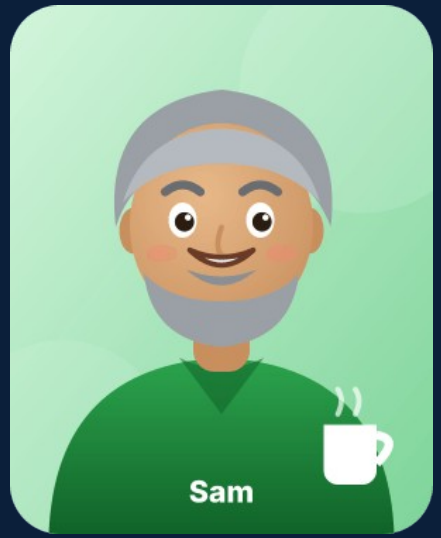
HEF-130 · MISSION 07 / 10

Bases de données et performance

Dans la plupart des applications, le goulot est la base de données. Index, plans d'exécution, requêtes en lot : quelques notions suffisent à transformer des secondes en millisecondes.





Badge : Dresseur de requêtes

105 min · avancé

L'histoire

La page d'administration de HashBox affiche la liste des inscriptions du mois. Au lancement, 400 lignes : instantané. Un an plus tard, 2 millions de lignes : 11 secondes. Aucun code n'a changé. Pire : pour chaque ligne, l'affichage lance une requête supplémentaire pour récupérer le nom du membre.

« Deux problèmes classiques, dit Sam. Le premier : la base lit toute la table pour trouver quelques lignes, faute d'index. Le deuxième : le "N+1", une requête par ligne au lieu d'une seule jointure. La base te dit elle-même comment elle exécute une requête, il suffit de lui demander. »



Tu apprends à lire un plan d'exécution, à poser les bons index (y compris composites et couvrants), à repérer et supprimer le N+1, à paginer correctement et à grouper les écritures.

Ta mission

À la fin, je serai capable de…

1. lire un plan d'exécution (`EXPLAIN QUERY PLAN`) et distinguer parcours complet et recherche par index
2. concevoir un index (simple, composite, couvrant) selon la requête
3. détecter et corriger le problème N+1
4. paginer efficacement (clé de pagination plutôt que décalage)
5. grouper les écritures en transactions et mesurer le gain

Défi d'échauffement

Défi d'échauffement

Une bibliothèque de 2 millions de livres. On te demande tous les livres de « Hugo ». Sans catalogue, tu parcours les rayons un par un. Avec un fichier par auteur, tu vas droit au bon endroit. Quel prix paie-t-on pour tenir ce fichier à jour à chaque nouveau livre ? Dans quels cas l'index ne servira pas (par exemple « tous les livres dont le titre contient *mer* ») ?

Le concept

Comment la base exécute une requête

Une requête SQL décrit **ce que tu veux**. L'**optimiseur** choisit **comment** : quel index, quel ordre de jointure. Le **plan d'exécution** montre ce choix :

```
EXPLAIN QUERY PLAN SELECT * FROM inscriptions WHERE membre_id = 42;
→ SCAN inscriptions                                  (parcours complet : O(n))
→ SEARCH inscriptions USING INDEX idx_membre (membre_id=?)   (recherche : O(log n)
   )
```

Premier réflexe devant une requête lente : **lire son plan** (`EXPLAIN` / `EXPLAIN ANALYZE` / `EXPLAIN QUERY PLAN` selon la base).

Le concept

Les index

Un **index** est une structure triée (arbre B) qui permet de retrouver des lignes en O(log n). Il a un **coût** : espace disque, et chaque écriture doit aussi le mettre à jour.

Type	Quand
Simple (`membre_id`)	filtre ou jointure sur une colonne
Composite (`membre_id, date`)	filtre sur plusieurs colonnes
Couvrant : contient toutes les colonnes de la requête	la table n'est même plus lue
Unique	contrainte et recherche
Partiel	seulement certaines lignes (`WHERE actif = 1`)

Le concept

Les index (suite)

Règle des colonnes d'un index composite : d'abord les colonnes testées par **égalité**, ensuite celle testée par **intervalle ou tri**. L'index `(membre_id, date)` sert `WHERE membre_id = ? AND date > ?` et `WHERE membre_id = ?`, mais **pas** `WHERE date > ?` seul (préfixe gauche).

Ce qui empêche d'utiliser un index

- fonction sur la colonne : `WHERE lower(nom) = 'ada'` ;
- recherche par motif commençant par un joker : `LIKE '%mer'` ;
- conversion implicite de type ;
- faible **sélectivité** (colonne booléenne : l'index n'aide guère) ;
- statistiques périmées.

