Atelier Migration PostgreSQL Migrator

Découverte de PostgreSQL Migrator

Contents

PostgreSQL Migrator

Modernize your database by moving to PostgreSQL.

1/ Introduction

1.1 TOUR DE TABLE

- « Avez-vous un ou des projets de migration vers PostgreSQL ? »
- « Depuis quel système ? Oracle, MySQL, SQL Server ? »
- « Quels outils ou techniques avez-vous employés ? »
- « Quelles ont été les principales difficultés ? »

1.2 DÉROULÉ DE L'ATELIER

- 3 heures
- Présentation de PostgreSQL Migrator
- Travaux pratiques sur la base **HR**
  - Manipulation de l'outil
  - Analyse de la complexité
  - Migration des tables et des données

1.3 PRÉREQUIS DE L'ATELIER

- Un terminal
- Une VM Rocky Linux 9 ou équivalent
- Compétences Linux et SQL
- Une instance Oracle (image `gvenzl/oracle-free:23-slim`)
- Une instance PostgreSQL 18

Environnement

L'ensemble des travaux pratiques nécessite une base de données Oracle dans un conteneur. Le modèle de données est le traditionnel schéma HR avec des adaptations.

> Installer `docker-ce` et `docker-compose-plugin` et démarrer le service.

https://docs.docker.com/engine/install/rhel/#installation-methods

```
$ sudo dnf -y update
$ sudo dnf -y install dnf-plugins-core
$ sudo dnf config-manager --add-repo
 ↪  https://download.docker.com/linux/rhel/docker-ce.repo
$ sudo dnf -y install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
$ sudo systemctl enable --now docker
```

> Démarrer les conteneurs Oracle et PostgreSQL

Un fichier `docker-compose.yml` est disponible en téléchargement.

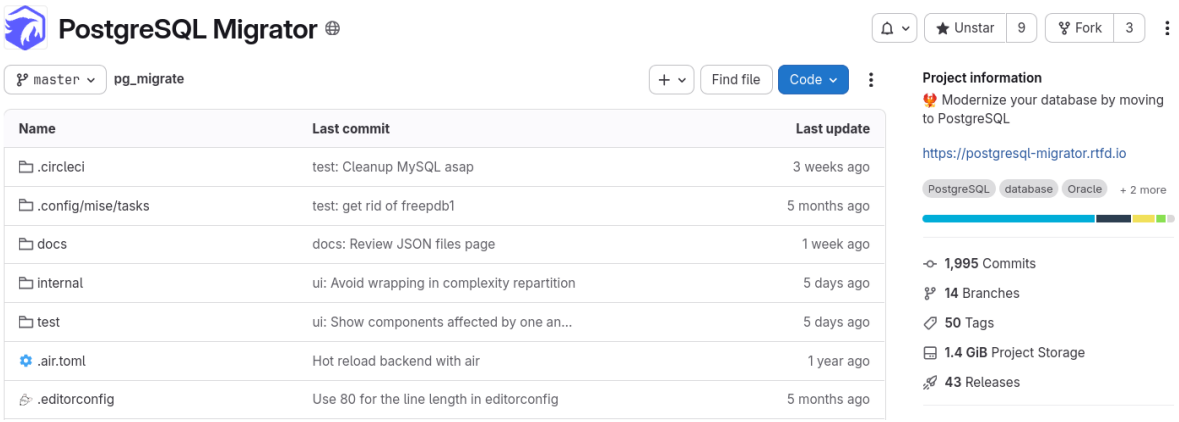
```
$ curl -s -kL https://dali.bo/tp_migpg -o - | tar xvz
$ sudo docker compose up -d oracle postgres
```

2/ Présentation

2.1 AMBITIONS (1/3)

Moderniser la migration avec des logiciels libres.

- Un outil universel
  - Couplé à `transqlate` pour la conversion du code
  - Sous licence PostgreSQL
  - Hébergé sur Gitlab.com
- Langages Go (*backend*) et Vue.js (*frontend*)



2.2 AMBITIONS (2/3)

Proposer une gouvernance solide

- Le projet rejoint le Dalibo Labs
- Travail conjoint entre les DBA et les DEV
  - Itérations de deux semaines
  - Intégration et livraison continues (*CI/CD*)


ÉTIENNE BERSAC
etienne.bersac@dalibo.com


PIERRE GIRAUD
pierre.giraud@dalibo.com


PIERRE-LOUIS GONON
pierre-louis.gonon@dalibo.com


MARION GIUSTI
marion.giusti@dalibo.com


FLORENT JARDIN
florent.jardin@dalibo.com

2.3 AMBITIONS (3/3)

Favoriser drastiquement la prise en main

- Simplification de l'installation
- Navigation graphique dans le modèle de données à convertir
- Identification des points chaud du chantier
- Peu de configurations superflues
- Orchestration des tâches de copie automatique et optimisée
- Documentation

2.4 FONCTIONNALITÉS (1/4)

Un binaire unique en Go

- Aucune dépendance à installer sur le système
- Pilotes communautaires : Oracle, MySQL
- Ligne de commande simple et moderne
- Serveur Web embarqué (Vue.js)

2.5 FONCTIONNALITÉS (2/4)

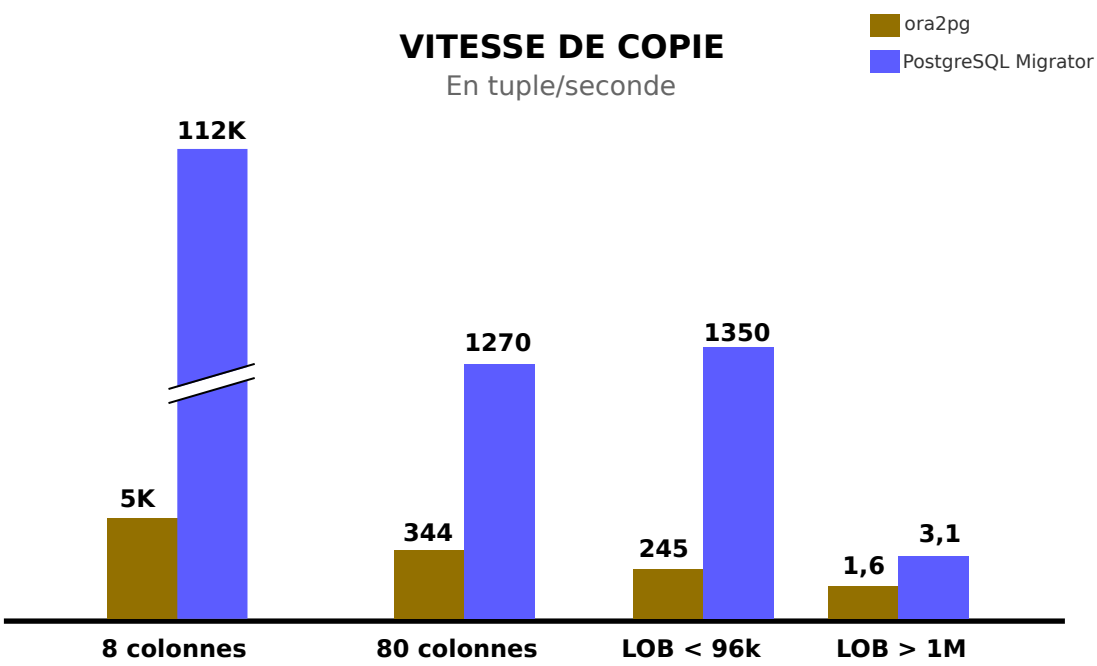
Mode hors-ligne

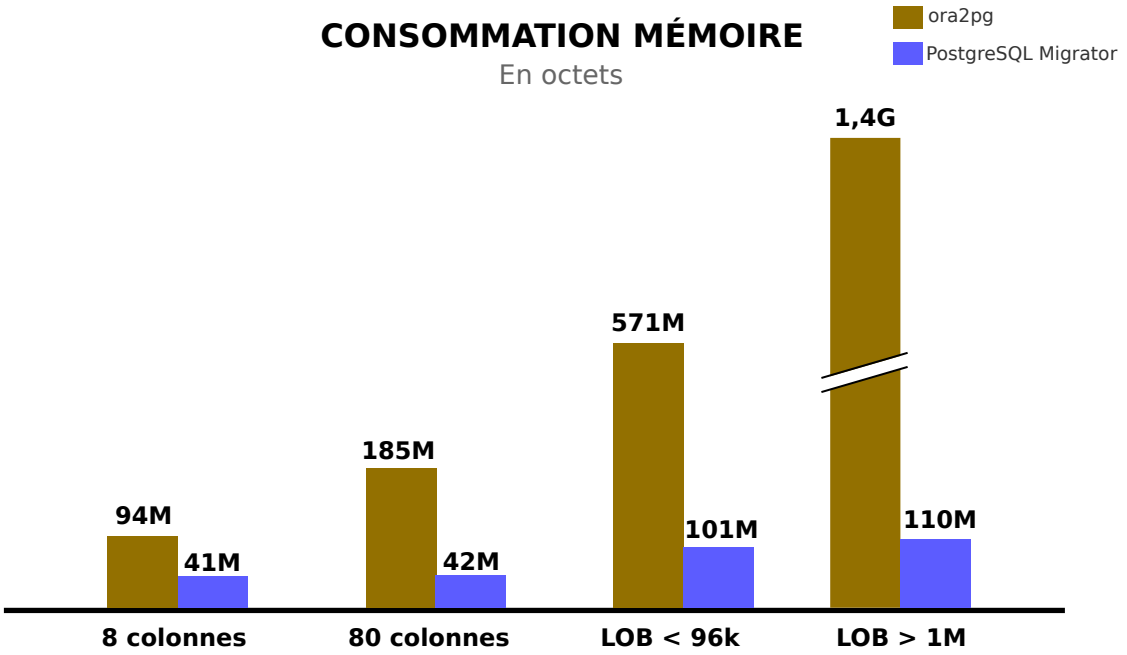
- Inspection de la base source vers des fichiers JSON
- Exploitation des données sans besoin de connexion
  - Analyse des points chauds
  - Conversion du modèle pour PostgreSQL au format SQL
  - Versionnement pour le travail collaboratif

