



STID
Aurillac
Statistique &
informatique
décisionnelle
Cybersécurité

UCA
INSTITUT DE
TECHNOLOGIE
UNIVERSITÉ
Clermont Auvergne

UNIVERSITÉ CLERMONT AUVERGNE — CAMPUS
AURILLAC

Projet — PostgreSQL → MongoDB

(Jointures, export JSON, import MongoDB, dénormalisation)

Réalisé par :

Auspicia DJIMA

Encadrant :

BENALI RAMZI

Date : 10 février 2026

Sommaire

1	Objectif du projet	2
2	Environnement et données	2
2.1	SGBD et outils utilisés	2
2.2	Tables / collections manipulées	2
3	Partie 1 — Exploration initiale sous PostgreSQL	2
3.1	Aperçu rapide des tables	2
3.2	Contrôle des volumes	3
3.3	Exploration	3
4	Partie 2 — Jointures SQL et optimisation (vue matérialisée)	4
4.1	Jointure <code>employees + salaries + titles</code>	4
4.2	Temps d'exécution mesuré	5
4.3	Création d'une vue matérialisée	5
4.4	Comparaison avec EXPLAIN (ANALYZE, BUFFERS)	5
4.5	Conclusion courte (Partie 2)	6
5	Partie 3 — Export PostgreSQL vers fichiers JSON	6
5.1	Export ligne par ligne avec <code>row_to_json</code>	6
5.2	Agrégation en tableau JSON avec <code>json_agg</code>	7
5.3	Export dans des fichiers via <code>COPY</code>	8
6	Partie 4 — Import MongoDB des fichiers JSON	9
6.1	Import via <code>mongoimport</code>	9
6.2	Vérifications dans <code>mongosh</code>	9
7	Partie 5 — Jointures MongoDB et dénormalisation	10
7.1	Création d'index pour accélérer <code>\$lookup</code>	10
7.2	Jointure <code>employees ↔ titles (1 \$lookup)</code>	10
7.3	Jointure <code>employees ↔ titles ↔ salaries (2 \$lookup)</code>	10
7.4	Temps d'exécution de la jointure MongoDB	11
7.5	Dénormalisation avec <code>\$project</code> (suppression des doublons)	11
7.6	Sauvegarde dans une nouvelle collection (<code>\$merge</code>)	12
7.7	Délai d'accès aux informations après dénormalisation	12
8	Partie 6 — Conclusions	13
8.1	Quand la dénormalisation est opportune	13
8.2	Quand la dénormalisation n'est pas opportune	13

1 Objectif du projet

Ce projet a pour objectif de :

- manipuler les données *employees* dans PostgreSQL (exploration, contrôles, jointures) ;
- mesurer le coût d'une jointure SQL (`employees + salaries + titles`) ;
- optimiser l'accès via une **vue matérialisée** ;
- exporter les tables PostgreSQL en fichiers **JSON** ;
- importer ces fichiers JSON dans MongoDB via **mongoimport** ;
- reproduire l'équivalent des jointures SQL dans MongoDB via **\$lookup** ;
- **dénormaliser** les données (embedding) pour accélérer l'accès aux informations par employé ;
- conclure sur les cas où cette dénormalisation est pertinente, et ceux où elle ne l'est pas.

2 Environnement et données

2.1 SGBD et outils utilisés

Les opérations ont été réalisées sur un environnement **Windows** (chemins `C:\...`) avec :

- **PostgreSQL 18** (répertoire de données observé : `C:/Program Files/PostgreSQL/18/data`) ;
- **MongoDB 8.2.0** et **mongosh 2.5.8** (versions affichées lors de la connexion).

2.2 Tables / collections manipulées

Les 6 entités suivantes ont été manipulées dans PostgreSQL puis importées dans MongoDB :

- `employees`
- `departments`
- `dept_emp`
- `dept_manager`
- `titles`
- `salaries`

3 Partie 1 — Exploration initiale sous PostgreSQL

3.1 Aperçu rapide des tables

Les requêtes suivantes permettent de vérifier rapidement le contenu et l'ordre des données.

