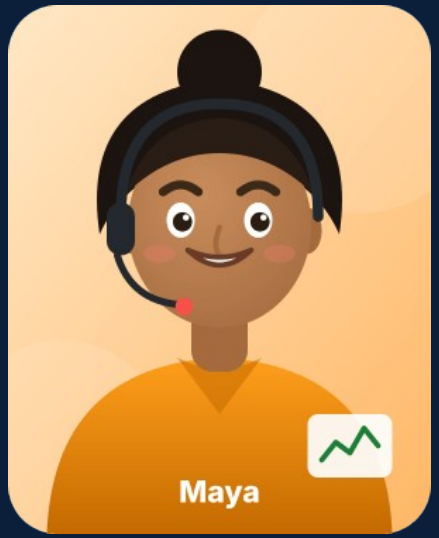
HEF-070 · MISSION 10 / 10

PostgreSQL en pratique

De la théorie à la production : types, plans réels, migrations, sauvegardes et bonnes habitudes.





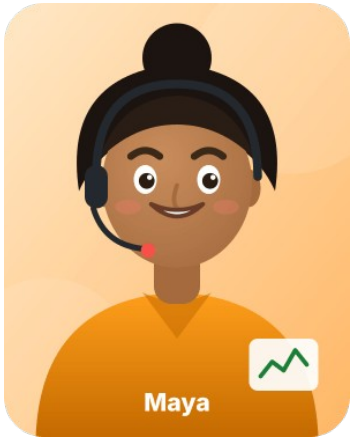
Badge : Ingénieur·e des données

95 min · intermédiaire

L'histoire

HashBox passe de SQLite à PostgreSQL : 20 000 utilisateurs simultanés, des sauvegardes automatiques, des analyses lourdes. Kodi retrouve ses repères, mais tout est plus exigeant : types stricts, droits, connexions limitées, plans d'exécution plus riches.

« Tu connais déjà les concepts, dit Sam. Il reste les habitudes de production : schéma versionné, requêtes mesurées, sauvegardes testées, accès restreints. »



Dans cette dernière mission, tu assembles le module et tu découvres les pratiques d'un vrai SGBD.

Ta mission

À la fin, je serai capable de…

1. utiliser les types PostgreSQL adaptés (NUMERIC, TIMESTAMPTZ, JSONB, UUID, identity)
2. lire un `EXPLAIN (ANALYZE)` PostgreSQL
3. écrire et appliquer des migrations versionnées
4. expliquer pooling de connexions, sauvegardes et droits
5. appliquer une checklist de mise en production

Défi d'échauffement

Défi d'échauffement

Tu emménages dans un grand entrepôt partagé. Quelles **règles** exiges-tu : qui a quelles clés, comment on documente le contenu quand tu réorganises les étagères, comment on sauvegarde l'inventaire, combien de personnes peuvent entrer en même temps ? Écris cinq règles. Lesquelles sont indispensables avant d'ouvrir ?

Le concept

Ce que PostgreSQL ajoute

PostgreSQL est un SGBD relationnel libre, très complet : MVCC, transactions robustes, types riches, index variés, extensions, réplication. Les concepts du module s'y appliquent tels quels.

Les types à bien choisir

Besoin	Type recommandé	À éviter
Identifiant	`BIGINT GENERATED ALWAYS AS IDENTITY` ou `UUID`	`SERIAL` (ancien), entiers 32 bits trop petits
Argent	`NUMERIC(12,2)` ou entier en centimes	`FLOAT` (arrondis !)
Date et heure	`TIMESTAMPTZ` (avec fuseau, stocké en UTC)	`TIMESTAMP` sans fuseau
Texte	`TEXT` (+ `CHECK` si besoin de longueur)	tailles arbitraires

Le concept

Les types à bien choisir (suite)