2.6 FONCTIONNALITÉS (3/4)

Transfert des données performant

- Langage compilé
- Techniques de transfert optimisé
  - COPY en flux continu
  - Maîtrise de la consommation mémoire
  - Réutilisation des connexions ouvertes
- Support des BLOB/CLOB




CONSOMMATION MÉMOIRE
En octets
ora2pg
PostgreSQL Migrator
94M
41M
185M
42M
571M
101M
1,4G
110M
8 colonnes
80 colonnes
LOB < 96k
LOB > 1M

2.7 FONCTIONNALITÉS (4/4)

Interface graphique (Web)

- Navigation dans les modèles source et cible
- Définition des objets relationnels et procéduraux
- Recherche globale
- **À terme**
  - Assistant à la conversion
  - Progression du transfert des données
  - Édition / correction du code PL/pgSQL

3/ Exercice #1

- Installation de PostgreSQL Migrator
- Inspection de la base source
- Générer un rapport d’évaluation

3.1 INSTALLATION DE POSTGRESQL MIGRATOR

> Télécharger et installer la dernière version de l'outil

Se rendre sur la page du projet et télécharger le dernier paquet disponible pour Redhat. Les formats `.deb` et `.rpm` sont à privilégier.

- https://gitlab.com/dalibo/pg_migrate/-/releases/permalink/latest

```
$ v=1.0.0-beta.7
$ d=https://gitlab.com/dalibo/pg_migrate/-/releases/v${v}/downloads
$ sudo dnf -y install ${d}/pg-migrate_${v}_linux_amd64.rpm
```

3.2 INSPECTER LE CATALOGUE

> Découvrir les sous-commandes de l'outil.

PostgreSQL Migrator dispose d'une interface en ligne de commande (CLI) nommée `pg_migrate`. Chaque étape du chantier de migration est rattachée à une des sous-commandes de la CLI.

```
$ pg_migrate --help
Usage:  pg_migrate [OPTIONS] COMMAND

Commands:
  convert                 Convert source catalog for PostgreSQL
  dump                    Dumps a source database as text files or to stdin
  init                    Initialize a new migration project
  inspect                 Fetch and analyse source database catalog
  report                  Generate a migration report
  status                  Describe a migration project
  ui                      Interactive audit web interface
```

Devant chaque sous-commande, il est possible de changer le comportement de la CLI avec les options générales comme `--verbose` ou `--plain`.

```
General Options:
  -C, --directory string    Change to directory before doing anything
  -?, --help                Print help and exit
      --offline             Prevent source database access
      --plain               Disable log coloration
      --skip-date-warning   Suppress the warning about outdated build
  -v, --verbose             Show debug log messages
  -V, --version             Print version and exit
```

Enfin, certaines variables d'environnement sont documentées et peuvent être valorisées depuis le contexte d'exécution.

```
Environment variables:

  PGMDIRECTORY      Change to directory before doing anything.
  PGMOFFLINE        Set to true prevent source database access.
```

> Initialiser le répertoire de travail.

Pour chaque projet de migration, il est nécessaire d'initialiser un répertoire de travail à partir de la chaîne de connexion et d'un compte de connexion.

```
$ pg_migrate init --source oracle://hr:phoenix@localhost/hr atelier
09:54:08 INFO   Opening database connection pool. key=source driver=oracle
 ↪  dsn=or:hr@localhost/hr
```

```
09:54:08 INFO   Connecting to database.          key=source count=1
09:54:08 INFO   Initialized migration project.   source="Oracle Database 23ai Free"
 ↪  version=23.0.0.0.0 path=atelier
09:54:08 INFO   Databases connections closed.    source=1 target=0
```

Découvrir les fichiers du projet.

La commande `init` crée une série de fichiers dans le dossier de travail. Le dossier caché `.pg_migrate` est garant de l'état du projet et ne doit pas être modifié à la main.

Le fichier `.env` contient les variables d'environnement du projet, tel que PGMSOURCE qui correspond au paramètre `--target` de la commande précédente. Enfin, le fichier `pg_migrate.toml` permet de configurer le comportement de PostgreSQL Migrator durant toutes les étapes du projet.

```
atelier
├── Annotations.jq
├── .env
├── .gitignore
├── .pg_migrate
│   └── Info.json
├── pg_migrate.toml
├── report.md.tmpl
├── Summary.jq
└── Total.jq
```

Les fichiers `.jq` et `.md.tmpl` sont respectivement les filtres pour manipuler les données JSON du projet, et les *templates* pour la présentation de ces données.

Exécuter la commande d'inspection.

Avant toute chose, il faut se positionner dans le répertoire de travail, là où la commande `pg_migrate` consulte les fichiers du dossier `.pg_migrate` et le contenu du fichier `.env`.

Lancer la sous-commande `inspect`. Si besoin, l'option `--verbose` peut s'intercaler entre la commande `pg_migrate` et la sous-commande.

```
$ cd atelier/
$ pg_migrate inspect
09:55:38 INFO   Opening database connection pool. key=source driver=oracle
 ↪  dsn=or:hr@localhost/hr
09:55:38 INFO   Connecting to database.          key=source count=1
09:55:38 INFO   Inspecting source database.      driver=oracle
09:55:38 INFO   Inspected metadata.              instance=FREE software="Oracle
 ↪  Database 23ai Free" version=23.9.0.25.07
09:55:38 WARN   Failed to query database size.   err="ORA-00942: table or view
 ↪  \"SYS\".\"DBA_SEGMENTS\" does not exist\n error occur at position: 44"
```

```
09:55:38 INFO   Found sequences.                 count=4
09:55:39 INFO   Found procedures.                count=2
09:55:39 INFO   Found functions.                 count=2
09:55:39 INFO   Found packages.                  count=2
09:55:39 INFO   Found views.                     count=1
09:55:41 INFO   Found tables.                    count=8
09:55:41 INFO   Connecting to database.         key=source count=5
09:55:41 INFO   Found table triggers.           count=6
09:55:47 INFO   Found tables keys.              count=9
09:55:49 INFO   Found foreign keys.             count=12
09:55:52 INFO   Found tables checks.            count=2
09:55:56 INFO   Found tables indexes.           count=2
09:55:56 INFO   Inspected schema.               name=HR
```

Plusieurs avertissements s'affichent à l'écran pour nous informer que l'inspection a rencontré une ou plusieurs erreurs à la lecture du catalogue distant. L'inspection est dite partielle, ce qui signifie que des informations sont manquantes pour établir une évaluation complète.

Dans le cas présent, le compte de connexion "HR" n'a pas les autorisations pour consulter les vues du catalogue Oracle.

```
09:55:56 WARN   Partial inspection. See errors below.
09:55:56 WARN   You can use the catalog but some objects may be missing or incomplete.
09:55:56 WARN   Ensure you have the necessary privileges.
09:55:56 WARN   Soure database version may not be supported.
```

L'inspection se poursuit avec une phase d'audit du catalogue extrait de la base distante. Cette étape permet d'identifier des faiblesses du modèle de données, ainsi que les complications éventuelles lors de la conversion. Durant cette étape, chaque composant (table, vue, index, procédure stockée) se voit attribué un score de complexité ainsi qu'une série d'annotations si nécessaire.

Une première conversion du modèle est réalisée dans la foulée. Cette opération va consulter le catalogue local de la base source, contenu dans le fichier `.pg_migrate/Source.json` pour en produire le fichier `.pg_migrate/Target.json`. Il contient la même structure que le fichier `Source.json` mais avec certaines conversions génériques.

```
09:55:56 INFO   Auditing catalog.                driver=oracle
09:55:56 INFO   Catalog audited.
09:55:56 INFO   Converted catalog for PostgreSQL.
09:55:56 INFO   Auditing catalog.                driver=oracle
09:55:56 WARN   Target catalog has pending annotations. count=16
09:55:56 INFO   Databases connections closed.    source=5 target=0
```

> Recommencer l'inspection avec un compte privilégié.