Listing 1 – Aperçu des tables (LIMIT 20)

```

SELECT * FROM departments ORDER BY dept_no LIMIT 20;
SELECT * FROM employees ORDER BY emp_no LIMIT 20;

SELECT * FROM dept_emp ORDER BY emp_no, from_date LIMIT 20;
SELECT * FROM dept_manager ORDER BY dept_no, from_date LIMIT 20;

SELECT * FROM titles ORDER BY emp_no, from_date LIMIT 20;
SELECT * FROM salaries ORDER BY emp_no, from_date LIMIT 20;

```

3.2 Contrôle des volumes

On vérifie ensuite le nombre de lignes dans chaque table.

Listing 2 – Comptage des lignes par table

```

SELECT 'departments' AS table, COUNT(*) AS nb_lignes FROM
departments
UNION ALL SELECT 'employees', COUNT(*) FROM employees
UNION ALL SELECT 'dept_emp', COUNT(*) FROM dept_emp
UNION ALL SELECT 'dept_manager', COUNT(*) FROM dept_manager
UNION ALL SELECT 'titles', COUNT(*) FROM titles
UNION ALL SELECT 'salaries', COUNT(*) FROM salaries;

```

Table	Nombre de lignes
departments	9
employees	300024
dept_emp	331603
dept_manager	24
titles	443308
salaries	2844047

TABLE 1 – Volumes observés (PostgreSQL puis confirmés dans MongoDB).

3.3 Exploration

Données *actuels*

On peut identifier les affectations/titres/salaires en cours via la valeur 9999-01-01.

Listing 3 – Département, titres, salaires en cours

```

-- Département actuel (affectations en cours)
SELECT * FROM dept_emp
WHERE to_date = DATE '9999-01-01'

```

```

ORDER BY emp_no LIMIT 20;

-- Titres actuels
SELECT * FROM titles
WHERE to_date = DATE '9999-01-01',
ORDER BY emp_no LIMIT 20;

-- Salaires actuels
SELECT * FROM salaries
WHERE to_date = DATE '9999-01-01',
ORDER BY emp_no LIMIT 20;

```

Schéma (colonnes)

Listing 4 – Structure des tables (information_schema)

```

SELECT table_name, column_name, data_type FROM information_schema.
    columns
WHERE table_schema = 'public'
    AND table_name IN ('employees', 'departments', 'dept_emp',
                        'dept_manager', 'titles', 'salaries')
ORDER BY table_name, ordinal_position;

```

4 Partie 2 — Jointures SQL et optimisation (vue matérialisée)

4.1 Jointure employees + salaries + titles

La jointure se fait via emp_no.

Listing 5 – Jointure SQL employees + salaries + titles

```

SELECT
    employe.emp_no,
    employe.first_name,
    employe.last_name,
    salaire.salary,
    salaire.from_date AS salaire_date_debut,
    salaire.to_date AS salaire_date_fin,
    titre.title,
    titre.from_date AS titre_date_debut,
    titre.to_date AS titre_date_fin
FROM employees AS employe

```

```

JOIN salaries AS salaire ON salaire.emp_no = employe.emp_no
JOIN titles AS titre ON titre.emp_no = employe.emp_no;

```

4.2 Temps d'exécution mesuré

Résultat observé lors de l'exécution :

- Total rows : 4 638 507
- Temps : 16 secs 967 msec (soit 16 967 ms)

4.3 Crédation d'une vue matérialisée

On crée une vue matérialisée contenant le résultat de la jointure.