Le concept

Le problème N+1

```
1 requête : SELECT * FROM inscriptions LIMIT 100      → 100 lignes
puis 100 requêtes : SELECT nom FROM membres WHERE id = ?   (une par ligne)
```

= **101 requêtes** (1 + N) au lieu d'**1 jointure**. Chaque requête a un coût fixe (réseau, analyse). Solution : **jointure** (`JOIN`), ou requête `IN (...)` en lot, ou chargement groupé (mission 6).

Pagination

Méthode	Principe	Problème
Décalage `LIMIT 50 OFFSET 100000`	saute 100 000 lignes	la base doit les parcourir : plus on va loin, plus c'est lent

Le concept

Pagination (suite)

Méthode	Principe	Problème
Clé de pagination (*keyset*) `WHERE id > :dernier ORDER BY id LIMIT 50`	repart de la dernière valeur vue, grâce à l'index	rapide quelle que soit la page

Écritures : les transactions

Chaque `INSERT` en **autocommit** déclenche un `COMMIT` (écriture durable, `fsync`). 10 000 insertions = 10 000 commits. En les groupant dans **une transaction** (ou par lots de 1 000), on divise le coût par des dizaines.

Autres leviers

Levier	Effet
Requêtes **préparées** / paramétrées	analyse réutilisée, sécurité
`SELECT` des **colonnes utiles** (pas *)	moins de données

Le concept

Autres leviers (suite)

Levier	Effet
Dénormalisation ciblée (compteur précalculé)	évite des agrégats coûteux
Vues matérialisées / tables de synthèse	résultats précalculés
Réplicas en lecture	répartir la lecture
Partitionnement	tables énormes
Pool de connexions	évite d'ouvrir une connexion par requête

Méthode

1. repérer les requêtes lentes (journal des requêtes lentes, `pg_stat_statements`, etc.) ;
2. lire le plan ;
3. corriger (index, réécriture, lot) ;

Le concept

Méthode (suite)

4. **re-mesurer** avec des données **réalistes en volume** ;
5. vérifier que les **écritures** ne sont pas trop ralenties.

🧠 ***Astuce*** *: une requête rapide sur 1 000 lignes peut être catastrophique sur 10 millions. Teste toujours avec un volume proche de la production.*

Exemple pas à pas

Problème : avec SQLite, lire des plans d'exécution avant/après index, compter les requêtes d'un N+1 et d'une jointure, comparer les paginations, et mesurer le gain des transactions.

```
import os, random, sqlite3, tempfile, time

def temps(f, n=3):
    meilleur = float("inf")
    for _ in range(n):
        d = time.perf_counter(); f(); meilleur = min(meilleur, time.perf_counter()
    - d)
    return meilleur

def plan(bd, requete, params=()):
    return " | ".join(r[3] for r in bd.execute("EXPLAIN QUERY PLAN " + requete, pa
   rams))

# ---------------- données ----------------
bd = sqlite3.connect(":memory:")
```

Exemple pas à pas (suite)

```
bd.executescript('''
CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT);
CREATE TABLE inscriptions (id INTEGER PRIMARY KEY, membre_id INTEGER, cours TEXT,
   jour INTEGER, montant INTEGER);
''')
rng = random.Random(1)
bd.executemany("INSERT INTO membres VALUES (?, ?)", ((i, f"membre-{i}") for i in r
   ange(1, 5001)))
bd.executemany("INSERT INTO inscriptions (membre_id, cours, jour, montant) VALUES
   (?,?,?,?)",
               ((rng.randint(1, 5000), f"HEF-{rng.randint(0, 89):03d}", rng.randin
   t(1, 365), rng.randint(10, 200)) for _ in range(300000)))
bd.commit()

# ---------------- 1. sans index : parcours complet ----------------
Q = "SELECT SUM(montant) FROM inscriptions WHERE membre_id = ?"
p_avant = plan(bd, Q, (42,))
t_avant = temps(lambda: [bd.execute(Q, (m,)).fetchone() for m in range(1, 41)])
bd.execute("CREATE INDEX idx_membre ON inscriptions (membre_id)")
```

