Atelier Migration FDW

Migrer avec les Foreign Data Wrappers

Contents

1/ Introduction

1.1 OBJECTIF DE L'ATELIER

- Découvrir les *Foreign Data Wrappers*
- Connaître les outils de migration
- Réaliser une migration de bout en bout

1.2 DÉROULÉ DE L'ATELIER

- 3 heures
- Travaux pratiques sur la base **Sakila**
  - Migration des tables et des données
  - Manipulation de l'extension `db_migrator`

1.3 PRÉREQUIS DE L'ATELIER

- Un terminal
- Une VM Rocky Linux 8 ou équivalent
- Compétences Linux et SQL
- Une instance Docker au choix avec la base **Sakila**
  - MySQL (image `mysql:8.2`)
  - Oracle (image `gvenzl/oracle-free:23-slim`)
- Une instance PostgreSQL 16

Les exercices de cet atelier sont accompagnés d'une correction valable pour un système d'exploitation Rocky Linux 8. Pour émuler les bases Oracle ou MySQL, il est nécessaire d'installer la version communautaire de Docker.

Installation et démarrage de Docker

```
sudo dnf config-manager --add-repo
 ↪  https://download.docker.com/linux/centos/docker-ce.repo
sudo dnf -y install docker-ce docker-ce-cli containerd.io
sudo systemctl start docker
sudo usermod -aG docker $(whoami)
```

Se reconnecter avec le compte pour bénéficier des droits sur l'instance Docker.

Création du conteneur de son choix

- MySQL Sakila

```
mkdir mysql
export R=https://github.com/ivanceras/sakila/raw/master/mysql-sakila-db
wget $R/sakila-schema.sql -O mysql/01-mysql-sakila-schema.sql
wget $R/mysql-sakila-insert-data.sql -O mysql/02-mysql-sakila-insert-data.sql

cat <<EOF > mysql.env
MYSQL_DATABASE=sakila
MYSQL_PASSWORD=sakila
MYSQL_ROOT_PASSWORD=root
MYSQL_USER=sakila
EOF

docker pull mysql:8.2
docker run --name mysql --env-file mysql.env \
  --publish 3306:3306 --volume $PWD/mysql:/docker-entrypoint-initdb.d \
  --detach mysql:8.2
```

- Oracle Sakila

Démarrer le conteneur Oracle

```
cat <<EOF > oracle.env
APP_USER_PASSWORD=sakila
APP_USER=sakila
ORACLE_DATABASE=sakila
ORACLE_PASSWORD=manager
EOF

docker pull gvenzl/oracle-free:23-slim
docker run --name oracle --env-file oracle.env \
  --publish 1521:1521 \
  --detach gvenzl/oracle-free:23-slim
```

Alimenter la base Sakila avec le jeu de données

```
mkdir oracle
export R=https://github.com/ivanceras/sakila/raw/master/oracle-sakila-db
wget $R/oracle-sakila-drop-objects.sql -O oracle/oracle-sakila-drop-objects.sql
wget $R/oracle-sakila-schema.sql -O oracle/oracle-sakila-schema.sql
wget $R/oracle-sakila-schema-pl-sql.sql -O oracle/oracle-sakila-schema-pl-sql.sql
wget $R/oracle-sakila-insert-data.sql -O oracle/oracle-sakila-insert-data.sql

source oracle.env
docker exec -i oracle sqlplus -S /nolog <<-EOF
  CONNECT ${APP_USER}/${APP_USER_PASSWORD}@localhost:1521/${ORACLE_DATABASE}
  $(cat oracle/oracle-sakila-drop-objects.sql)
  $(cat oracle/oracle-sakila-schema.sql)
  $(cat oracle/oracle-sakila-schema-pl-sql.sql)
  $(cat oracle/oracle-sakila-insert-data.sql)
  exit
EOF
```

Installation et démarrage de l'instance PostgreSQL

La version communautaire est directement installée sur la VM. La communauté propose un guide de téléchargement et d'installation à l'adresse suivante : https://www.postgresql.org/download/linux/.

```
export R=https://download.postgresql.org/pub/repos/yum/reporpms/EL-8-x86_64
sudo dnf install -y $R/pgdg-redhat-repo-latest.noarch.rpm
sudo dnf -qy module disable postgresql

sudo dnf install -y postgresql16-server
sudo /usr/pgsql-16/bin/postgresql-16-setup initdb
sudo systemctl enable postgresql-16
```

```
sudo systemctl start postgresql-16
sudo -iu postgres createuser --superuser $(whoami)
```

2/ Foreign Data Wrappers

2.1 HISTORIQUE

- Norme **ISO/IEC 9075-9** (révision 2003)
  - SQL/MED = SQL Management of External Data
  - Introduction des *foreign-data wrappers* et *dblink*
  - Système de bases de données fédérées

C'est lors de la révision de l'année 2003 que le standard **ISO/IEC** fut subdivisé en 9 parties issues du standard précédent, chacune d'entre elles ayant pour ambition de couvrir un aspect différent du langage. Parmi elles, la norme ISO/IEC 9075-9, *Management of External Data*, aussi appelée **SQL/MED** est créée de toute pièce.

Ce chapitre de la norme propose les concepts de *datalink* et de *foreign-data wrapper*, ainsi que les différentes syntaxes pour les manipuler. Ces éléments peuvent paraître familiers, il s'agit de la même terminologie qu'emploie PostgreSQL pour répondre à la norme. Ce standard impliquerait que les données d'un système soient dites externes, si elles sont disponibles et gérés par un autre système de base de donnée.

Une telle architecture répondant à ces contraintes permet l'émergence des systèmes de bases de données fédérées, responsables de la gestion d'un ensemble de données autonomes et hétérogènes. Sur le plan théorique, les utilisateurs et les applications ne se connectent plus qu'à un seul point d'accès et seraient capables de consulter et modifier les données éparpillées sur différents moteurs de bases de données.

Implémentations de la norme SQL/MED dans PostgreSQL

- 2009 (8.4) : Ajout de l'infrastructure pour SQL/MED
- 2011 (9.1) : Tables externes en lecture seule, contribution `file_fdw`
- 2013 (9.3) : Tables externes en écriture, contribution `postgres_fdw`
- 2016 (9.5) : Support de l'instruction `IMPORT FOREIGN SCHEMA`
- 2018 (11) : Routage des écritures pour les tables partitionnées
- 2021 (14) : Optimisation des `INSERT` avec l'option `batch_size`, support des exécutions asynchrones
- 2023 (16) : Prise en compte de l'option `batch_size` pour les instructions `COPY`

Plus de détails : https://pgpedia.info/f/foreign-data-wrapper-fdw.html

---

2.2 SYNTAXE DE CRÉATION D'UN SERVEUR

```
CREATE EXTENSION postgres_fdw;

CREATE SERVER localhost FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host '/tmp', dbname 'pagila', port '5432');

CREATE USER MAPPING FOR dalibo SERVER localhost
    OPTIONS (user 'dalibo', password '');
```

Les options sont définies par l'extension et peuvent varier de l'une à l'autre. L'exemple proposé correspond à la création d'un *wrapper* vers une instance PostgreSQL qui écoute sur le socket local afin de lire les données des tables contenues dans une autre base de données de l'instance.