Listing 6 – Crédation de la vue matérialisée

```

CREATE MATERIALIZED VIEW vue_mat_employees_salaires_titres AS
SELECT
    employe.emp_no,
    employe.first_name,
    employe.last_name,
    salaire.salary,
    salaire.from_date AS salaire_date_debut,
    salaire.to_date AS salaire_date_fin,
    titre.title,
    titre.from_date AS titre_date_debut,
    titre.to_date AS titre_date_fin
FROM employees AS employe
JOIN salaries AS salaire ON salaire.emp_no = employe.emp_no
JOIN titles AS titre ON titre.emp_no = employe.emp_no;

```

4.4 Comparaison avec EXPLAIN (ANALYZE, BUFFERS)

Accès à la vue matérialisée

Listing 7 – EXPLAIN sur COUNT(*) de la vue matérialisée

```

EXPLAIN (ANALYZE, BUFFERS)
SELECT COUNT(*)
FROM vue_mat_employees_salaires_titres;

```

Temps observé : Execution Time : 532.128 ms.

Accès à la jointure directe

Listing 8 – EXPLAIN sur COUNT(*) de la jointure directe

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT COUNT(*)
FROM employees AS employe
JOIN salaries AS salaire ON salaire.emp_no = employe.emp_no
JOIN titles AS titre ON titre.emp_no = employe.emp_no;
```

Temps observé : **Execution Time : 1460.525 ms.**

Le plan montre notamment des *Parallel Seq Scan* et des *Parallel Hash Join*, ainsi que des lectures/disques via Buffers: `shared hit=7610 read=13116`.

Opération	Temps observé
Jointure SQL (résultat complet, 4 638 507 lignes)	16 967 ms
COUNT(*) sur vue matérialisée (EX- PLAIN ANALYZE)	532.128 ms

TABLE 2 – Comparaison des coûts (SQL).

4.5 Conclusion courte (Partie 2)

La vue matérialisée réduit le temps d'accès pour compter les lignes (**532.128 ms** contre **16 967 ms**) car elle évite de recalculer la jointure à chaque requête : les données jointes sont déjà matérialisées et peuvent être relues plus directement.

5 Partie 3 — Export PostgreSQL vers fichiers JSON

5.1 Export ligne par ligne avec `row_to_json`

Chaque enregistrement est converti en document JSON.

Listing 9 – `row_to_json` : un document JSON par ligne (exemples)

```
-- employees
SELECT row_to_json(ligne)
FROM (SELECT * FROM employees ORDER BY emp_no) AS ligne;

-- departments
SELECT row_to_json(ligne)
FROM (SELECT * FROM departments ORDER BY dept_no) AS ligne;

-- dept_emp
SELECT row_to_json(ligne)
FROM (SELECT * FROM dept_emp ORDER BY emp_no, dept_no, from_date)
AS ligne;
```

```

-- dept_manager
SELECT row_to_json(ligne)
FROM (SELECT * FROM dept_manager ORDER BY dept_no, emp_no,
from_date) AS ligne;

-- titles
SELECT row_to_json(ligne)
FROM (SELECT * FROM titles ORDER BY emp_no, from_date, title) AS
ligne;

-- salaries
SELECT row_to_json(ligne)
FROM (SELECT * FROM salaries ORDER BY emp_no, from_date) AS ligne;

```

5.2 Agrégation en tableau JSON avec json_agg

Ici, l'ensemble des lignes est regroupé dans un **tableau JSON** unique.

Listing 10 – json_agg : agrégation des documents en tableau JSON (exemples)

```

-- employees
SELECT json_agg(row_to_json(ligne))
FROM (SELECT * FROM employees ORDER BY emp_no) AS ligne;

-- departments
SELECT json_agg(row_to_json(ligne))
FROM (SELECT * FROM departments ORDER BY dept_no) AS ligne;

-- dept_emp
SELECT json_agg(row_to_json(ligne))
FROM (SELECT * FROM dept_emp ORDER BY emp_no, dept_no, from_date)
AS ligne;

-- dept_manager
SELECT json_agg(row_to_json(ligne))
FROM (SELECT * FROM dept_manager ORDER BY dept_no, emp_no,
from_date) AS ligne;