Relancer la commande `init` depuis le répertoire pour modifier le compte de connexion. Un compte

privilégié comme **system** ne rencontrera pas les erreurs d'inspection. Il est aussi possible d'octroyer les bons privilèges au compte **hr** en respectant les prérequis indiqués dans la documentation[1].

```
$ pg_migrate init --source oracle://system:manager@localhost/hr
```

Relancer l'inspection. Le catalogue ainsi extrait est stocké dans le fichier `.pg_migrate/Source.json`.

```
$ pg_migrate inspect
```

[1] https://postgresql-migrator.readthedocs.io/en/latest/references/requirements/

3.3 CRÉATION DU RAPPORT D'ÉVALUATION

> Exécuter la sous-commande `report` depuis votre répertoire de travail.

```
$ pg_migrate report
16:25:41 INFO   Generating JSON report.          report=Annotations.json
 ↪  filter=Annotations.jq
16:25:41 INFO   Generating JSON report.         report=Summary.json filter=Summary.jq
16:25:41 INFO   Generating JSON report.          report=Total.json filter=Total.jq
16:25:41 INFO   Generating Markdown report.      report=report.md
 ↪  template=report.md.tmpl
```

Consulter librement le contenu du rapport nommé `report.md`. Un tableau récapitulatif reprend le décompte des composants du catalogue, leur taille si applicable, leur score et le nombre d'annotations pour chaque catégorie.

Object	Count	Size	Score	Annotations
Roles	2		2.0	0
Schemas	1		0.1	0
Sequences	4		0.4	0
Tables	8	1.2GB	25.8	7
Triggers	6		12.7	4
Views	1		8.1	1
Indexes	13	4.1MB	2.7	0
Procedures	2		4.7	0
Functions	2		4.9	0
Packages	2		59.8	5

Plus bas, une liste des annotations est fournie pour mettre en lumière les principales causes de la complexité de la base de donnée.

- 2 Column with *missing precision*.
- 1 View with *read only view*.
- 1 Package with *transqlate: NULL comparison on identifier*.
- 2 Procedure with *transqlate: NULL comparison on identifier*.
- 4 Trigger with *transqlate: NULL comparison on identifier*.
- 1 Column with *type CLOB*.

- 2 Package with *unparsed statements*.

> Enrichir le rapport en ajoutant un décompte des colonnes par type de données.

Il est possible de composer un rapport en manipulant deux catégories de fichiers. La nature flexible et programmable de ce mécanisme de composition vous permet d'enrichir le rapport avec les données de votre choix.

- Les filtres `.jq` permettent de réduire et de manipuler des données brutes au format JSON, ici `Source.json` ;
- Les *templates* `.md.tmpl` incorporent les données filtrées dans un fichier final au format Markdown.

Nous souhaitons décompter les nombres de colonnes en fonction de leur type de données. Pour y parvenir, créer un fichier `Types.jq` dans le projet avec le contenu suivant :

```
[ $Source[].Tables[]?.Columns[].Type.Name ]
| group_by(.)
| map({Name: .[0], Count: length})
| sort_by(.Count)
| reverse
```

Ce filtre liste les types des toutes les colonnes de table du fichier `Source.json` et les regroupe par nom en créant un tableau associatif trié. Ce tableau peut ensuite être manipulé dans un *template* à l'aide d'une boucle `range`. Consulter la documentation text/template[2] du langage Go pour plus de précision.

Modifier le fichier `report.md.tmpl` en ajoutant le bout de code suivant :

```
## Types de colonnes
{{ range $type := .Types }}
- {{ $type.Count }} {{ $type.Name }}
{{- end }}
```

> Exécuter à nouveau la sous-commande `report`.

Le moteur de rapport détecte que le *template* a été modifié et s'appuie sur la donnée du fichier `Types.json`, lui-même déclaré par le filtre `Types.jq`. Le fichier intermédiaire est créé et le rapport `report.md` est mis à jour avec la nouvelle section « Types de colonnes ».

```
$ pg_migrate report
15:18:17 INFO    Generating JSON report.          report=Types.json filter=Types.jq
15:18:17 INFO    Generating Markdown report.      report=report.md
 ↪  template=report.md.tmpl
```

[2] https://pkg.go.dev/text/template

La base HR contient 21 colonnes, dont la répartition par type est listée ci-dessous.

- 19 NUMBER
- 15 VARCHAR2
- 4 DATE
- 2 CHAR
- 1 CLOB

4/ Exercice #2

- Écrire des règles de conversion
- Naviguer dans l'interface graphique
- Exporter le modèle dans des fichiers plats

4.1 CONVERSIONS GÉNÉRIQUES

> Exécuter la commande `pg_migrate convert --refresh` en mode verbeux.

Se positionner dans le dossier du projet et relancer une opération de conversion. La page de documentation[1] décrit les conversions génériques réalisées durant cette étape.

```
$ pg_migrate --verbose convert --refresh
...
... from=CHAR(2) to=char(2) path=Tables/HR.COUNTRIES/Columns/COUNTRY_ID
... from=VARCHAR2(60) to=varchar(60) path=Tables/HR.COUNTRIES/Columns/COUNTRY_NAME
... from=NUMBER to=numeric path=Tables/HR.COUNTRIES/Columns/REGION_ID
... from="NUMBER(4, 0)" to=smallint path=Tables/HR.DEPARTMENTS/Columns/DEPARTMENT_ID
... from=VARCHAR2(30) to=varchar(30)
 ↪  path=Tables/HR.DEPARTMENTS/Columns/DEPARTMENT_NAME
... from="NUMBER(6, 0)" to=integer path=Tables/HR.DEPARTMENTS/Columns/MANAGER_ID
```

Le mode verbeux vous donne plus de détails sur les modifications opérées. Par exemple, les types `NUMBER(6, 0)` d'Oracle sont convertis en `integer`. L'option `--verbose` est globale et se positionne avant la sous-commande. La deuxième option `--refresh` est propre à la sous-commande et se positionne après cette dernière. Elle force la réécriture intégrale du fichier `Source.json`.

Une exploration plus périlleuse des fichiers JSON est tout à fait possible. L'exemple ci-dessous permet de cibler les différences entre les deux fichiers pour la colonne `employees.employee_id`.

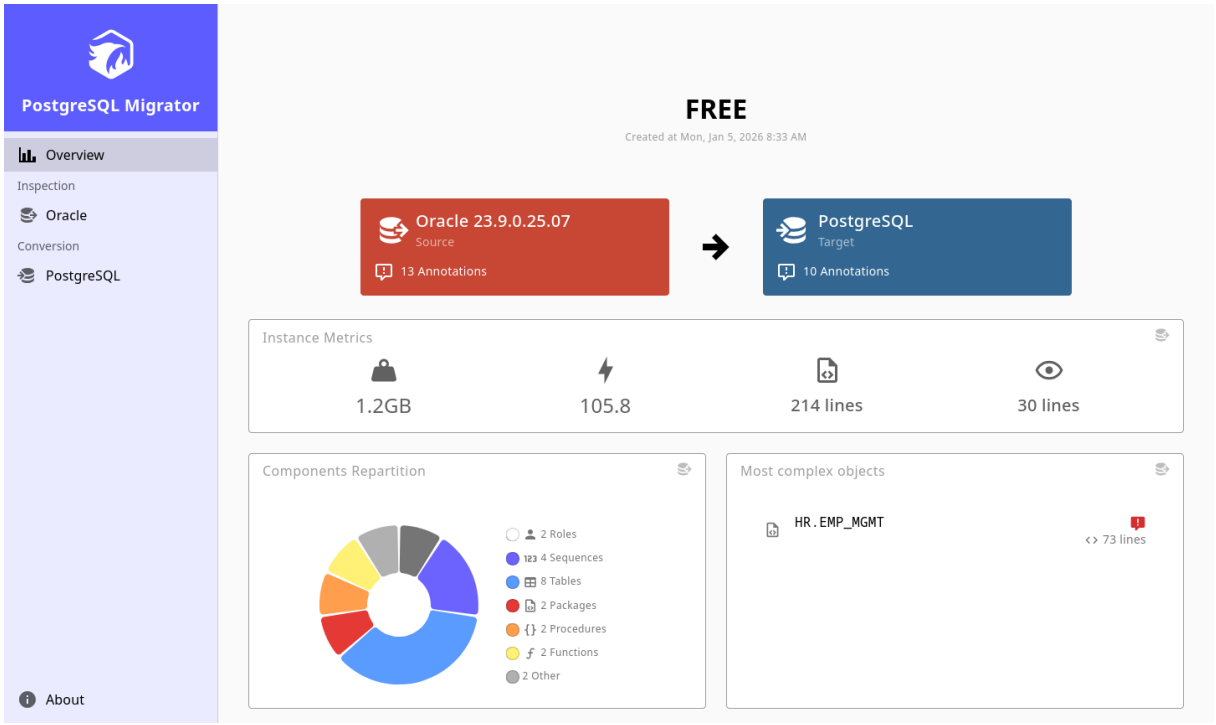
```
$ expr='.Tables[].Columns[] | select(.objectPath |
 ↪  match("Tables/hr.employees/Columns/employee_id";"i")) | .Type'
$ diff <(jq -r "$expr" .pg_migrate/Source.json) <(jq -r "$expr"
 ↪  .pg_migrate/Target.json)
2,6c2
<   "Name": "NUMBER",
<   "Params": [
<     "6",
<     "0"
<   ]
---
>   "Name": "integer"
```

[1] https://postgresql-migrator.readthedocs.io/en/latest/references/conversion/

4.2 NAVIGUER DANS L'INTERFACE GRAPHIQUE

> Exécuter la commande `pg_migrate ui`.

