMP
Constraints (foreign keys)	ALTER TABLE table_name ADD CONSTRAINT (FOREIGN KEY (column_name) REFERENCES ref_table CONSTRAINT constraint_name);	ALTER TABLE table_name ADD CONSTRAINT constraint_name FOREIGN KEY REFERENCES ref_table (column_name);
Indexes	CREATE INDEX index_name ON table_name (column_name) USING btree;	CREATE INDEX index_name ON table_name USING btree(column_name);

13.3.2 Plan to follow when migrating databases

The following steps can be followed for a migration:

- In the source (Informix) databases, first remove all stored procedures that are not used by applications, in order to avoid migrating processes to PostgreSQL that will never be executed;
- Create the databases;

- Load the data;
- Create the indexes and constraints. Adding the indexes and constraints after loading the data will result in a faster execution time of the process.

13.3.3 Integrating the Hibernate framework

If an application framework to manage the persistence of relational database objects in not being used, use the *Hibernate* framework. Applications that use the *Hibernate* framework manage the persistence of objects in relational databases. SQL queries are therefore not written in the code, but generated by *Hibernate*.

The applications are then less dependent on a specific DBMS (with any stored procedures as an exception). A migration from Informix to PostgreSQL will then require minimal changes to the source code of these applications.

However, testing is required to ensure there are no regressions..

13.3.4 Risks

Some risks that can be identified:

- Interruption of the use of applications during migration, during the data loading phase from Informix to PostgreSQL;
- A solution to reduce the execution time is to load the static data (such as reference tables) first, in order to load only the data likely to evolve (requests, files, etc.) on the day of migration to production. This solution should be considered if the loading time is too long and if time is saved (this depends on the proportion of data that can be qualified as static data);
- Certain processes, which had been optimized for the Informix IDS DBMS, may become slower.

13.4 Migration from MSSQL

Some opensource tools that can help facilitate the data migration are:

- https://github.com/dalibo/sqlserver2pgsql;
- http://pgloader.io/.

The PostgreSQL wiki also gives a list of such tools on https://wiki.postgresql.org/wiki/Converting_from_other_Databases_to_PostgreSQL#Microsoft_SQL_Server. For the migration of procedures, the code is very different and must be rewritten.

A very different approach was taken by https://babelfishpg.org/. This PostgreSQL fork by AWS speaks the SQL Server wire protocol and TSQL, which means that applications running on SQL Server should be able to use it with little change.

13.5 Some references

Some published examples of organisations deploying and managing PostgreSQL instances at scale:

GitLab https://about.gitlab.com/blog/2020/09/11/gitlab-pg-upgrade Data volume: > 6 TB;

OneSignal https://onesignal.com/blog/lessons-learned-from-5-years-of-scaling-postgresql 75 TB across 40 servers;

Discourse https://blog.discourse.org/2021/04/standing-on-the-shoulders-of-a-giant-elephant Upgrading to v13;

Meteo France http://www.postgresql.fr/temoignages:meteo_france (in French) France weather forecast. Data volume: 3.5 TB;

Le Bon Coin https://medium.com/leboncoin-engineering-blog/managing-postgresql-backup-and-replication-for-very-large-databases-61fb36e815a0 One of France most visited websites

13.6 Extensions and plugins for PostgreSQL

The table below shows some plugins mentioned in this document:

name	function
auto_explain	Explain plans in the logs
PostGIS	Spatial and geograghic
PGAudit	Audit
pg_repack	Table reorganization (bloat removal)
pg_stat_statements	Statistics
pgcrypto	Cryptography

13.7 Third-party tools for PostgreSQL

The table below shows some third-party tools mentioned in the document:

name	function
Barman	Backup / restore
check_postgres	Monitoring
code2pg	Migration tool
ora2pg	Migration tool
PAF	High availibility
Patroni	High availibility
PgAdmin	GUI
pgBackRest	Backup / restore
pgBadger	Log analysis
PgBouncer	Connection pool management
pgloader	Data loading
Pgpool-II	Connection pool management
pgtop	Statistics
pgwatch2	Monitoring
pitrery	Backup / restore
repmgr	High availibility
temBoard	Monitoring

14 References

PostgreSQL documentation and help:

- Manual: https://www.postgresql.org/docs/current/index.html
- Wiki: https://wiki.postgresql.org
- Mailing lists: https://www.postgresql.org/list/
- IRC general technical channel: irc://irc.libera.chat/postgresql
- Slack: https://postgres-slack.herokuapp.com/

Documentation in French:

- Manual: https://docs.postgresql.fr/current/
- Forums: https://forums.postgresql.fr/