Besoin	Type recommandé	À éviter
Booléen	`BOOLEAN`	`0/1` en entier
Données semi-structurées	`JSONB` (indexable)	texte JSON brut
Énumérations	`CHECK (statut IN (...))` ou type `ENUM`	texte libre non contraint

```
CREATE TABLE commandes (
    id         BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    membre_id  BIGINT NOT NULL REFERENCES membres(id),
    montant    NUMERIC(12,2) NOT NULL CHECK (montant >= 0),
    statut     TEXT NOT NULL DEFAULT 'en_attente' CHECK (statut IN ('en_attente','
   payee','annulee')),
    cree_le    TIMESTAMPTZ NOT NULL DEFAULT now(),
    meta       JSONB NOT NULL DEFAULT '{}'
);
CREATE INDEX idx_commandes_membre ON commandes (membre_id, cree_le DESC);
CREATE INDEX idx_commandes_attente ON commandes (cree_le) WHERE statut = 'en_atten
```

Le concept

Les types à bien choisir (suite)

```
   te';   -- index partiel
```

Lire un plan : EXPLAIN (ANALYZE)

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM commandes WHERE membre_id = 42 ORDER BY cree_le DESC LIMIT 10;

Limit  (cost=0.43..8.9 rows=10) (actual time=0.050..0.061 rows=10 loops=1)
  ->  Index Scan using idx_commandes_membre on commandes  (cost=0.43..120.4 rows=1
   42) (actual time=0.048..0.058 rows=10 loops=1)
        Index Cond: (membre_id = 42)
Planning Time: 0.15 ms
Execution Time: 0.09 ms
```

À regarder	Signification
Seq Scan	balayage complet

Le concept

Lire un plan : EXPLAIN (ANALYZE) (suite)

À regarder	Signification
`Index Scan` / `Index Only Scan`	via l'index / index seul
`Bitmap Heap Scan`	plusieurs lignes via l'index
`rows=` estimé vs `actual rows=`	si très différents, statistiques périmées (`ANALYZE`)
`Sort`, `Hash Join`, `Nested Loop`	opérations coûteuses
`actual time`	le temps **réel**

`EXPLAIN` seul n'exécute pas la requête ; `EXPLAIN ANALYZE` l'**exécute** (attention aux `UPDATE`/`DELETE` : encadre-les d'un `BEGIN ... ROLLBACK`).

Connexions et pooling

Le concept

Connexions et pooling (suite)

Une connexion PostgreSQL est coûteuse (un processus). Avec des milliers de clients, on place un **pool de connexions** (PgBouncer, pool applicatif) : quelques dizaines de connexions réutilisées.

Les droits (rôles)

```
CREATE ROLE app_web LOGIN PASSWORD '...';
GRANT SELECT, INSERT, UPDATE ON membres, commandes TO app_web;     -- pas de DROP,
    pas de DELETE
REVOKE ALL ON SCHEMA public FROM PUBLIC;
```

Principe du moindre privilège : l'application n'est **pas** propriétaire du schéma.

Migrations

Le concept

Migrations (suite)

Le schéma évolue. Une **migration** est un petit script versionné, appliqué dans l'ordre, une seule fois. Une table `schema_migrations` mémorise les versions appliquées.