Syntaxe : https://www.postgresql.org/docs/current/sql-createserver.html

---

2.3 SYNTAXE DE CRÉATION D'UNE TABLE EXTERNE

```
CREATE FOREIGN TABLE public.films (
    film_id integer NOT NULL,
    title varchar(255) NOT NULL,
    description text
) SERVER localhost OPTIONS (
    schema_name 'public',
    table_name 'films'
);

ALTER FOREIGN TABLE public.films
    ALTER COLUMN description OPTIONS (
        column_name 'desc'
    );
```

Syntaxe : https://www.postgresql.org/docs/current/sql-createforeigntable.html

2.4 CONTRIBUTIONS DE LA COMMUNAUTÉ

Dans le domaine des systèmes relationnels

- **oracle_fdw** : Laurenz Albe (Cybertec)
- **mysql_fdw** : David Page, Ibrar Ahmed, Jeevan Chalke (EntrepriseDB)
- **tds_fdw** : Geoff Montee, Julio González
- **db2_fdw** : Wolfgang Brandl

Plus de détails : https://wiki.postgresql.org/wiki/Foreign_data_wrappers

3/ Travaux pratiques #1

- Installation des FDW
- Configuration du serveur et de l'authentification
- Import automatique avec `IMPORT FOREIGN SCHEMA`
- Transfert des données avec `INSERT`

3.1 INSTALLATION

- `mysql_fdw` disponible depuis le dépôt RPM classique
- `oracle_fdw` disponible depuis le dépôt RPM « non-free »
    - Dépendance avec Oracle InstantClient

Installation de l'extension mysql_fdw

```
sudo dnf install -y mysql_fdw_16
```

Installation de l'extension `oracle_fdw`

```
export R=https://download.oracle.com/otn_software/linux/instantclient/2112000
sudo dnf install -y $R/oracle-instantclient-basic-21.12.0.0.0-1.el8.x86_64.rpm
sudo dnf install -y $R/oracle-instantclient-sqlplus-21.12.0.0.0-1.el8.x86_64.rpm
sudo dnf install -y $R/oracle-instantclient-devel-21.12.0.0.0-1.el8.x86_64.rpm

export R=https://download.postgresql.org/pub/repos/yum/reporpms/non-free/EL-8-x86_64
sudo dnf install -y $R/pgdg-redhat-nonfree-repo-latest.noarch.rpm
sudo dnf update -y
sudo dnf install -y oracle_fdw_16
```

3.2 CONFIGURATION

- À l'intérieur de la base PostgreSQL cible
- Création de l'extension de son choix
- Création du serveur avec la chaîne de connexion
- Authentification

Créer les composants pour l'instance MySQL Sakila

```
export PGDATABASE=sakila_mysql
createdb --owner $(whoami)
```

```
CREATE EXTENSION mysql_fdw;

CREATE SERVER sakila_mysql FOREIGN DATA WRAPPER mysql_fdw
    OPTIONS (host '127.0.0.1', port '3306');

CREATE USER MAPPING FOR public SERVER sakila_mysql
    OPTIONS (username 'sakila', password 'sakila');
```

Créer les composants pour l'instance Oracle Sakila

```
export PGDATABASE=sakila_oracle
createdb --owner $(whoami)
```

```
CREATE EXTENSION oracle_fdw;

CREATE SERVER sakila_oracle FOREIGN DATA WRAPPER oracle_fdw
    OPTIONS (dbserver '//localhost:1521/sakila');

CREATE USER MAPPING FOR public SERVER sakila_oracle
    OPTIONS (user 'sakila', password 'sakila');
```

3.3 IMPORT AUTOMATIQUE DES TABLES

- Instruction `IMPORT FOREIGN SCHEMA`
  - importe les définitions d'une table d'une instance distante
  - collecte automatique des noms des tables et colonnes
  - correspondance du typage des colonnes entre les deux systèmes
  - prise en charge partielle des contraintes

Import du schéma depuis MySQL

L'extension ne fait la distinction stricte entre une table et une vue depuis l'instance MySQL distante. Pour cette raison, l'instruction `IMPORT FOREIGN SCHEMA` est enrichie de la liste `EXCEPT` pour exclure les noms des vues issues de la base `sakila` distante.

L'import des tables se déroule dans un schéma dédié nommé `mysql`.

```
CREATE SCHEMA fdw;
SET search_path = fdw,public;
IMPORT FOREIGN SCHEMA "sakila"
    EXCEPT ("actor_info", "customer_list", "film_list",
            "nicer_but_slower_film_list", "sales_by_film_category",
            "sales_by_store", "staff_list")
    FROM SERVER sakila_mysql INTO fdw;
```

L'import aboutit avec quelques messages d'erreur :

- Un type ENUM a été rencontré sur la table `film`, une instruction `CREATE TYPE` est proposée ;
- Un type `SET` a été rencontré sur la table `film` et ne peut être transposé avec PostgreSQL, le wrapper prend la décision de ne pas l'importer la table.

```
NOTICE:  error while generating the table definition
HINT:  If you encounter an error, you may need to execute the following first:
       ... CREATE TYPE film_rating_t AS enum('G','PG','PG-13','R','NC-17'); ...
WARNING:  skipping import for relation "film"
DETAIL:  MySQL SET columns are not supported.
IMPORT FOREIGN SCHEMA
```

Pour la table `film`, l'import est effectivement impossible, il est nécessaire de la recréer de toute pièce avec des colonnes de type `text` et des contraintes CHECK adaptées pour les types ENUM et SET pour la table finale.