-- titles
SELECT json_agg(row_to_json(ligne))
FROM (SELECT * FROM titles ORDER BY emp_no, from_date, title) AS
ligne;

```

```
-- salaries
SELECT json_agg(row_to_json(ligne))
FROM (SELECT * FROM salaries ORDER BY emp_no, from_date) AS ligne;
```

5.3 Export dans des fichiers via COPY

Problème rencontré : COPY TO écrit un fichier **côté serveur PostgreSQL**.

Solution appliquée : création d'un répertoire local accessible au serveur (instance locale) : C:\pg_exports.

Listing 11 – COPY : export des 6 tables en fichiers JSON

```
-- Dossier cree au prealable : C:\pg_exports

COPY (
    SELECT json_agg(row_to_json(ligne))
    FROM (SELECT * FROM employees ORDER BY emp_no) AS ligne
) TO 'C:/pg_exports/employees.json';

COPY (
    SELECT json_agg(row_to_json(ligne))
    FROM (SELECT * FROM departments ORDER BY dept_no) AS ligne
) TO 'C:/pg_exports/departments.json';

COPY (
    SELECT json_agg(row_to_json(ligne))
    FROM (SELECT * FROM dept_emp ORDER BY emp_no, dept_no, from_date
        ) AS ligne
) TO 'C:/pg_exports/dept_emp.json';

COPY (
    SELECT json_agg(row_to_json(ligne))
    FROM (SELECT * FROM dept_manager ORDER BY dept_no, emp_no,
        from_date) AS ligne
) TO 'C:/pg_exports/dept_manager.json';

COPY (
    SELECT json_agg(row_to_json(ligne))
    FROM (SELECT * FROM titles ORDER BY emp_no, from_date, title) AS ligne
) TO 'C:/pg_exports/titles.json';

COPY (
    SELECT json_agg(row_to_json(ligne))
    FROM (SELECT * FROM salaries ORDER BY emp_no, from_date) AS ligne
)
```

```
) TO 'C:/pg_exports/salaries.json';
```

6 Partie 4 — Import MongoDB des fichiers JSON

6.1 Import via mongoimport

On importe chaque fichier JSON (tableau JSON) dans une base MongoDB nommée `employees`, avec une collection par fichier.

Listing 12 – Commandes mongoimport (6 collections)

```
mongoimport --db employees --collection employees      --file "C:\  
    pg_exports\employees.json"      --jsonArray --drop  
mongoimport --db employees --collection departments   --file "C:\  
    pg_exports\departments.json"    --jsonArray --drop  
mongoimport --db employees --collection dept_emp     --file "C:\  
    pg_exports\dept_emp.json"      --jsonArray --drop  
mongoimport --db employees --collection dept_manager  --file "C:\  
    pg_exports\dept_manager.json"  --jsonArray --drop  
mongoimport --db employees --collection titles        --file "C:\  
    pg_exports\titles.json"        --jsonArray --drop  
mongoimport --db employees --collection salaries       --file "C:\  
    pg_exports\salaries.json"      --jsonArray --drop
```

Résultats observés lors des imports :

- `employees` : **300024** documents importés
- `departments` : **9** documents importés
- `dept_emp` : **331603** documents importés
- `dept_manager` : **24** documents importés
- `titles` : **443308** documents importés
- `salaries` : **2844047** documents importés

6.2 Vérifications dans mongosh

Listing 13 – Vérification des collections et comptages

```
use employees  
show collections  
  
db.employees.countDocuments()  
db.departments.countDocuments()  
db.dept_emp.countDocuments()  
db.dept_manager.countDocuments()
```

```

db.titles.countDocuments()
db.salaries.countDocuments()

db.employees.findOne()

```

Exemple observé (`findOne`) :