```
001_creer_membres.sql
002_ajouter_ville.sql
003_index_commandes.sql
```

Règles : jamais modifier une migration déjà appliquée ; migrations **réversibles** si possible ; changements risqués en plusieurs étapes (ajouter la colonne, remplir, rendre obligatoire) pour ne pas bloquer la production.

Sauvegardes

Méthode	Usage
`pg_dump`	sauvegarde logique d'une base (SQL), simple

Le concept

Sauvegardes (suite)

Méthode	Usage
sauvegarde physique + WAL archivés	restauration à un instant précis (PITR)
Test de restauration	une sauvegarde non testée n'existe pas

Observer

- `pg_stat_statements` : requêtes les plus coûteuses.
- Journal des requêtes lentes (`log_min_duration_statement`).
- `pg_stat_activity` : qui fait quoi maintenant, requêtes bloquées.
- `VACUUM` / autovacuum : nettoyage des anciennes versions (MVCC).

Le concept

Une checklist de mise en production

1. Schéma versionné par migrations.
2. Clés étrangères, contraintes, types stricts.
3. Index justifiés par des requêtes réelles ; plans lus.
4. Droits minimaux ; mots de passe hors du code.
5. Pool de connexions ; délais d'attente (`statement_timeout`).
6. Sauvegardes automatiques **et restauration testée**.
7. Supervision : requêtes lentes, espace disque, connexions, retard de réplication.
8. Procédure de retour arrière documentée.

Exemple pas à pas

Problème : écrire un petit gestionnaire de migrations (testé sur SQLite, la logique est identique pour PostgreSQL).

```
import sqlite3

MIGRATIONS = [
    (1, "creer_membres", "CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT N
   OT NULL)"),
    (2, "ajouter_ville", "ALTER TABLE membres ADD COLUMN ville TEXT"),
    (3, "creer_index_ville", "CREATE INDEX idx_membres_ville ON membres(ville)"),
]

def appliquer(bd, migrations):
    bd.execute("CREATE TABLE IF NOT EXISTS schema_migrations (version INTEGER PRIM
   ARY KEY, nom TEXT NOT NULL, appliquee_le TEXT NOT NULL DEFAULT CURRENT_TIMESTAM
   P)")
    deja = {v for (v,) in bd.execute("SELECT version FROM schema_migrations")}
    appliquees = []
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    for version, nom, sql in sorted(migrations):
        if version in deja:
            continue
        with bd:                                          # chaque migration est
   atomique
            bd.execute(sql)
            bd.execute("INSERT INTO schema_migrations (version, nom) VALUES (?, ?)
   ", (version, nom))
        appliquees.append(version)
    return appliquees

bd = sqlite3.connect(":memory:")
print(appliquer(bd, MIGRATIONS))          # [1, 2, 3]
print(appliquer(bd, MIGRATIONS))          # [] : idempotent, rien à refaire
print(appliquer(bd, MIGRATIONS + [(4, "ajouter_score", "ALTER TABLE membres ADD CO
   LUMN score INTEGER NOT NULL DEFAULT 0")]))   # [4]
print([c[1] for c in bd.execute("PRAGMA table_info(membres)")])    # ['id', 'nom',
    'ville', 'score']
```

Exemple pas à pas (suite)

Lecture : le gestionnaire est **idempotent** (relancé, il ne refait rien), **ordonné**, et **atomique** par migration : si une migration échoue, sa version n'est pas enregistrée et rien n'est à moitié appliqué.
Pourquoi SERIAL → IDENTITY, FLOAT → NUMERIC ?

```
from decimal import Decimal

print(sum([0.1] * 10) == 1.0)                          # False : les flottants se tro
   mpent
print(sum([Decimal("0.1")] * 10) == Decimal("1.0"))  # True : NUMERIC / Decimal so
   nt exacts
```

Un `EXPLAIN` à lire : si l'estimation dit `rows=10` mais que la réalité est `actual rows=500000`, le planificateur a choisi un plan adapté à 10 lignes. La cause fréquente : statistiques périmées → `ANALYZE membres;`.

À toi, avec aide

À toi, avec aide

Voici un schéma écrit « à la va-vite » :

```
CREATE TABLE paiements (
    id SERIAL,
    client VARCHAR(10),
    montant FLOAT,
    date TIMESTAMP,
    statut VARCHAR(255)
);
```

1. Liste au moins 6 problèmes.
2. Réécris le schéma pour PostgreSQL de façon professionnelle.
3. Propose deux index utiles et un droit minimal pour un rôle applicatif.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Il manque la clé primaire et la clé étrangère vers les clients.

Indice 2

Argent en `FLOAT` : arrondis.

Indice 3

`TIMESTAMP` sans fuseau, `VARCHAR(10)` arbitraire, statut libre.

Réponse

Problèmes : pas de clé primaire déclarée ; `SERIAL` ancien (préférer `IDENTITY`) ; `client` en texte sans clé étrangère ni `NOT NULL` ; `FLOAT` pour l'argent ; `TIMESTAMP` sans fuseau ; `statut` libre sans contrainte ; aucune valeur par défaut ni date de création ; tailles arbitraires.