Voici le résultat de la commande `SHOW CREATE TABLE sakila.film` :

```
Table Create Table film CREATE TABLE `film` (
 `film_id` smallint unsigned NOT NULL AUTO_INCREMENT,
 `title` varchar(255) NOT NULL,
```

```
 `description` text,
 `release_year` year DEFAULT NULL,
 `language_id` tinyint unsigned NOT NULL,
 `original_language_id` tinyint unsigned DEFAULT NULL,
 `rental_duration` tinyint unsigned NOT NULL DEFAULT '3',
 `rental_rate` decimal(4,2) NOT NULL DEFAULT '4.99',
 `length` smallint unsigned DEFAULT NULL,
 `replacement_cost` decimal(5,2) NOT NULL DEFAULT '19.99',
 `rating` enum('G','PG','PG-13','R','NC-17') DEFAULT 'G',
 `special_features` set('Trailers','Commentaries','Deleted Scenes','Behind the
↪  Scenes') DEFAULT NULL,
 `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
↪  CURRENT_TIMESTAMP,
 PRIMARY KEY (`film_id`),
 KEY `idx_title` (`title`),
 KEY `idx_fk_language_id` (`language_id`),
 KEY `idx_fk_original_language_id` (`original_language_id`),
 CONSTRAINT `fk_film_language` FOREIGN KEY (`language_id`)
  REFERENCES `language` (`language_id`) ON DELETE RESTRICT ON UPDATE CASCADE,
 CONSTRAINT `fk_film_language_original` FOREIGN KEY (`original_language_id`)
  REFERENCES `language` (`language_id`) ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1001 DEFAULT CHARSET=utf8mb3
```

L'instruction de création de la table `film` avec la bonne correspondance du typage de colonnes serait la suivante :

```
CREATE FOREIGN TABLE fdw.film (
    film_id smallint NOT NULL,
    title varchar(255),
    description text,
    release_year smallint,
    language_id smallint NOT NULL,
    original_language_id smallint,
    rental_duration smallint NOT NULL,
    rental_rate numeric(4,2) NOT NULL,
    length smallint,
    replacement_cost numeric(5,2) NOT NULL,
    rating text,
    special_features text,
    last_update timestamp without time zone
)
SERVER sakila_mysql
OPTIONS (
    dbname 'sakila',
    table_name 'film'
);
```

Ainsi, les 16 tables externes sont présentes dans le schéma `fdw` :

```
sakila_mysql=# \d
                List of relations
 Schema |     Name      |     Type      |  Owner
--------+---------------+---------------+----------
 fdw    | actor         | foreign table | postgres
 fdw    | address       | foreign table | postgres
 fdw    | category      | foreign table | postgres
 fdw    | city          | foreign table | postgres
 fdw    | country       | foreign table | postgres
 fdw    | customer      | foreign table | postgres
 fdw    | film          | foreign table | postgres
 fdw    | film_actor    | foreign table | postgres
 fdw    | film_category | foreign table | postgres
 fdw    | film_text     | foreign table | postgres
 fdw    | inventory     | foreign table | postgres
 fdw    | language      | foreign table | postgres
 fdw    | payment       | foreign table | postgres
 fdw    | rental        | foreign table | postgres
 fdw    | staff         | foreign table | postgres
 fdw    | store         | foreign table | postgres
(16 rows)
```

Import du schéma depuis Oracle

Les vues doivent être exclues de l'import avec l'option `EXCEPT` de l'instruction `IMPORT FOREIGN SCHEMA`.

```
CREATE SCHEMA fdw;
SET search_path = fdw,public;
IMPORT FOREIGN SCHEMA "SAKILA"
    EXCEPT ("actor_info", "customer_list", "film_list",
            "sales_by_film_category", "sales_by_store", "staff_list")
    FROM SERVER sakila_oracle INTO fdw;
```

Les 16 tables sont importées sans erreur dans le schéma `fdw` :

```
sakila_oracle=# \d
                List of relations
 Schema |     Name      |     Type      |  Owner
--------+---------------+---------------+----------
 fdw    | actor         | foreign table | postgres
 fdw    | address       | foreign table | postgres
 fdw    | category      | foreign table | postgres
 fdw    | city          | foreign table | postgres
 fdw    | country       | foreign table | postgres
 fdw    | customer      | foreign table | postgres
```

```
 fdw    | film          | foreign table | postgres
 fdw    | film_actor    | foreign table | postgres
 fdw    | film_category | foreign table | postgres
 fdw    | film_text     | foreign table | postgres
 fdw    | inventory     | foreign table | postgres
 fdw    | language      | foreign table | postgres
 fdw    | payment       | foreign table | postgres
 fdw    | rental        | foreign table | postgres
 fdw    | staff         | foreign table | postgres
 fdw    | store         | foreign table | postgres
(16 rows)
```

3.4 TRANSFERT DES DONNÉES

- Transfert sans transformation
- Transformation à la volée
  - Intervertir la position des colonnes
  - Ignorer une ou plusieurs colonnes
  - Changer l'encodage d'un texte
  - Appliquer une *time zone*
  - TP : colonne `film.special_features`

Transfert sans transformation

Création des tables permanentes

```
CREATE TABLE public.actor (LIKE fdw.actor);
CREATE TABLE public.address (LIKE fdw.address);
CREATE TABLE public.category (LIKE fdw.category);
CREATE TABLE public.city (LIKE fdw.city);
CREATE TABLE public.country (LIKE fdw.country);
CREATE TABLE public.customer (LIKE fdw.customer);
CREATE TABLE public.film(LIKE fdw.film);
CREATE TABLE public.film_actor (LIKE fdw.film_actor);
CREATE TABLE public.film_category (LIKE fdw.film_category);
CREATE TABLE public.film_text (LIKE fdw.film_text);
CREATE TABLE public.inventory (LIKE fdw.inventory);
CREATE TABLE public.language (LIKE fdw.language);
CREATE TABLE public.payment (LIKE fdw.payment);
CREATE TABLE public.rental (LIKE fdw.rental);
CREATE TABLE public.staff (LIKE fdw.staff);
CREATE TABLE public.store (LIKE fdw.store);
```

Insertion des lignes

```
INSERT INTO public.actor SELECT * FROM fdw.actor;
INSERT INTO public.address SELECT * FROM fdw.address;
INSERT INTO public.category SELECT * FROM fdw.category;
INSERT INTO public.city SELECT * FROM fdw.city;
INSERT INTO public.country SELECT * FROM fdw.country;
INSERT INTO public.customer SELECT * FROM fdw.customer;
INSERT INTO public.film SELECT * FROM fdw.film;
INSERT INTO public.film_actor SELECT * FROM fdw.film_actor;
INSERT INTO public.film_category SELECT * FROM fdw.film_category;
INSERT INTO public.film_text SELECT * FROM fdw.film_text;
INSERT INTO public.inventory SELECT * FROM fdw.inventory;
INSERT INTO public.language SELECT * FROM fdw.language;
```

```
INSERT INTO public.payment SELECT * FROM fdw.payment;
INSERT INTO public.rental SELECT * FROM fdw.rental;
INSERT INTO public.staff SELECT * FROM fdw.staff;
INSERT INTO public.store SELECT * FROM fdw.store;
```