L'outil PostgreSQL Migrator embarque un serveur web interne pour présenter le contenu des fichiers JSON sous une forme graphique et interactive. La page d'accueil vous fournit les éléments clés de l'inspection, similaire au rapport Markdown généré précédemment. Sur la page d'accueil, on y retrouve la version du moteur, le poids cumulé des tables et des index, la somme du nombre de lignes issues des composants procéduraux, etc.



> Consulter tous les composants de la base de données.

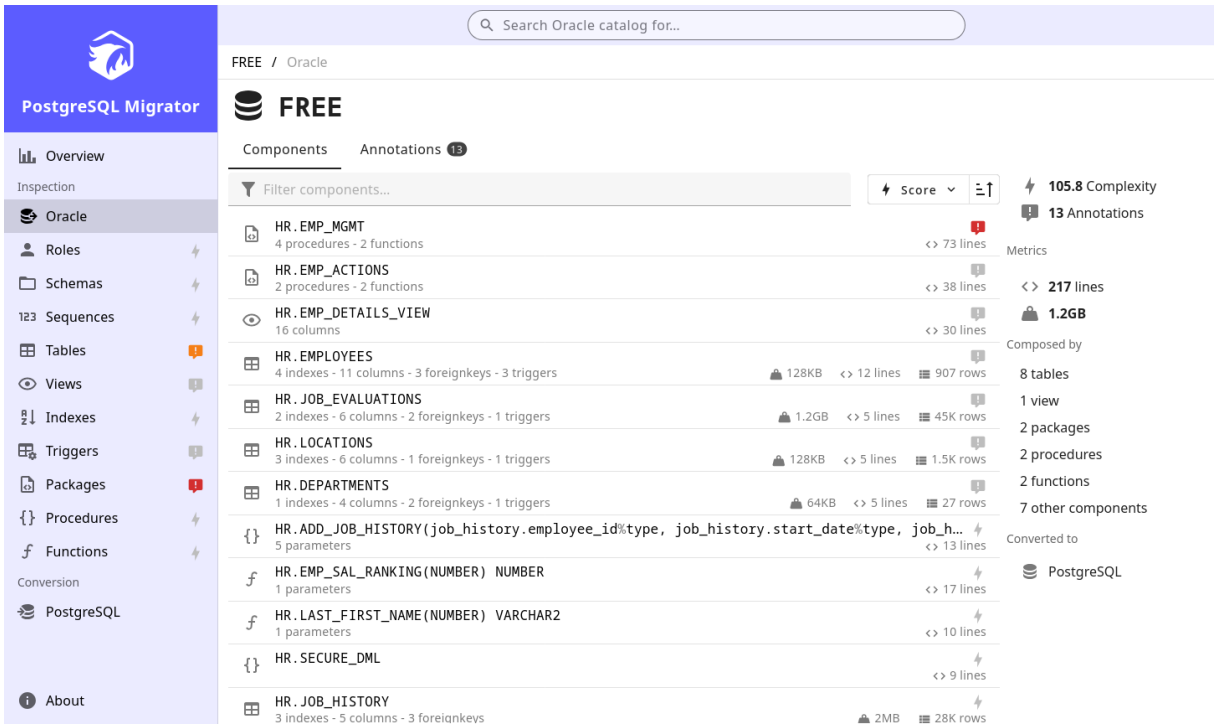
Naviguer à l'intérieur de la base Oracle pour obtenir la liste de ses objets. Il est possible de filtrer depuis une barre de recherche pour accéder à un objet en particulier. Si vous tapez le mot « history » dans la recherche globale, un résultat s'affiche pour lister les composants comportant ce mot dans leur définition ou la définition de leur parent.

> Trier par score.

Le score d'un composant est calculé selon plusieurs critères, tels que son type, le nombre de colonne ou des lignes de codes qui le constituent. Un tableau récapitulatif de tous les scores est accessible sur une page de la documentation[2]. La somme de tous les scores permet de déterminer la complexité de

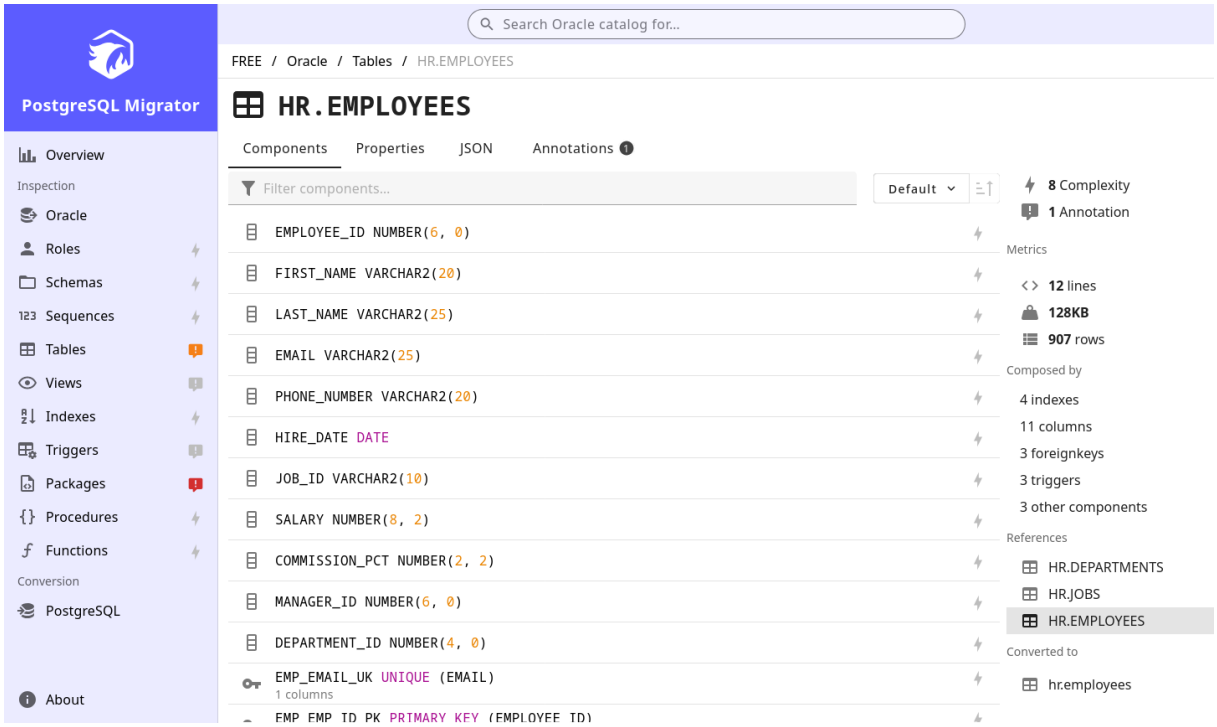
[2] https://postgresql-migrator.readthedocs.io/en/latest/references/toml/#scores-section

la migration d'une base de données dans son ensemble. Les composants dont les scores sont les plus élevés, sont *a priori* les plus complexes à porter vers PostgreSQL.