```

— emp_no: 10001
— birth_date: '1953-09-02'
— first_name: 'Georgi'
— last_name: 'Facello'
— gender: 'M'
— hire_date: '1986-06-26'

```

7 Partie 5 — Jointures MongoDB et dénormalisation

7.1 Crédation d'index pour accélérer \$lookup

Avant d'effectuer des jointures, on crée des index sur `emp_no` dans les collections du côté “many”.

Listing 14 – Index sur `emp_no` pour `titles` et `salaries`

```

db.titles.createIndex({ emp_no: 1 })
db.salaries.createIndex({ emp_no: 1 })

```

7.2 Jointure `employees` ↔ `titles` (1 \$lookup)

Listing 15 – Jointure MongoDB `employees` + `titles` (test sur 1 employé)

```

db.employees.aggregate([
  { $match: { emp_no: 10001 } },
  { $lookup: { from: "titles", localField: "emp_no", foreignField:
    "emp_no", as: "titles" } }
]).toArray()

```

7.3 Jointure `employees` ↔ `titles` ↔ `salaries` (2 \$lookup)

Listing 16 – Pipeline de jointure `employees` + `titles` + `salaries`

```

const pipelineJointure_3 = [
  { $lookup: { from: "titles", localField: "emp_no",
    foreignField: "emp_no", as: "titles" } },
  { $lookup: { from: "salaries", localField: "emp_no",
    foreignField: "emp_no", as: "salaries" } }
]

```

7.4 Temps d'exécution de la jointure MongoDB

Mesure réalisée via `explain("executionStats")` :

Listing 17 – Mesure du cout de jointure MongoDB

```
db.employees.explain("executionStats").aggregate(  
  pipeline_jointure_3,  
  { allowDiskUse: true }  
)
```

Valeurs observées :

- **nReturned : 300024**
- **executionTimeMillis : 56400** (soit **56.4 s**)
- Stratégie indiquée : **IndexedLoopJoin** (index `emp_no_1` utilisé)

7.5 Dénormalisation avec \$project (suppression des doublons)

Objectif : produire un document employé enrichi avec deux tableaux :

- **titles** : liste des titres (avec dates)
- **salaries** : liste des salaires (avec dates)

et éviter d'embarquer des champs inutiles (doublons `emp_no`, doublons `_id` des collections importées).

Listing 18 – Pipeline de denormalisation (projection + mapping)

```
let pipeline_denormalisation = [  
  ...pipeline_jointure_3, {$project: { _id: "$emp_no", birth_date:  
    1, first_name: 1, last_name: 1,  
    gender: 1, hire_date: 1,  
    titles: {$map: {input: "$titles", as: "t", in: {title: "$$t.  
      title", from_date: "$$t.from_date",  
      to_date: "$$t.to_date"}},  
    salaries: {$map: {input: "$salaries", as: "s", in: {salary:  
      "$$s.salary",  
      from_date: "$$s.from_date", to_date: "$$s.to_date  
      }}}}}];  
  
-- J'test sur 1 employ pour vérifier  
db.employees.aggregate([{ $match: { emp_no: 10001 } }, ...  
  pipeline_denormalisation], { allowDiskUse: true }).toArray();
```

Test sur un employé (validation) :

Listing 19 – Vérification sur emp_no = 10001

```

db.employees.aggregate(
  [{ $match: { emp_no: 10001 } }, ...pipeline_denormalisation],
  { allowDiskUse: true }
).toArray();

```

Résultat observé (extrait) :

- `_id` = 10001
- `titles` contient 1 document (“Senior Engineer”, `to_date` = 9999-01-01)
- `salaries` contient 17 documents pour cet employé (dates et montants)

7.6 Sauvegarde dans une nouvelle collection (\$merge)

On sauvegarde le résultat dans `employees_denormalises`.

Mesure du temps via `console.time`.

Listing 20 – Sauvegarde avec \$merge + mesure du temps