Transformation des données à la volée

Cas particulier pour la colonne `film.special_features`

- Le type SET est similaire à un type ARRAY pour PostgreSQL
- Une contrainte d'intégrité permettrait de respecter le nombre de valeurs possibles dans le tableau

Une première proposition de portage du type `text` vers `text[]` consiste à changer la colonne avec une fonction de conversion des données insérées :

```
ALTER TABLE public.film
    ALTER COLUMN special_features TYPE text[]
       USING string_to_array(special_features, ',');
```

Dans le cas où la transformation de la table est trop coûteuse avec un nombre de lignes trop important, il est possible de partir d'une table `public.film` vide avec la bonne définition de la colonne `special_features` puis d'importer les données avec une transformation à la volée :

```
TRUNCATE public.film;
INSERT INTO public.film
    SELECT film_id, title, description, release_year,
           language_id, original_language_id, rental_duration,
           rental_rate, length, replacement_cost, rating,
           string_to_array(special_features, ','), last_update
      FROM fdw.film;
```

Le portage de la colonne `special_features` s'accompagne d'une contrainte CHECK pour respecter le besoin initial que proposait le type SET.

```
ALTER TABLE public.film ADD CHECK (
  special_features <@ ARRAY['Behind the Scenes', 'Commentaries', 'Deleted Scenes',
↪  'Trailers']
);
```

3.5 BILAN

- Les types des colonnes peuvent ne pas être pertinents
  - … voire complétement ignorés lors d'un import automatique

---

- Le transfert des données repose sur une instruction `INSERT ... SELECT`
  - Copie à l'identique
  - Transformation de données à la volée
  - Jointure complexe pour consolider plusieurs tables en une seule
  - … ou éclater une table en plusieurs

- Uniquement les tables et leurs colonnes sont importées
  - Pas d'index
  - Pas de définition des vues
  - Pas de contrainte primaire, unique ou étrangère
  - Pas de procédures stockées ni de fonctions

4/ Les outils de migration

Dans le domaine des migrations, il est courant de choisir un outil polyvalent pour automatiser les grandes étapes du projet, tel que la conversion du modèle de données ou le transfert optimisée des lignes.

Les outils ci-après ont été sélectionnés pour leur licence libre, leur réputation ainsi que leur prise en compte de la technologie des *Foreign Data Wrappers*.

4.1 ORA2PG

- Génère les instructions DDL pour tous les schémas
  - Tables, partitions et colonnes compatibles avec PostgreSQL
  - Contraintes et index
  - Vues, vues matérialisées, fonctions
- Exporte les données à travers les *Foreign Data Wrappers*
  - Disponible depuis la version 22.0 (août 2021)
  - Directive `FDW_SERVER` à renseigner
  - Gain observé de 30 à 40% sur le débit de transfert

Ora2Pg est l'un des outils de migration les plus avancés en matière de migration vers PostgreSQL. À l'origine, il a été conçu pour accompagner les équipes dans le portage d'Oracle vers PostgreSQL, puis il s'est enrichi avec le support de MySQL (version 16.0, octobre 2015) et le support de SQL Server (version 24.0, juillet 2023).

La version 22.0 s'est doté d'un nouveau mode de transfert avec la directive `FDW_SERVER`. Dès lors que cette dernière est valorisée, Ora2Pg crée automatiquement les tables externes en respectant la configuration de l'utilisateur et déclenche le transfert des lignes avec des instructions `INSERT ... SELECT`. Plusieurs tables peuvent être exportées en parallèle, à l'image des autres modes de transferts historiques.

Notes de sortie : https://github.com/darold/ora2pg/releases/tag/v22.0

L'auteur, dans un article, a réalisé un *benckmark* entre les deux modes de transfert et annonce des gains significatifs sur les temps de copie des données.

Source : https://www.migops.com/blog/ora2pg-now-supports-oracle_fdw-to-increase-the-data-migration-speed/

4.2 DB_MIGRATOR

- Collections d'extensions entièrement en PL/pgSQL
  - `ora_migrator` (et `oracle_fdw`)
  - `mysql_migrator` (et `mysql_fdw`)
  - `mssql_migrator` (et `tds_fdw`)
- Exporte la définition des objets d'un schéma dans un catalogue
- Automatise les grandes étapes de migration
  - Transforme les tables externes en tables permanentes
  - Recrée les contraintes et les index
  - Reporte la définition des vues et fonctions (… sans les convertir)

L'outil `db_migrator` est une proposition de Laurenz Albe, le principal contributeur de l'extension `oracle_fdw`. L'outil se veut être un *framework* bas niveau pour simplifier les migrations. L'architecture logicielle permet l'ajout de *plugins* pour supporter facilement de nouveaux systèmes de bases de données comme point de départ.

Les *plugins* `mysql_migrator` et `mssql_migrator` ont été conçues dans un cadre de recherches et ont permis de faire évoluer `db_migrator` significativement, notamment avec l'ajout du partitionnement et la sortie de la version 1.0 en février 2023.

Notes de sortie : https://github.com/cybertec-postgresql/db_migrator/blob/master/CHANGELOG.md

5/ Travaux pratiques #2

- Installation de `db_migrator` et de ses *plugins*
- Récupération et configuration du catalogue (*snapshot*)
- Transfert des données avec `db_migrate_tables`
- Création des vues, contraintes et index

5.1 INSTALLATION

- Pas (encore) de paquets
- Téléchargement de la version en développement
- Déploiement avec `CREATE EXTENSION`

Téléchargement

Les extensions ne sont pas encore empaquetées, il est nécessaire de télécharger leur source pour les déployer au niveau de l'installation de l'instance.

```
export P=https://codeload.github.com
export R=zip/refs/heads

wget $P/cybertec-postgresql/db_migrator/$R/master -O db_migrator.zip
wget $P/cybertec-postgresql/ora_migrator/$R/master -O ora_migrator.zip
wget $P/fljdin/mysql_migrator/$R/main -O mysql_migrator.zip

unzip -o \*.zip
```

Déploiement

Puisque les extensions n'ont pas besoin d'être compilées, une simple installation grâce aux copies des fichiers `control` et `sql` est suffisante. *Dans le cas d'un déploiement plus complexe, il aurait été nécessaire d'installer les en-têtes de développement de PostgreSQL ainsi que le composant `make`.*

```
export PATH=/usr/pgsql-16/bin:$PATH
export EXT=$(pg_config --sharedir)/extension

sudo install -v -c -m 644 */{*.control,*--*.sql} $EXT
```

Installation

Enfin, il reste à installer le bon plugin dans la base de données de son choix.