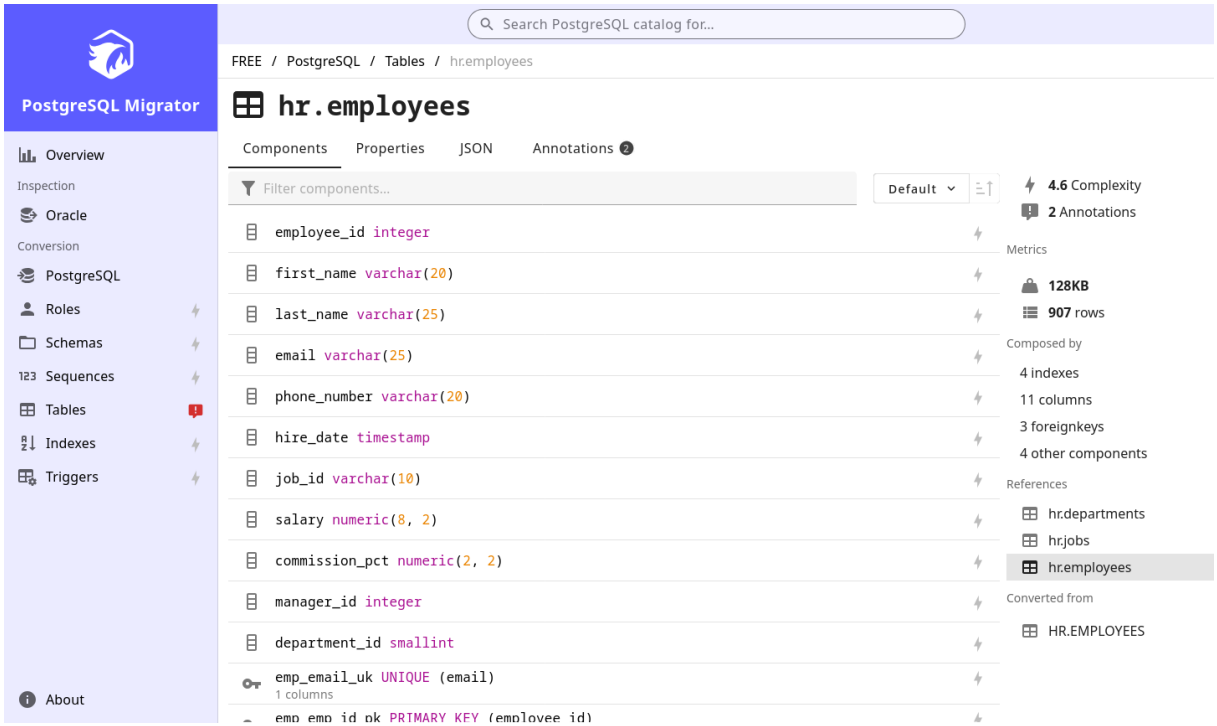
> Consulter la définition de la table EMPLOYEES.

Cliquer sur la table HR.EMPLOYEES de la liste et sélectionner le tri par défaut. Les composants représentent tout objet rattaché dans la table, comme les colonnes, les index ou les triggers. L'onglet « JSON » fournit le détail du composant, tel que présenté dans le fichier d'inventaire.



Consulter la version convertie de la table EMPLOYEES.

Dans la colonne de droite, un lien permet d'aller vers la définition de l'objet converti. On y retrouve les conversions de colonnes `NUMBER(6, 0)` en `integer`.



4.3 PROCÉDER À DES MODIFICATIONS DU MODÈLE

> Renommer le schéma HR en `public`.

Il est possible de définir des conversions personnalisées avec le fichier de configuration `pg_migrate.toml`. Chaque composant du modèle source dispose d'un chemin unique (le *path*) pour le distinguer et lui appliquer des transformations. Éditer le fichier avec un éditeur de texte et copier les lignes suivantes :

```
[[Convert.Rules]]
Path = "Schemas/HR"
Name = "public"
```

> Déclarer toutes les colonnes faisant référence à `DEPARTMENTS.DEPARTMENT_ID` en `integer`.

Nous voulons définir des règles de transformation sur les colonnes de certaines tables. Éditer à nouveau le fichier `pg_migrate.toml`.

```
[[Convert.Rules]]
Path = "Tables/HR.EMPLOYEES/Columns/DEPARTMENT_ID"
Type = "integer"

[[Convert.Rules]]
Path = "Tables/HR.DEPARTMENTS/Columns/DEPARTMENT_ID"
Type = "integer"

[[Convert.Rules]]
Path = "Tables/HR.JOB_HISTORY/Columns/DEPARTMENT_ID"
Type = "integer"
```

> Déclarer toutes les colonnes de type `DATE` en `date`, sauf pour la colonne `JOB_EVALUATIONS.EVALUATION_DA`
> qui devra être convertie en `timestamp without time zone`.

La règle `Convert.DataTypes` surcharge les conversions génériques et sera appliquée pour toutes les colonnes dont le type est déclaré à la gauche du symbole « = ». La casse est ignorée.

```
[Convert.DataTypes]
"DATE" = "date"

[[Convert.Rules]]
Path = "Tables/HR.JOB_EVALUATIONS/Columns/EVALUATION_DATE"
Type = "timestamp without time zone"
```

N'oubliez pas de relancer une conversion du modèle afin que la configuration puisse être prise en compte. Pour aller plus loin, une page de documentation[3] reprend les syntaxes et les règles de transformation possibles.

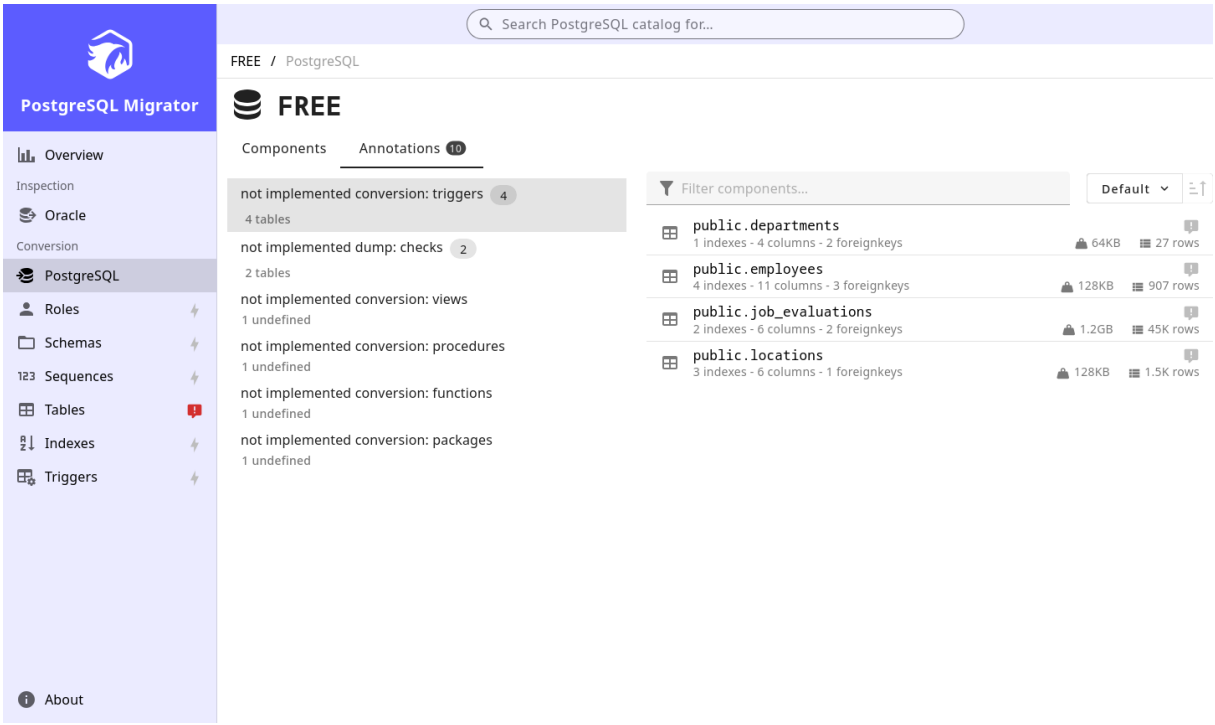
```
$ pg_migrate --verbose convert --refresh
...
... Overriding datatype rule.        match=date
... Converting object identifier.    from=HR to=public path=Schemas/HR
... Converting column type.          from="NUMBER(4, 0)" to=bigint
 ↪  path=Tables/HR.DEPARTMENTS/Columns/DEPARTMENT_ID
... Converting column type.          from="NUMBER(4, 0)" to=bigint
 ↪  path=Tables/HR.EMPLOYEES/Columns/DEPARTMENT_ID
... Converting column type.          from="NUMBER(4, 0)" to=bigint
 ↪  path=Tables/HR.JOB_HISTORY/Columns/DEPARTMENT_ID
... Converting column type.          from=DATE to="timestamp without time zone"
 ↪  path=Tables/HR.JOB_EVALUATIONS/Columns/EVALUATION_DATE
... Converting column type.          from=DATE to=date
 ↪  path=Tables/HR.EMPLOYEES/Columns/HIRE_DATE
... Converting column type.          from=DATE to=date
 ↪  path=Tables/HR.JOB_HISTORY/Columns/START_DATE
... Converting column type.          from=DATE to=date
 ↪  path=Tables/HR.JOB_HISTORY/Columns/END_DATE
```

[3] https://postgresql-migrator.readthedocs.io/en/latest/references/toml/#convertrules

4.4 GÉNÉRER LE MODÈLE DE DONNÉES CONVERTI AU FORMAT SQL

> Consulter les annotations du modèle converti.

Revenez sur la page d'accueil pour naviguer dans le catalogue PostgreSQL. L'onglet « Annotations » vous révèle les actions requises avant l'export, ainsi que les fonctionnalités non implémentées de PostgreSQL Migrator.