```
CREATE TABLE paiements (
    id         BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    client_id  BIGINT NOT NULL REFERENCES clients(id),
    montant    NUMERIC(12,2) NOT NULL CHECK (montant > 0),
    paye_le    TIMESTAMPTZ NOT NULL DEFAULT now(),
    statut     TEXT NOT NULL DEFAULT 'en_attente' CHECK (statut IN ('en_attente','
   accepte','refuse','rembourse'))
);
CREATE INDEX idx_paiements_client ON paiements (client_id, paye_le DESC);
CREATE INDEX idx_paiements_attente ON paiements (paye_le) WHERE statut = 'en_atten
   te';
GRANT SELECT, INSERT, UPDATE ON paiements TO app_web;       -- pas de DELETE ni DR
   OP
```

À toi, en solo

Migrations réversibles. Améliore le gestionnaire pour qu'il gère `monter` et `descendre` :

- chaque migration a un SQL `up` et un SQL `down` ;
- `migrer(bd, version_cible)` monte ou descend jusqu'à la version demandée ;
- une migration en échec n'enregistre rien et arrête le processus ;
- vérifie sur 4 migrations : monter à 4, descendre à 2, remonter à 4.

(SQLite ≥ 3.35 permet `ALTER TABLE ... DROP COLUMN`.)

À toi, en solo · réponse

Pistes de correction

```
import sqlite3

MIGRATIONS = {
    1: ("CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT NOT NULL)", "DROP
   TABLE membres"),
    2: ("ALTER TABLE membres ADD COLUMN ville TEXT", "ALTER TABLE membres DROP COL
   UMN ville"),
    3: ("CREATE INDEX idx_membres_nom ON membres(nom)", "DROP INDEX idx_membres_no
   m"),
    4: ("ALTER TABLE membres ADD COLUMN score INTEGER NOT NULL DEFAULT 0", "ALTER
   TABLE membres DROP COLUMN score"),
}

def version_courante(bd):
    bd.execute("CREATE TABLE IF NOT EXISTS schema_migrations (version INTEGER PRIM
   ARY KEY)")
    return bd.execute("SELECT COALESCE(MAX(version), 0) FROM schema_migrations").f
   etchone()[0]
```

Pistes de correction (suite)

```
def migrer(bd, cible):
    courante = version_courante(bd)
    if cible > courante:
        etapes = [(v, MIGRATIONS[v][0], "up") for v in range(courante + 1, cible +
    1)]
    else:
        etapes = [(v, MIGRATIONS[v][1], "down") for v in range(courante, cible, -1
   )]
    for version, sql, sens in etapes:
        with bd:                                    # atomique : échec = rien n'es
   t enregistré
            bd.execute(sql)
            if sens == "up":
                bd.execute("INSERT INTO schema_migrations VALUES (?)", (version,))
            else:
                bd.execute("DELETE FROM schema_migrations WHERE version = ?", (ver
   sion,))
    return version_courante(bd)
```

Pistes de correction (suite)