- Pour les travaux sur la base MySQL Sakila

```
export PGDATABASE=sakila_mysql

CREATE EXTENSION mysql_migrator CASCADE;
```

- Pour les travaux sur la base Oracle Sakila

Comme l'indique la documentation du plugin, le compte `SAKILA` qui est employé pour consulter le catalogue doit disposer des droits de lecture sur les tables du dictionnaire. La commande suivante permet de mettre en place les privilèges :

```
source oracle.env
docker exec -i oracle sqlplus -S /nolog <<-EOF
  CONNECT system/${ORACLE_PASSWORD}@localhost:1521/${ORACLE_DATABASE}
  GRANT SELECT ANY DICTIONARY TO SAKILA;
  exit
EOF

export PGDATABASE=sakila_oracle

CREATE EXTENSION ora_migrator CASCADE;
```

5.2 RÉCUPÉRATION DU CATALOGUE DISTANT

- Méthode `db_migrate_prepare`
  - `plugin` : nom du plugin à utiliser
  - `server` : le nom du serveur à créer au préalable
  - `staging_schema` : emplacement des tables externes connectées au catalogue distant
  - `pgstage_schema` : emplacement du catalogue normalisé (ou *snapshot*)
  - `only_schemas` : liste des schémas distants à exporter
  - `options` (optionnel) : propre à chaque plugin

La méthode `db_migrate_prepare` s'appuie sur les règles définies par le plugin pour créer de nouvelles tables externes dans un schéma temporaire afin de consulter les vues et tables du catalogue distant.

Les résultats obtenus sont normalisés pour être stockés dans une série de tables à l'intérieur d'un autre schéma, appelé `pgstage` ou `pgsql_stage`. Le paramètre `only_schemas` permet de limiter le nombre de schémas à traiter dans le *snapshot*.

La méthode `db_migrate_refresh` est disponible pour mettre à jour partiellement le *snapshot* déjà présent dans le schéma `pgstage`, pour peu qu'aucune table ou objet n'ait été renommé ou supprimé.

Récupération du catalogue

- depuis MySQL

```
SELECT db_migrate_prepare(
  plugin => 'mysql_migrator',
  server => 'sakila_mysql',
  only_schemas => '{sakila}'
);
```

- depuis Oracle

```
SELECT db_migrate_prepare(
  plugin => 'ora_migrator',
  server => 'sakila_oracle',
  only_schemas => '{SAKILA}'
);
```

Consultation des objets créés

- Pour la base MySQL

```
sakila_mysql=# \det fdw_stage.*
               List of foreign tables
  Schema   |          Table          |    Server
-----------+-------------------------+--------------
 fdw_stage | CHECK_CONSTRAINTS       | sakila_mysql
 fdw_stage | COLUMNS                 | sakila_mysql
 fdw_stage | COLUMN_PRIVILEGES       | sakila_mysql
 fdw_stage | KEY_COLUMN_USAGE        | sakila_mysql
 fdw_stage | PARAMETERS              | sakila_mysql
 fdw_stage | PARTITIONS              | sakila_mysql
 fdw_stage | REFERENTIAL_CONSTRAINTS | sakila_mysql
 fdw_stage | ROUTINES                | sakila_mysql
 fdw_stage | SCHEMATA                | sakila_mysql
 fdw_stage | STATISTICS              | sakila_mysql
 fdw_stage | TABLES                  | sakila_mysql
 fdw_stage | TABLE_CONSTRAINTS       | sakila_mysql
 fdw_stage | TABLE_PRIVILEGES        | sakila_mysql
 fdw_stage | TRIGGERS                | sakila_mysql
 fdw_stage | VIEWS                   | sakila_mysql
 fdw_stage | innodb_index_stats      | sakila_mysql
```

- Pour la base Oracle

```
sakila_oracle=# \det fdw_stage.*
              List of foreign tables
  Schema   |        Table         |    Server
-----------+----------------------+---------------
 fdw_stage | checks               | sakila_oracle
 fdw_stage | column_privs         | sakila_oracle
 fdw_stage | columns              | sakila_oracle
 fdw_stage | foreign_keys         | sakila_oracle
 fdw_stage | func_src             | sakila_oracle
 fdw_stage | index_exp            | sakila_oracle
 fdw_stage | keys                 | sakila_oracle
 fdw_stage | pack_src             | sakila_oracle
 fdw_stage | partition_columns    | sakila_oracle
 fdw_stage | schemas              | sakila_oracle
 fdw_stage | segments             | sakila_oracle
 fdw_stage | sequences            | sakila_oracle
 fdw_stage | subpartition_columns | sakila_oracle
 fdw_stage | table_privs          | sakila_oracle
 fdw_stage | tables               | sakila_oracle
 fdw_stage | trig                 | sakila_oracle
 fdw_stage | views                | sakila_oracle
```

- Le *snapshot* est constitué de plusieurs tables dans le schéma `pgsql_stage`

```
sakila_mysql=# \dt pgsql_stage.*
```

5.3 CRÉATION DES TABLES EXTERNES

- Méthode `db_migrate_mkforeign`
  - Création des schémas et des séquences au préalable
  - Une table externe pour chaque table à migrer
- Correspond à l'instruction `IMPORT FOREIGN SCHEMA`
  - Renommage des noms des relations ou colonnes
  - Ajustement du type des colonnes

Création du schéma et des tables externes

L'ensemble des étapes qui suivent s'appuient sur les tables du *snapshot*. La première consiste à créer les schémas présents dans la table `pgsql_stage.schemas` ainsi que les séquences issues de la table `pgsql_stage.sequences`. Les tables externes sont ensuite créées sur la base des éléments collectés et présents dans les tables `pgsql_stage.tables` et `pgsql_stage.columns`.

Il est possible d'ajuster les noms de schémas, des séquences, de tables ou des colonnes en réaliser une série d'instructions UPDATE si besoin.