```

console.time("lecture_complete_denormalise");
db.employees_denormalises.find({}).toArray();
console.timeEnd("lecture_complete_denormalise");

```

Temps observé :

- `creation_employees_denormalises` 27.262s s

7.7 Délai d'accès aux informations après dénormalisation

On mesure l'accès à toutes les informations d'un employé via la nouvelle collection.

Exemple : `_id` = 10001.

Listing 21 – Accès à un document dénormalisé + explain

```

db.employees_denormalises.explain("executionStats").find({ _id:
  10001 }).limit(1);

```

Valeurs observées :

- Plan : **IXSCAN** sur `_id_` (index natif)
- `executionTimeMillis` : 0
- `totalKeysExamined` : 1, `totalDocsExamined` : 1

Opération	Temps observé (ms)
Jointure MongoDB (<code>employees</code> + <code>titles</code> + <code>salaries</code>)	56 400 ms
Construction de la collection dénormalisée (\$merge)	27.262 s

TABLE 3 – Comparaison des coûts (MongoDB).

8 Partie 6 — Conclusions

Dans quel(s) cas de figure cette dénormalisation, ainsi proposée, est-elle opportune ? Et dans quel(s) cas ne l'est-elle pas ?

8.1 Quand la dénormalisation est opportune

La dénormalisation proposée (un document `employee` contenant `titles[]` et `salaries[]`) est pertinente lorsque :

- **Les accès sont centrés sur l'employé** : les requêtes typiques demandent “toutes les infos d'un employé” (identité + titres + salaires).
- **Le système est orienté lecture (read-heavy)** : on lit plus souvent qu'on ne met à jour.
- **On veut éviter le coût récurrent des jointures** : ici, la jointure MongoDB sur tous les données (`employees` + `titles` + `salaries`) a coûté **56.4 s**, alors qu'après dénormalisation l'accès par `_id` est immédiat (explain : **0 ms** observé).
- **Le “many” reste raisonnablement borné** : par exemple, pour `emp_no=10001`, on observe **1 titre** courant et **17 salaires** historisés, ce qui reste compatible avec un document unique.
- **On accepte la duplication comme compromis** : on stocke les informations au même endroit pour accélérer la lecture (coût : plus d'espace et un traitement de construction).

8.2 Quand la dénormalisation n'est pas opportune

Cette dénormalisation devient peu adaptée lorsque :

- **Les mises à jour sont fréquentes** (ex. ajout régulier de salaires/titres, corrections) : on doit maintenir des tableaux internes, ce qui complique l'écriture et augmente le risque d'incohérences si plusieurs copies existent.
- **Le côté “many” est très volumineux / non borné** : les tableaux peuvent grossir fortement et mener à des documents trop lourds.
- **On dépasse les limites de taille d'un document** : MongoDB impose une taille maximale pour un document BSON¹.
- **Les requêtes sont transverses** (ex. statistiques globales sur `salaries` ou `titles` indépendamment de l'employé) : garder des collections séparées peut rester préférable (indexation, agrégations plus directes, moindre duplication).
- **On veut conserver une normalisation forte** : contrainte d'intégrité et simplicité de maintenance côté données (logique proche d'un modèle relationnel).

1. MongoDB Docs — Limits: BSON Document Size (consulté via la documentation officielle).

Synthèse

Oui, la dénormalisation est opportune si l'on vise un accès rapide et complet aux informations *par employé*, avec un volume de sous-documents raisonnable.

Non, elle ne l'est pas si les sous-listes sont massives, très dynamiques, ou si l'on a besoin d'analyses globales fréquentes sur les entités “many” sans passer par l'employé.

Références

- PostgreSQL — JSON functions (*row_to_json, json_agg*)
- PostgreSQL — Commande *COPY*
- PostgreSQL — *CREATE MATERIALIZED VIEW*
- MongoDB — Aggregation *\$lookup*
- MongoDB — Aggregation *\$project*
- MongoDB — Aggregation *\$merge*
- MongoDB — Limits (dont taille maximale des documents *BSON*)