L'outil est réglé pour ne pas exporter le schéma et ses données s'il existe des annotations non prises en compte. Un message d'erreur survient lors de l'export avec la commande `pg_migrate dump`.

```
$ pg_migrate dump
15:59:13 ERROR  Target model has pending annotations. len=16
15:59:13 ERROR  Use --force to dump anyway.
```

Pour ignorer ce comportement, nous allons ajouter une instruction dans le fichier `pg_migrate.toml`.

```
[Dump]
Force = true
```

> Exécuter la commande `pg_migrate dump --target files --schema-only`.

La sous-commande dump est responsable de l'export des objets et de leurs données dans un format SQL compatible avec PostgreSQL. L'option `--target` permet de définir l'endroit où seront écrites les

instructions. Pour le moment, nous souhaitons un aperçu des structures converties dans des fichiers plats.

```
$ pg_migrate dump --target files --schema-only
17:22:58 INFO   Writing queries to dump/ directory.
17:22:58 INFO   Create role.     name=pdbadmin path=00002-Roles-pdbadmin-create.sql
17:22:58 INFO   Create role.     name=hr path=00001-Roles-hr-create.sql
17:22:58 INFO   Create schema.   name=public path=00003-Schemas-public-create.sql
17:22:58 INFO   Create table.    name=public.employees
 ↪  path=00006-Tables-public.employees-create.sql
17:22:58 INFO   Create table.    name=public.jobs
 ↪  path=00007-Tables-public.jobs-create.sql
17:22:58 INFO   Create table.    name=public.departments
 ↪  path=00005-Tables-public.departments-create.sql
17:22:58 INFO   Create table.    name=public.locations
 ↪  path=00009-Tables-public.locations-create.sql
17:22:58 INFO   Create table.    name=public.countries
 ↪  path=00004-Tables-public.countries-create.sql
17:22:58 INFO   Create table.    name=public.job_history
 ↪  path=00008-Tables-public.job_history-create.sql
17:22:58 INFO   Create sequence. name=public.locations_seq.
 ↪  path=00014-Sequences-public.locations_seq-create.sql
17:22:58 INFO   Create sequence. name=public.evaluations_seq.
 ↪  path=00013-Sequences-public.evaluations_seq-create.sql
17:22:58 INFO   Create table.    name=public.regions
 ↪  path=00010-Tables-public.regions-create.sql
17:22:58 INFO   Create sequence. name=public.employees_seq.
 ↪  path=00012-Sequences-public.employees_seq-create.sql
17:22:58 INFO   Create sequence. name=public.departments_seq.
 ↪  path=00011-Sequences-public.departments_seq-create.sql
...
```

L'option `--schema-only` est similaire à celle de l'outil `pg_dump`. Il s'agit d'un raccourci pour les options `--section=pre-data` et `--section=post-data`. N'hésitez pas à consulter l'aide depuis la ligne de commande pour plus de détails.

```
$ pg_migrate dump --help
Usage: pg_migrate [OPTIONS] dump [OPTIONS]

Generates DDL and/or COPY for target PostgreSQL.
Executes statements in defined PostgreSQL target database
or writes to stdout or files.
Requires successful convert.

Options:
  -c, --clean            clean (drop) objects before recreating
                         or truncate table before copy
```

```
  -a, --data-only        dump only the data, not the schema
  -f, --force            ignore unhandled annotations
  -?, --help             Show help
  -j, --jobs int         use this many parallel jobs to dump (default 4)
      --refresh-stats    refresh table statistics before planning dump (default true)
  -s, --schema-only      dump only the schema, no data
      --section string   dump named section (pre-data, data, post-data)
      --target string    DSN for query execution

--target can be stdout, files or a PostgreSQL DSN.

If stdout is not a terminal, writes to stdout.
Else if PGMTARGET is defined, sends to PostgreSQL target database.
Force file output with --target=files.

See pg_migrate --help for more informations.
```

> Consulter les fichiers présents dans le répertoire `dump/`.

Les fichiers SQL générés par la commande précédente sont accessibles depuis le dossier `dump/` du répertoire de travail.

```
$ tree dump/
dump/
├── 00001-Roles-hr-create.sql
├── 00002-Roles-pdbadmin-create.sql
├── 00003-Schemas-public-create.sql
├── 00004-Tables-public.countries-create.sql
├── 00005-Tables-public.departments-create.sql
├── 00006-Tables-public.employees-create.sql
├── 00007-Tables-public.job_evaluations-create.sql
├── 00008-Tables-public.job_history-create.sql
├── 00009-Tables-public.jobs-create.sql
├── 00010-Tables-public.locations-create.sql
├── 00011-Tables-public.regions-create.sql
...
├── 20029-Tables-public.job_evaluations-ForeignKeys-eval_employee_fk-create.sql
├── 20030-Tables-public.job_evaluations-ForeignKeys-eval_evaluator_fk-create.sql
├── 20031-Tables-public.job_history-ForeignKeys-jhist_dept_fk-create.sql
├── 20032-Tables-public.job_history-ForeignKeys-jhist_emp_fk-create.sql
├── 20033-Tables-public.job_history-ForeignKeys-jhist_job_fk-create.sql
└── 20034-Tables-public.locations-ForeignKeys-loc_c_id_fk-create.sql
```

La table `employees` est conforme à ce que nous attendions. Le schéma de destination est bien `public` et les colonnes `department_id` et `hire_date` sont respectivement de type `integer` et `date`.

```
--
-- Name: employees; Type: TABLE; Schema: public; Owner: unknown
--

CREATE TABLE "public"."employees" (
  "employee_id" integer,
  "first_name" varchar(20),
  "last_name" varchar(25),
  "email" varchar(25),
  "phone_number" varchar(20),
  "hire_date" date,
  "job_id" varchar(10),
  "salary" numeric(8, 2),
  "commission_pct" numeric(2, 2),
  "manager_id" integer,
  "department_id" integer
);
```

5/ Exercice #3

- Création des tables
- Copie des données
- Création des index et des contraintes

5.1 CONFIGURATION DE LA BASE CIBLE

> Démarrer l'instance PostgreSQL.

Si ce n'est pas déjà le cas, démarrer le conteneur PostgreSQL dans lequel nous réaliserons la copie des données. L'authentification `trust` est active et permet de ne pas s'encombrer d'un mot de passe (**mais ne le faites pas chez vous !**).

```
$ sudo docker compose up -d postgres
```

> Créer une nouvelle base nommée `hr`.

La base `hr` doit être créée comme suit. Le compte **postgres** en est le propriétaire durant la phase de migration et pourra être changé *a posteriori*.

```
$ sudo docker compose exec -ti postgres createdb -U postgres hr
```

> Configurer le projet pour se connecter à l'instance.

Pour la suite des commandes, nous souhaitons renseigner l'adresse de connexion de cette nouvelle instance pour que PostgreSQL Migrator y crée les composants à notre place. Lancer une initialisation avec un nouvel argument `--target` depuis le répertoire de projet.

```
$ pg_migrate init \
  --source=oracle://system:manager@localhost/hr \
  --target=postgres://postgres@localhost/hr
```

Cela revient à ajouter contrôler l'accès et l'authentification à l'instance avant d'ajouter une ligne dans le fichier `.env`. La variable d'environnement `PGMTARGET` est alors disponible pour déterminer la destination des instructions SQL lors de l'exécution de la commande `pg_migrate dump`.

```
$ cat .env
PGMSOURCE=oracle://system:manager@localhost/hr
PGMTARGET=postgresql://postgres@localhost/hr
```

5.2 CRÉATION DES TABLES

La première étape de la migration consiste à créer les tables converties dans la base vierge. Nous aurions pu utiliser les fichiers plats exportés précédemment avec l'option `--target=files`, nous jugeons que PostgreSQL Migrator peut s'en charger aussi bien avec son moteur interne d'exécution et d'orchestration.

> Exécuter la commande `pg_migrate dump --section=pre-data`.

La phase dite « pre-data » réunit les composants initiaux qui accueilleront la donnée (comme les schémas et leurs tables), ainsi que tout autre composant dont la structure peut être définie sans relation avec de la donnée (comme les rôles, les séquences ou les vues).

```
$ pg_migrate dump --section=pre-data
17:11:14 WARN   Ignoring unhandled annotations in target model. len=10
17:11:14 INFO   Executing queries in target PostgreSQL.
17:11:14 INFO   Create role.                    name=pdbadmin sn=00002
17:11:14 INFO   Create schema.                  name=public sn=00003
17:11:14 INFO   Create role.                    name=hr sn=00001
17:11:14 INFO   Create table.                   name=public.jobs sn=00009
17:11:14 INFO   Create table.                  name=public.job_evaluations sn=00007
17:11:14 INFO   Create table.                   name=public.departments sn=00005
17:11:14 INFO   Create table.                   name=public.regions sn=00011
17:11:14 INFO   Create table.                   name=public.employees sn=00006
17:11:14 INFO   Create table.                   name=public.locations sn=00010
17:11:14 INFO   Create table.                   name=public.job_history sn=00008
17:11:14 INFO   Create table.                   name=public.countries sn=00004
17:11:14 INFO   Create sequence.                name=public.locations_seq. sn=00015
17:11:14 INFO   Create sequence.               name=public.evaluations_seq. sn=00014
17:11:14 INFO   Create sequence.                name=public.employees_seq. sn=00013
17:11:14 INFO   Create sequence.               name=public.departments_seq. sn=00012
```

> Contrôler la structure de la table `public.employees`.