- depuis MySQL

```
SELECT db_migrate_mkforeign(
  plugin => 'mysql_migrator',
  server => 'sakila_mysql'
);
```

- depuis Oracle

```
SELECT db_migrate_mkforeign(
  plugin => 'ora_migrator',
  server => 'sakila_oracle'
);
```

Consultation des objets créés

```
sakila_mysql=# \det sakila.*
        List of foreign tables
 Schema |     Table     |    Server
--------+---------------+--------------
 sakila | actor         | sakila_mysql
 sakila | address       | sakila_mysql
 sakila | category      | sakila_mysql
 sakila | city          | sakila_mysql
 sakila | country       | sakila_mysql
```

```
 sakila | customer      | sakila_mysql
 sakila | film          | sakila_mysql
 sakila | film_actor    | sakila_mysql
 sakila | film_category | sakila_mysql
 sakila | film_text     | sakila_mysql
 sakila | inventory     | sakila_mysql
 sakila | language      | sakila_mysql
 sakila | payment       | sakila_mysql
 sakila | rental        | sakila_mysql
 sakila | staff         | sakila_mysql
 sakila | store         | sakila_mysql

sakila_mysql=# \ds sakila.*
               List of relations
 Schema |     Name      |   Type   |  Owner
--------+---------------+----------+----------
 sakila | actor_seq     | sequence | postgres
 sakila | address_seq   | sequence | postgres
 sakila | category_seq  | sequence | postgres
 sakila | city_seq      | sequence | postgres
 sakila | country_seq   | sequence | postgres
 sakila | customer_seq  | sequence | postgres
 sakila | film_seq      | sequence | postgres
 sakila | film_text_seq | sequence | postgres
 sakila | inventory_seq | sequence | postgres
 sakila | language_seq  | sequence | postgres
 sakila | payment_seq   | sequence | postgres
 sakila | rental_seq    | sequence | postgres
 sakila | staff_seq     | sequence | postgres
 sakila | store_seq     | sequence | postgres
```

5.4 TRANSFERT DES DONNÉES

- Méthode `db_migrate_tables`
  - Réalise une boucle sur les tables à migrer
  - Matérialise les tables les unes après les autres
- Méthode `materialize_foreign_table`
  - Créer la table (et ses partitions si requises)
  - Réalise le transfert avec des instructions `INSERT`
  - Détruit la table externe

Migration automatique des données

La première méthode `db_migrate_tables` se charge de récupérer toutes les tables à migrer, celles dont le champ `migrate` est actif dans le *snapshot* (`pgsql_stage.tables`). Ensuite, chaque table est traitée une à une au sein d'une transaction qui lui est propre à l'aide d'une autre méthode nommée `materialize_foreign_table` :

1) Renommage de la table externe pour ne pas entrer en conflit
2) Création de la table permanente avec son nom final
3) Création des partitions de la table si nécessaire
4) Transfert des données si l'option `with_data` est active
5) Suppression de la table externe

En cas d'anomalie dans l'un de ces étapes, c'est l'entièreté des opérations qui sont annulées à l'aide d'un ROLLBACK.

- depuis MySQL

```
SELECT db_migrate_tables(
  plugin => 'mysql_migrator',
  with_data => true
);
```

- depuis Oracle

```
SELECT db_migrate_tables(
  plugin => 'ora_migrator',
  with_data => true
);
```

Migration fine des données

Dans certaines situations, il peut être nécessaire de filtrer certaines lignes lors de la migration, par exemple pour anticiper la copie des lignes mortes et les exclure lors de l'opération de bascule.

L'exemple précédant le démontrait : la méthode `materialize_foreign_table` ne permet pas de filtrer les lignes lors de l'instruction `INSERT`. Il est alors nécessaire d'utiliser une autre méthode de bas niveau pour construire la table externe à traiter dans un schéma à part, par exemple `sakila_fdw`.

```
CREATE SCHEMA sakila_fdw;

-- pour la base MySQL
SELECT regexp_replace(statement,
        'TABLE ([A-Za-z0-9_]+)\.([A-Za-z0-9_]+)',
        'TABLE \1_fdw.\2')
  FROM construct_foreign_tables_statements(
    plugin => 'mysql_migrator',
    server => 'sakila_mysql')
 WHERE schema_name = 'sakila' \gexec

-- pour la base Oracle
SELECT regexp_replace(statement,
        'TABLE ([A-Za-z0-9_]+)\.([A-Za-z0-9_]+)',
        'TABLE \1_fdw.\2')
  FROM construct_foreign_tables_statements(
    plugin => 'ora_migrator',
    server => 'sakila_oracle')
 WHERE schema_name = 'sakila' \gexec
```

Il est alors possible de rédiger ses propres instructions `INSERT`, par exemple pour la table `rental` que nous vidons au préalable :

```
TRUNCATE TABLE sakila.rental;
INSERT INTO sakila.rental SELECT * FROM sakila_fdw.rental WHERE rental_date <
 ↪  '2005-08-01';
INSERT INTO sakila.rental SELECT * FROM sakila_fdw.rental WHERE rental_date BETWEEN
 ↪  '2005-08-01' AND '2006-01-01';
```

Cette méthode permet également de contrôler finement les étapes, et d'ajouter un contrôle intermédiaire avant la suppression des tables externes :

```
CREATE TABLE public.migration_report (
  schema_name   name,
  table_name    name,
  total_rows    bigint,
  imported_rows bigint,
  completed     char(1)
    GENERATED ALWAYS AS (
```

```
      CASE WHEN total_rows = imported_rows
           THEN '☑' ELSE '☒'
      END
    ) STORED,
  PRIMARY KEY (schema_name, table_name)
);

DO $$
DECLARE
  rec      record;
  total    bigint;
  imported bigint;
BEGIN
  FOR rec IN
    SELECT schema, table_name
      FROM pgsql_stage.tables WHERE migrate
  LOOP
    EXECUTE format('SELECT count(*) FROM %s_fdw.%s', rec.schema, rec.table_name)
     ↪  INTO total;
    EXECUTE format('SELECT count(*) FROM %s.%s', rec.schema, rec.table_name) INTO
     ↪  imported;
    INSERT INTO public.migration_report
      VALUES (rec.schema, rec.table_name, total, imported);
  END LOOP;
END; $$;
```

Résultat de la copie des données :

```
SELECT * FROM public.migration_report
ORDER BY schema_name, table_name;
```

```
 schema_name |  table_name   | total_rows | imported_rows | completed
-------------+---------------+------------+---------------+-----------
 sakila      | actor         |        200 |           200 | ☑
 sakila      | address       |        603 |           603 | ☑
 sakila      | category      |         16 |            16 | ☑
 sakila      | city          |        600 |           600 | ☑
 sakila      | country       |        109 |           109 | ☑
 sakila      | customer      |        599 |           599 | ☑
 sakila      | film          |       1000 |          1000 | ☑
 sakila      | film_actor    |       5462 |          5462 | ☑
 sakila      | film_category |       1000 |          1000 | ☑
 sakila      | film_text     |       1000 |          1000 | ☑
 sakila      | inventory     |       4581 |          4581 | ☑
 sakila      | language      |          6 |             6 | ☑
 sakila      | payment       |      16049 |         16049 | ☑
 sakila      | rental        |      16044 |         15862 | ☒
```

```
 sakila      | staff         |          2 |            2 | ☒
 sakila      | store         |          2 |            2 | ☒
```