```
colonnes = lambda bd: [c[1] for c in bd.execute("PRAGMA table_info(membres)")]
bd = sqlite3.connect(":memory:")
print(migrer(bd, 4), colonnes(bd))        # 4 ['id', 'nom', 'ville', 'score']
print(migrer(bd, 2), colonnes(bd))        # 2 ['id', 'nom', 'ville']
print(migrer(bd, 4), colonnes(bd))        # 4 ['id', 'nom', 'ville', 'score']
```

En production, les migrations « down » sont un filet de sécurité ; on privilégie souvent les migrations **additives** (ajouter plutôt que supprimer) pour pouvoir revenir en arrière sans perdre de données.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
`FLOAT` pour l'argent	Erreurs d'arrondi	`NUMERIC` ou entier en centimes
Modifier le schéma à la main en production	Environnements qui divergent	Migrations versionnées
Sauvegarde jamais restaurée	On découvre le problème le jour du sinistre	Tester la restauration régulièrement
Application connectée en superutilisateur	Une faille donne tous les droits	Rôle applicatif aux droits minimaux

Mini-boss

Dossier de mise en production d'une base. Pour le service de réservation de salles du module (ou pour une application de ton choix) :

1. Écris le schéma cible pour PostgreSQL (types adaptés, contraintes, index justifiés, rôle applicatif avec droits minimaux).
2. Écris le gestionnaire de migrations (version montante/descendante) et quatre migrations réalistes, dont une qui ajoute une colonne obligatoire en trois étapes.
3. Teste le tout sur SQLite avec des tests automatiques (appliquer, rejouer, descendre, échec d'une migration).
4. Rédige : plan de sauvegarde et de **test de restauration**, supervision (4 indicateurs), procédure de retour arrière.
5. Justifie 5 choix par une notion du module (ACID, index, normalisation, isolation, contraintes).

Mini-boss · réponse

Pistes de correction

Un bon dossier : le schéma PostgreSQL utilise `IDENTITY`, `TIMESTAMPTZ`, `NUMERIC`, `CHECK`, une contrainte d'exclusion pour le chevauchement (`EXCLUDE USING gist (salle_id WITH =, tstzrange(debut, fin) WITH &&)`), des index partiels, un rôle `app_reservations` avec `SELECT/INSERT/UPDATE` seulement. La migration en trois étapes : (1) ajouter la colonne nullable, (2) remplir par lots, (3) ajouter `NOT NULL`. Les tests couvrent : première application, idempotence, descente, et une migration défaillante