Utiliser la commande `docker compose exec` pour interagir avec l'instance du conteneur et l'invite `psql`. La table `employees` est conforme aux conversions vues précédemment.

```
$ sudo docker compose exec -ti postgres psql -U postgres -d hr -c "\d
 ↪  public.employees"
                    Table "public.employees"
     Column     |         Type          | Collation | Nullable | Default
----------------+-----------------------+-----------+----------+---------
 employee_id    | integer               |           |          |
 first_name     | character varying(20) |           |          |
 last_name      | character varying(25) |           |          |
```

```
email          | character varying(25) |           |          |
phone_number   | character varying(20) |           |          |
hire_date      | date                  |           |          |
job_id         | character varying(10) |           |          |
salary         | numeric(8,2)          |           |          |
commission_pct | numeric(2,2)          |           |          |
manager_id     | integer               |           |          |
department_id  | integer               |           |          |
```

5.3 COPIE DES DONNÉES

L'étape suivante va alimenter les tables PostgreSQL avec les données en provenance des tables Oracle. L'orchestrateur interne sélectionne les tables les plus volumineuses en priorité et crée autant de processus que définis par l'argument `--jobs` (4 par défaut).

> Exécuter la commande `pg_migrate dump --data-only`.

L'option `--data-only` est équivalente à `--section=data`. Si vous souhaitez recommencer la copie intégrale des tables, l'option `--clean` ajoutera une instruction `truncate` avant chaque table. Toutefois, assurez-vous de ne pas avoir créé de contraintes de clé étrangère.

Enfin, il est possible de désactiver la collecte des statistiques avec l'option `--refresh-stats=false` s'il y a eu peu de changement entre deux exports. Les statistiques servent à déterminer la priorité des tables entre elles.

```
$ pg_migrate dump --section=data
17:35:17 INFO   Executing queries in target PostgreSQL.
17:35:17 INFO   Collecting statistics, this may take a while...
```

À l'issue du traitement d'une table, une série de métriques permet d'appréhender le débit de transfert pour cette migration et d'envisager des optimisations pour tenter d'accélérer la phase de copie des données.

```
... table=public.regions elapsed=80ms data=54B rows=5 rate="62 rows/s" sn=10008
... table=public.departments elapsed=90ms data=643B rows=27 rate="300 rows/s"
 ↪  sn=10002
... table=public.jobs elapsed=10ms data=1.1KB rows=29 rate="2.9K rows/s" sn=10006
... table=public.employees elapsed=50ms data=78KB rows=907 rate="18.1K rows/s"
 ↪  sn=10003
... table=public.locations elapsed=60ms data=68.2KB rows=1523 rate="25.4K rows/s"
 ↪  sn=10007
... table=public.countries elapsed=10ms data=577B rows=35 rate="3.5K rows/s" sn=10001
... table=public.job_history elapsed=390ms data=1.6MB rows=28010 rate="71.8K rows/s"
 ↪  sn=10005
... table=public.job_evaluations elapsed=17.17s data=401.5MB rows=45000 rate="2.6K
 ↪  rows/s" sn=10004
```

Un résumé de l'ensemble est fourni dès que la dernière table a été traitée. Il permet de contrôler que le débit global est satisfaisant et de connaître la quantité de mémoire consommée au plus haut de l'activité de transfert (*memory peak*).

```
17:35:40 INFO   Copy completed. tables=8 data=403.1MB elapsed=17.62s
 ↪  throughput=17.1MB/s jobs=4 mem=37.2MB
```

Une des tâches de copie consiste à rattraper les valeurs pour toutes les séquences du modèle converti.

```
17:35:23 INFO   Restart sequences.              schema=public sn=10009
```

5.4 CRÉATION DES INDEX ET DES CONTRAINTES

> Exécuter la commande `pg_migrate dump --section=post-data`.

La création des index et l'activation des contraintes des tables arrivent en dernière étape, celle dite « post-data ». Durant cette étape, l'intégrité des données est mise à l'épreuve avec la validation des références de clés étrangères ou de l'unicité des données de colonne.

L'orchestrateur interne de PostgreSQL Migrator se charge de construire les index et les clés primaires en priorité et résout les dépendances croisées entre tables référencées pour limiter le risque de verrous (*deadlocks*).

```
$ pg_migrate dump --section=post-data
09:02:53 INFO   Executing queries in target PostgreSQL.
09:02:53 INFO   Create table key.               name=public.jobs key=job_id_pk
 ↪  sn=20007
09:02:53 INFO   Create table key.               name=public.regions key=reg_id_pk
 ↪  sn=20009
09:02:53 INFO   Create table key.               name=public.job_evaluations
 ↪  key=eval_id_pk sn=20005
09:02:53 INFO   Create table key.               name=public.departments
 ↪  key=dept_id_pk sn=20002
09:02:53 INFO   Create table key.               name=public.employees
 ↪  key=emp_email_uk sn=20003
09:02:53 INFO   Create table key.               name=public.locations key=loc_id_pk
 ↪  sn=20008
09:02:53 INFO   Create table key.               name=public.job_history
 ↪  key=jhist_emp_id_st_date_pk sn=20006
09:02:53 INFO   Create table key.               name=public.countries
 ↪  key=country_c_id_pk sn=20001
09:02:53 INFO   Create table key.               name=public.employees
 ↪  key=emp_emp_id_pk sn=20004
09:02:53 INFO   Create Index.                   name=public.departments
 ↪  idx=departments_location_id_idx sn=20010
09:02:53 INFO   Create Index.                   name=public.locations
 ↪  idx=locations_city_idx sn=20020
09:02:53 INFO   Create Index.                   name=public.employees
 ↪  idx=employees_department_id_idx sn=20011
09:02:53 INFO   Create foreign key.             name=public.countries
 ↪  fk=countr_reg_fk sn=20023
09:02:53 INFO   Create Index.                   name=public.employees
 ↪  idx=employees_job_id_idx sn=20012
09:02:53 INFO   Create Index.                   name=public.locations
 ↪  idx=locations_country_id_idx sn=20021
09:02:53 INFO   Create Index.                   name=public.employees
 ↪  idx=employees_manager_id_idx sn=20013
```

```
09:02:53 INFO   Create Index.                    name=public.locations
 ↪  idx=locations_state_province_idx sn=20022
09:02:53 INFO   Create Index.                    name=public.employees
 ↪  idx=employees_last_name_first_name_idx sn=20014
09:02:53 INFO   Create foreign key.              name=public.departments
 ↪  fk=dept_loc_fk sn=20024
09:02:53 INFO   Create foreign key.             name=public.locations fk=loc_c_id_fk
 ↪  sn=20034
09:02:53 INFO   Create Index.                    name=public.job_history
 ↪  idx=job_history_department_id_idx sn=20017
09:02:53 INFO   Create foreign key.              name=public.departments
 ↪  fk=dept_mgr_fk sn=20025
09:02:53 INFO   Create Index.                    name=public.job_evaluations
 ↪  idx=job_evaluations_employee_id_idx sn=20015
09:02:53 INFO   Create foreign key.             name=public.employees fk=emp_dept_fk
 ↪  sn=20026
09:02:53 INFO   Create foreign key.              name=public.employees fk=emp_job_fk
 ↪  sn=20027
09:02:53 INFO   Create foreign key.              name=public.employees
 ↪  fk=emp_manager_fk sn=20028
09:02:53 INFO   Create Index.                    name=public.job_history
 ↪  idx=job_history_employee_id_idx sn=20018
09:02:53 INFO   Create Index.                    name=public.job_history
 ↪  idx=job_history_job_id_idx sn=20019
09:02:53 INFO   Create Index.                    name=public.job_evaluations
 ↪  idx=job_evaluations_evaluator_id_idx sn=20016
09:02:53 INFO   Create foreign key.              name=public.job_history
 ↪  fk=jhist_dept_fk sn=20031
09:02:53 INFO   Create foreign key.              name=public.job_evaluations
 ↪  fk=eval_employee_fk sn=20029
09:02:53 INFO   Create foreign key.              name=public.job_history
 ↪  fk=jhist_emp_fk sn=20032
09:02:53 INFO   Create foreign key.              name=public.job_evaluations
 ↪  fk=eval_evaluator_fk sn=20030
09:02:53 INFO   Create foreign key.              name=public.job_history
 ↪  fk=jhist_job_fk sn=20033
```

Contrôler la structure de la table `public.employees`.