La table `rental` requiert une attention particulière ; en effet, les données supérieures à 2016 ont été volontairement ignorées. La commande suivante permet de rattraper les lignes manquantes et de terminer la migration.

```
INSERT INTO sakila.rental SELECT * FROM sakila_fdw.rental WHERE rental_date >
 ↪  '2006-01-01';
```

À l'issue de la migration, le schéma temporaire des tables externes peut être supprimé.

```
DROP SCHEMA sakila_fdw CASCADE;
```

5.5 CRÉATION DES OBJETS COMPLEXES

- Fonctions et procédures avec `db_migrate_functions`
- Triggers avec `db_migrate_triggers`
- Vues avec `db_migrate_views`
- **Pas de conversion automatique**

Cette étape permet de récupérer la définition des objets complexes depuis l'instance distante, mais **ne procède pas** à leur conversion. Il n'est pas assuré que ces objets soient compatibles avec PostgreSQL selon la manière dont ils ont été écrits.

Le plugin `mysql_migrator` ne permet pas de récupérer le corps des fonctions et des procédures.
L'export des triggers avec `db_migrator` est actuellement en erreur et doit faire l'objet d'un patch correctif. Pour une cause similaire, l'export des vues avec `ora_migrator` est en erreur.

Exemple d'une vue avec MySQL

Pour commencer, il est nécessaire de désactiver la migration de toutes les vues afin que la méthode `db_migrate_views` puisse les ignorer. Leurs créations nécessitent une revue manuelle avec des ajustements pour rendre les requêtes SQL compatibles avec PostgreSQL.

```
UPDATE pgsql_stage.views SET migrate = false;
```

L'obtention de la définition originale se réalise en consultant la table `pgsql_stage.views`. Ici, nous concentrons nos efforts sur une seule vue, nommée « customer_list » :

```
SELECT orig_def FROM pgsql_stage.views
 WHERE view_name = 'customer_list' \g (tuples_only)
```

```
select `cu`.`customer_id` AS `ID`,concat(`cu`.`first_name`,' ',`cu`.`last_name`)
AS `name`,`a`.`address` AS `address`,`a`.`postal_code` AS `zip code`,`a`.`phone`
AS `phone`,`sakila`.`city`.`city` AS `city`,`sakila`.`country`.`country` AS
`country`,if(`cu`.`active`,'active','') AS `notes`,`cu`.`store_id` AS `SID` from
(((`sakila`.`customer` `cu` join `sakila`.`address` `a` on((`cu`.`address_id` =
`a`.`address_id`))) join `sakila`.`city` on((`a`.`city_id` =
`sakila`.`city`.`city_id`))) join `sakila`.`country`
on((`sakila`.`city`.`country_id` = `sakila`.`country`.`country_id`)))
```

La définition provient directement du catalogue MySQL, tel que réécrit par le moteur lui-même. Pour rendre cette requête compatible avec PostgreSQL, il est nécessaire de la réécrire avec les quelques astuces suivantes :

- Les guillements inversés (*backquotes*) peuvent être ignorés ;
- L'alias sur la colonne `postal_code` doit être échappé par des guillemets puisqu'il contient un caractère d'espacement ;
- La méthode `IF` doit être remplacée par une instruction CASE.

```
UPDATE pgsql_stage.views SET migrate = true,
  definition = $$
 select cu.customer_id AS ID,
    concat(cu.first_name,' ',cu.last_name) AS name,
    a.address AS address,
    a.postal_code AS "zip code",
    a.phone AS phone,
    sakila.city.city AS city,
    sakila.country.country AS country,
    (CASE WHEN cu.active > 0 THEN 'active' ELSE '' END) AS notes,
    cu.store_id AS SID
   from (((sakila.customer cu
      join sakila.address a on((cu.address_id = a.address_id)))
      join sakila.city on((a.city_id = sakila.city.city_id)))
      join sakila.country on((sakila.city.country_id = sakila.country.country_id)))
  $$
WHERE view_name = 'customer_list';
```

Enfin, la méthode `db_migrate_views` peut être invoquée.

```
SELECT db_migrate_views(plugin => 'mysql_migrator');
```

5.6 CRÉATION DES INDEX ET DES CONTRAINTES

- Méthode `db_migrate_indexes`
  - `construct_indexes_statements`
- Méthode `db_migrate_constraints`
  - `construct_key_constraints_statements`
  - `construct_fkey_constraints_statements`
  - `construct_check_constraints_statements`
  - `construct_defaults_statements`

La dernière étape d'une migration consiste à recréer les contraintes et les index. Ces deux opérations nécessitent que les données soient importées au préalable pour économiser des opérations disques.

L'extension `db_migrator` propose deux méthodes distinctes pour les créer. Il est recommandé de créer les index avant les contraintes, pour optimiser les contrôles d'intégrité, notamment sur les colonnes externes lors de l'ajout de contraintes étrangères.

Création des index

- depuis MySQL

```
SELECT db_migrate_indexes(plugin => 'mysql_migrator');
```

- depuis Oracle

```
SELECT db_migrate_indexes(plugin => 'ora_migrator');
```

Création des contraintes

La méthode `db_migrate_constraints` est responsable de recréer les contraintes issues du modèle distant, à savoir : clés primaires, clés étrangères, contraintes d'unicité et contraintes de type CHECK. Elle embarque également une dernière étape qui correspond à l'ajout des valeurs par défaut des colonnes de tables.

- depuis MySQL

```
SELECT db_migrate_constraints(plugin => 'mysql_migrator');
```

- depuis Oracle

```
SELECT db_migrate_constraints(plugin => 'ora_migrator');
```

Exécution parallélisée

Dans la vie réelle, l'étape finale de création des index et des contraintes sur des données volumineuses ne peut être réalisée avec cette approche séquentielle. Les instructions `CREATE INDEX` et `ALTER TABLE` doivent pouvoir être exécutées en parallèle, à l'aide de plusieurs processus.

Les méthodes de bas-niveau fournies par `db_migrator` permettent d'obtenir ces instructions. Grâce à elles, il devient possible d'utiliser des outils tiers pour ouvrir plusieurs processus et y répartir les opérations à travers plusieurs connexions.