qui ne laisse aucune trace. Le plan de sauvegarde prévoit `pg_dump` quotidien, archivage WAL, et un test de restauration mensuel chronométré ; supervision : requêtes lentes, espace disque, connexions utilisées, retard de réplication ; retour arrière : migrations additives + sauvegarde avant migration. Les justifications renvoient à ACID (transaction par migration), index (partiel), normalisation (3FN), isolation (réessai sur conflit), contraintes (exclusion).

Quiz éclair

Quiz éclair

1. Pour stocker de l'argent, on préfère :

A. NUMERIC ou un entier en centimes
B. FLOAT
C. TEXT

2. Une migration est :

A. une copie de la base
B. un script de changement de schéma versionné
C. un type d'index

3. `EXPLAIN ANALYZE` :

A. affiche seulement un schéma

Quiz éclair

Quiz éclair (suite)

B. supprime les index
C. exécute la requête et donne les temps réels

4. 🔁 *Révision mission 09* : Une mise à jour perdue se corrige avec :

A. une opération atomique ou un verrouillage optimiste
B. un redémarrage
C. un index

5. 🔁 *Révision mission 09* : MVCC permet :

A. de supprimer les index
B. aux lecteurs de ne pas bloquer les écrivains
C. de chiffrer la base

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

1. Quelle habitude de production (sauvegarde testée, accès minimaux, changements versionnés) adopterais-tu dès demain dans tes propres projets ?
2. Qu'est-ce qui t'a le plus surpris dans la gestion de données sérieuses ?

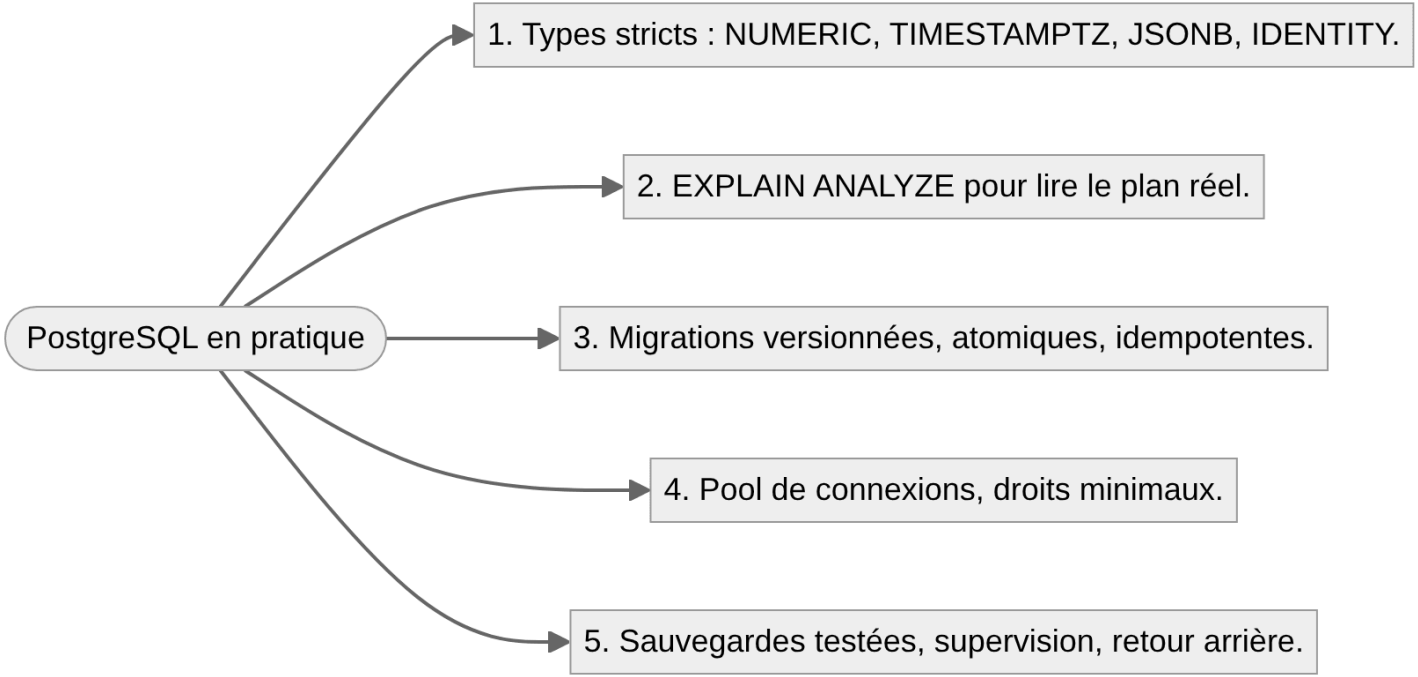
À retenir

À retenir

1. Types stricts : NUMERIC, TIMESTAMPTZ, JSONB, IDENTITY.
2. EXPLAIN (ANALYZE) pour lire le plan réel.
3. Migrations versionnées, atomiques, idempotentes.
4. Pool de connexions, droits minimaux.
5. Sauvegardes testées, supervision, retour arrière.

À retenir

Le schéma de la mission



Félicitations

Tu as terminé **HEF-070 — Bases de données** et le **bloc Systèmes**. Tu sais modéliser et normaliser, écrire du SQL de lecture et d'écriture, imposer des contraintes, joindre et agréger, lire un plan et choisir des index, raisonner sur les transactions et la concurrence, et préparer une base pour la production.
→ Prochain bloc : **Ingénierie** — commence par **HEF-080 — Génie logiciel**

MISSION ACCOMPLIE

+120 XP

Badge débloqué : Ingénieur·e des données

Fin du module HEF-070. Bravo !