Les tables disposent bien de leur clé primaire et index. Les références de clé étrangères sont en place. Aucune erreur n'a été observée durant la phase *post-data*, les données sont cohérentes entre elles.

```
$ sudo docker compose exec -ti postgres psql -U postgres -d hr -c "\d
 ↪  public.employees"
                      Table "public.employees"
     Column     |         Type          | Collation | Nullable | Default
----------------+-----------------------+-----------+----------+---------
```

```
 employee_id    | integer               |           | not null |
 first_name     | character varying(20) |           |          |
 last_name      | character varying(25) |           |          |
 email          | character varying(25) |           |          |
 phone_number   | character varying(20) |           |          |
 hire_date      | date                  |           |          |
 job_id         | character varying(10) |           |          |
 salary         | numeric(8,2)          |           |          |
 commission_pct | numeric(2,2)          |           |          |
 manager_id     | integer               |           |          |
 department_id  | integer               |           |          |
Indexes:
    "emp_emp_id_pk" PRIMARY KEY, btree (employee_id)
    "emp_email_uk" UNIQUE CONSTRAINT, btree (email)
    "employees_department_id_idx" btree (department_id)
    "employees_job_id_idx" btree (job_id)
    "employees_last_name_first_name_idx" btree (last_name, first_name)
    "employees_manager_id_idx" btree (manager_id)
Foreign-key constraints:
    "emp_dept_fk" FOREIGN KEY (department_id) REFERENCES departments(department_id)
    "emp_job_fk" FOREIGN KEY (job_id) REFERENCES jobs(job_id)
    "emp_manager_fk" FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
Referenced by:
    TABLE "departments" CONSTRAINT "dept_mgr_fk" FOREIGN KEY (manager_id) REFERENCES
↪   employees(employee_id)
    TABLE "employees" CONSTRAINT "emp_manager_fk" FOREIGN KEY (manager_id)
↪   REFERENCES employees(employee_id)
    TABLE "job_evaluations" CONSTRAINT "eval_employee_fk" FOREIGN KEY (employee_id)
↪   REFERENCES employees(employee_id)
    TABLE "job_evaluations" CONSTRAINT "eval_evaluator_fk" FOREIGN KEY
↪   (evaluator_id) REFERENCES employees(employee_id)
    TABLE "job_history" CONSTRAINT "jhist_emp_fk" FOREIGN KEY (employee_id)
↪   REFERENCES employees(employee_id)
```

6/ Exercice #4

- Décompter les lignes entre les deux bases
- Contrôler l'intégrité des données
- Déclencher un `VACUUM FREEZE`

6.1 VALIDATION DES DONNÉES

La validation de l'état des lignes est une étape cruciale pour prendre la décision de basculer les applications sur la nouvelle base PostgreSQL sans risque de perte d'information. Si le moindre doute subsiste sur la qualité ou la quantité de données, il faut envisager d'annuler les opérations de migration.

Pour le moment, PostgreSQL Migrator ne propose pas de rapport de validation d'une migration de données.

> Décompter les lignes de plusieurs tables.

Il nous appartient de réaliser le décompte du nombre de lignes entre la base source et la base cible PostgreSQL. Les fonctions suivantes établissent une connexion à leur base respective et décomptent les lignes des tables issues du fichier `Source.json`.

```
function count_source {
    local SQLPLUS="sudo docker compose exec -T oracle sqlplus -S
    ↪  hr/phoenix@localhost/hr"
    local filter=".Tables | sort_by(.Name | ascii_downcase) | .[] | @text
    ↪  \"\(.Schema).\(.Name)\""
    local tables=$(jq -r "$filter" .pg_migrate/Source.json)
    for table in $tables; do
        cat <<- EOF | $SQLPLUS | grep -v '^$'
set newpage none head off feedback off
set markup csv on quote off
SELECT '$table', count(*) FROM $table;
EOF
    done
}

function count_target {
    local PSQL="sudo docker compose exec -T postgres psql -d hr -U postgres"
    local filter=".Tables | sort_by(.Name | ascii_downcase) | .[] | @text
    ↪  \"\(.Schema).\(.Name)\""
    local tables=$(jq -r "$filter" .pg_migrate/Target.json)
    for table in $tables; do
        $PSQL --csv --pset pager=off -tc "SELECT '$table', count(*) FROM $table"
    done
}
```

Exécuter le décompte avec la commande suivante.

```
$ paste -d ',' <(count_source) <(count_target) | column -s, -t
HR.COUNTRIES        35     public.countries        35
HR.DEPARTMENTS      27     public.departments      27
```

```
HR.EMPLOYEES         907    public.employees         907
HR.JOB_EVALUATIONS   45000  public.job_evaluations   45000
HR.JOB_HISTORY       28010  public.job_history       28010
HR.JOBS              29     public.jobs              29
HR.LOCATIONS         1523   public.locations         1523
HR.REGIONS           5      public.regions           5
```

> Valider l'intégrité d'un échantillon de données.

Le deuxième contrôle vise à certifier que la donnée n'a pas subi de transformation au cours de la copie. Le changement d'encodage des chaînes de caractères ou la troncature non désirée d'une valeur sont des erreurs que nous souhaitons éviter à tout prix.

Créer les deux fonctions `select_*` suivantes. Elles établissent une connexion à leur base respective, exécutent la requête passée en paramètre et formatent le résultat au format CSV. Une attention particulière est nécessaire pour afficher correctement les données afin qu'elles aient un rendu équivalent entre les deux systèmes.

```
function select_source {
    test -z "$1" && return
    local SQLPLUS="sudo docker compose exec -T oracle sqlplus -S
     ↪  hr/phoenix@localhost/hr"
    n=$'\n'; cat <<- EOF | $SQLPLUS | grep -v '^$'
set newpage none head off feedback off
set markup csv on quote off
column salary format 999999.99
column commission_pct format 0.99
column score format 99.9
alter session set nls_date_format='YYYY-MM-DD';
alter session set nls_timestamp_format='YYYY-MM-DD HH24:MI:SS';
alter session set nls_timestamp_tz_format='YYYY-MM-DD HH24:MI:SSTZH';
$1;
EOF
}

function select_target {
    test -z "$1" && return
    local PSQL="sudo docker compose exec -T postgres psql -d hr -U postgres"
    $PSQL --csv --pset pager=off -tc "$1"
}
```

À présent, nous pouvons lancer la commande suivante pour comparer les résultats d'une requête qui parcourt une fraction de la table `employees`. Si la commande n'affiche aucun résultat, les données sont scrupuleusement identiques.

```
diff -u \
```

```
  <(select_source "SELECT * FROM hr.employees WHERE MOD(employee_id, 10) = 0 ORDER BY
   ↪  employee_id") \
  <(select_target "SELECT * FROM public.employees WHERE employee_id % 10 = 0 ORDER BY
   ↪  employee_id")
```

La comparaison des données CLOB est plus coûteuse en ressources. Pour la table `job_evaluations`, nous allons uniquement compter le nombre de caractères dans la colonne `comments` à l'aide des méthodes `DBMS_LOB.GETLENGTH` et `char_length`.

```
diff -u \
  <(select_source "alter session set nls_date_format='YYYY-MM-DD HH24:MI:SS';
                   SELECT evaluation_id, employee_id, evaluator_id,
                          evaluation_date, score, DBMS_LOB.GETLENGTH(comments)
                     FROM hr.job_evaluations
                    WHERE MOD(evaluation_id, 500) = 0
                    ORDER BY evaluation_id") \
  <(select_target "SELECT evaluation_id, employee_id, evaluator_id,
                          evaluation_date, score, char_length(comments)
                     FROM public.job_evaluations
                    WHERE evaluation_id % 500 = 0
                    ORDER BY evaluation_id")
```

6.2 TÂCHES DE MAINTENANCE

La bascule du modèle et des données vers PostgreSQL s'accompagne d'une série d'opérations de maintenance qui dépend de votre organisation. L'activation de les sondes de supervision et la planification des sauvegardes sont sans conteste les premières actions à entreprendre pour assurer le suivi et la sécurité de la plateforme.

> Réaliser un `VACUUM FREEZE` pour sécuriser les données à terme.

Une opération préventive, appelée *gel des lignes*, est fortement recommandée à l'issue d'un chargement massif de lignes dans une base PostgreSQL.

Rappelons qu'un numéro de transaction est attaché à chaque nouvelle ligne dans une base, afin de garantir l'isolation des sessions entre elles. Or ces numéros de transaction sont encodés sur 32 bits et sont recyclés à terme. Il y a donc un risque de mélanger les opérations passées et celles à venir au moment du rebouclage (*wraparound*). Afin d'éviter ce phénomène, l'opération `VACUUM FREEZE` « gèle » les vieux enregistrements, afin que ceux-ci ne se retrouvent pas brusquement dans le futur.

Exécuter la commande suivante pour déclencher un gel préventif des table de la base `hr`. Il est possible de traiter plusieurs tables simultanément avec l'option `--jobs`.

```
$ sudo docker compose exec -ti postgres vacuumdb -U postgres --freeze --jobs=4 hr
vacuumdb: vacuuming database "hr"
```

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.

7/ 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.