- `construct_indexes_statements`
- `construct_key_constraints_statements`
- `construct_fkey_constraints_statements`
- `construct_check_constraints_statements`
- `construct_defaults_statements`

La première étape consiste à exporter les instructions dans des fichiers distincts. Ici, nous séparons les instructions dans trois fichiers (index et clés uniques, contraintes étrangères et valeurs par défaut des colonnes de tables).

```
-- changer la variable pour utiliser le bon plugin
\set plugin ora_migrator

-- l'export se réalise avec la méta-commande \g et ses options
-- les clés primaires et contraintes d'unicité sont similaires à la création d'index
SELECT statement FROM construct_indexes_statements(plugin => :'plugin')
  UNION
    SELECT statement FROM construct_key_constraints_statements(plugin => :'plugin')
  \g (format=unaligned tuples_only) indexes.sql

-- le reste des contraintes est exporté dans un même fichier
-- chaque instruction doit être exportée sans saut de ligne avec REPLACE
SELECT statement FROM construct_fkey_constraints_statements(plugin => :'plugin')
  UNION
    SELECT replace(statement, E'\n', '')
      FROM construct_check_constraints_statements(plugin => :'plugin')
  \g (format=unaligned tuples_only) constraints.sql

-- ajout d'un point-virgule à chaque instruction pour que le fichier
-- soit correctement traité par l'option -f de psql
SELECT CONCAT(statement, ';') FROM construct_defaults_statements(plugin => :'plugin')
  \g (format=unaligned tuples_only) defaults.sql
```

Dans le cas où les index et contraintes sont déjà présents dans le schéma `sakila` de la base cible, il est possible de substituer les instructions de création par des instructions de suppression pour nettoyer le schéma.

```
drop_index='s/.* INDEX ([^ ]+).*/DROP INDEX \1;/'
drop_constraint='s/(.*) ADD CONSTRAINT ([^ ]+).*/\1 DROP CONSTRAINT \2 CASCADE;/'

export PGOPTIONS="-c search_path=sakila,public"
perl -pe "$drop_index,$drop_constraint" indexes.sql constraints.sql | psql
```

La commande `xargs` permet ensuite de récupérer les instructions lignes à lignes depuis les fichiers et de les faire exécuter par le premier processus disponible. Par exemple, pour répartir les instructions de création d'index sur 4 processus, la commande `xargs` prend la forme suivante :

```
xargs -P 4 -a indexes.sql -d '\n' -I % sh -c 'psql -c "%"'
```

De la même manière, il devient aisé de créer les contraintes en parallèle :

```
xargs -P 4 -a constraints.sql -d '\n' -I % sh -c 'psql -c "%"'
```

La remise en place des valeurs par défaut des colonnes de tables n'est pas coûteuse et les instructions du fichier peuvent être exécutées dans la même connexion :

```
psql -f defaults.sql
```

5.7 FINALISER LA MIGRATION

- Méthode `db_migrate_finish`
  - Suppression des schémas internes
- Retrait des extensions

Dès que les contrôles de fin de migration sont positifs, l'ensemble des objets créés par l'extension `db_migrator` peuvent être supprimés à l'aide de la méthode `db_migrate_finish`.

```
SELECT db_migrate_finish();
```

Enfin, les extensions peuvent être supprimées de la base de données.

```
DROP EXTENSION db_migrator CASCADE;
DROP EXTENSION mysql_fdw;
DROP EXTENSION oracle_fdw;
```

5.8 BILAN

- Les outils de migration sont bien plus complets
  - **Ora2Pg** est un client en Perl, clé en main
  - **db_migrator** est un *framework* en PL/pgSQL

- **db_migrator** n'a pas encore trouvé son public
  - Contribuez !
  - Ouvrez des issues sur GitHub !
  - Créez des *plugins* !

URL des projets :

- db_migrator https://github.com/cybertec-postgresql/db_migrator/issues
- ora_migrator https://github.com/cybertec-postgresql/ora_migrator/issues
- mysql_migrator https://github.com/fljdin/mysql_migrator/issues
- mssql_migrator https://github.com/fljdin/mssql_migrator/issues

Idées de *plugins* à créer :

- Sybase ASE avec l'extension **tds_fdw**
- DB2 avec l'extension **db2_fdw**

6/ Pour aller plus loin

- Conférences de migrations aux PG Sessions et PG Day France
- Articles de blog sur https://fljd.in
- Formation MIGORPG dispensée par Dalibo
- Guide de portage Oracle vers PostgreSQL

Conférences des années passées

- REX sur une migration d'Oracle à PostgreSQL[1] - par Cédric Champmartin, Université de Lorraine (2023)
- Migration vers PostgreSQL : mener de gros volumes de données à bon port[2] - par Philippe Beaudoin, DALIBO (2022)
- La validation de migration facilitée avec Ora2Pg[3] - par Gilles Darold, MigOps Inc (2021)
- Migrez vos bases de données vers PostgreSQL et retrouvez une vraie liberté[4] - Fabrice Viault, Cheops technologies (2019)
- Démarche de transition à PostgreSQL et outillage open source[5] - Anthony Nowocien, Société générale (2019)

Série d'articles sur les techniques de migration

- Parlons un peu des données externes[6] (juillet 2021)
- Migrer vers PostgreSQL[7] (décembre 2021)
- En route vers la liberté avec db_migrator[8] (août 2023)
- Les modes de transfert dans une migration[9] (octobre 2023)

Contenus pédagogiques maintenus par Dalibo

- Formation MIGORPG : Migrer d'Oracle à PostgreSQL[10]

[1] https://www.youtube.com/watch?v=XzWCnuNX3bs
[2] https://www.youtube.com/watch?v=CR67iLHTocY
[3] https://www.youtube.com/watch?v=OY3p7uhriZ8
[4] https://www.youtube.com/watch?v=Fx-Vs1M9AgQ
[5] https://www.youtube.com/watch?v=P8F1Ie69XV8
[6] https://fljd.in/2021/07/16/parlons-un-peu-des-donnees-externes/
[7] https://fljd.in/2021/12/06/migrer-vers-postgresql/
[8] https://fljd.in/2023/07/28/en-route-vers-la-liberte-avec-db_migrator/
[9] https://fljd.in/2023/10/11/les-modes-de-transfert-dans-une-migration/
[10] https://dali.bo/migorpg_html

- Guide de portage Oracle vers PostgreSQL[11]

[11] https://dalibo.github.io/from-oracle-to-postgresql/fr/

7/ Questions ?

Notes

Notes

Notes

Nos autres publications

FORMATIONS

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

LIVRES BLANCS

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.

8/ DALIBO, L'Expertise PostgreSQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.