Exemple pas à pas (suite)

```
p_apres = plan(bd, Q, (42,))
t_apres = temps(lambda: [bd.execute(Q, (m,)).fetchone() for m in range(1, 41)])
print("avant :", p_avant); print("après :", p_apres)
print(f"40 requêtes : {t_avant * 1000:8.1f} ms → {t_apres * 1000:6.2f} ms (x{t_ava
   nt / t_apres:.0f})")
assert "SCAN" in p_avant and "SEARCH" in p_apres and "idx_membre" in p_apres
assert t_avant > 20 * t_apres

# ---------------- 2. index composite : l'ordre des colonnes compte --------------
   --
Q2 = "SELECT montant FROM inscriptions WHERE membre_id = ? AND jour BETWEEN ? AND
   ?"
bd.execute("CREATE INDEX idx_membre_jour ON inscriptions (membre_id, jour, montant
   )")      # égalité, puis intervalle, puis colonne lue (couvrant)
p = plan(bd, Q2, (42, 10, 100))
print("composite couvrant :", p)
assert "COVERING INDEX idx_membre_jour" in p
p_seul = plan(bd, "SELECT montant FROM inscriptions WHERE jour BETWEEN ? AND ?", (
   10, 100))
```

Exemple pas à pas (suite)

```
print("filtre sur « jour » seul (préfixe absent) :", p_seul)
assert "idx_membre_jour" not in p_seul or "SCAN" in p_seul

# ---------------- 3. fonction sur la colonne : l'index est ignoré ---------------
   -
bd.execute("CREATE INDEX idx_cours ON inscriptions (cours)")
p_ok = plan(bd, "SELECT COUNT(*) FROM inscriptions WHERE cours = 'HEF-010'")
p_ko = plan(bd, "SELECT COUNT(*) FROM inscriptions WHERE lower(cours) = 'hef-010'"
   )
print("égalité simple :", p_ok, "| avec lower() :", p_ko)
assert "SEARCH" in p_ok and "SCAN" in p_ko or "COVERING" in p_ko and "SCAN" not in
    p_ok

# ---------------- 4. N+1 contre jointure : on compte les requêtes ---------------
   -
requetes = []
bd.set_trace_callback(lambda sql: requetes.append(sql))
lignes = bd.execute("SELECT id, membre_id FROM inscriptions LIMIT 100").fetchall()
noms = [bd.execute("SELECT nom FROM membres WHERE id = ?", (m,)).fetchone()[0] for
```

Exemple pas à pas (suite)

```
    _, m in lignes]       # N+1
n_plus_1 = len(requetes)
requetes.clear()
joint = bd.execute("SELECT i.id, m.nom FROM inscriptions i JOIN membres m ON m.id
   = i.membre_id LIMIT 100").fetchall()
une_jointure = len(requetes)
bd.set_trace_callback(None)
print(f"N+1 : {n_plus_1} requêtes | jointure : {une_jointure} requête")
assert n_plus_1 == 101 and une_jointure == 1 and [n for _, n in joint] == noms

# ---------------- 5. pagination : décalage contre clé ----------------
def page_offset(decalage): return bd.execute("SELECT id, montant FROM inscriptions
    ORDER BY id LIMIT 50 OFFSET ?", (decalage,)).fetchall()
def page_cle(dernier_id):  return bd.execute("SELECT id, montant FROM inscriptions
    WHERE id > ? ORDER BY id LIMIT 50", (dernier_id,)).fetchall()
assert page_offset(250000) == page_cle(250000)                          # même pag
   e (les identifiants sont 1..300000)
t_off, t_cle = temps(lambda: page_offset(250000), 5), temps(lambda: page_cle(25000
   0), 5)
```

Exemple pas à pas (suite)

```
print(f"page 5000 : décalage {t_off * 1000:7.2f} ms | clé {t_cle * 1000:6.3f} ms (
   x{t_off / t_cle:.0f})")
assert t_off > 5 * t_cle

# ---------------- 6. écritures : autocommit contre transaction (base sur fichier)
    ----------------
with tempfile.TemporaryDirectory() as d:
    def ecrire(autocommit_par_ligne, n=400):
        chemin = os.path.join(d, f"b{autocommit_par_ligne}.db")
        c = sqlite3.connect(chemin); c.execute("CREATE TABLE t (x INTEGER)")
        c.commit()
        debut = time.perf_counter()
        if autocommit_par_ligne:
            for i in range(n):
                c.execute("INSERT INTO t VALUES (?)", (i,)); c.commit()
   # un COMMIT par ligne
        else:
            c.executemany("INSERT INTO t VALUES (?)", ((i,) for i in range(n))); c
   .commit()   # un seul COMMIT
```

Exemple pas à pas (suite)

```
        duree = time.perf_counter() - debut
        compte = c.execute("SELECT COUNT(*) FROM t").fetchone()[0]; c.close()
        return duree, compte
    t1, c1 = ecrire(True); t2, c2 = ecrire(False)
    print(f"400 insertions : un commit par ligne {t1 * 1000:7.1f} ms | un seul com
mit {t2 * 1000:6.2f} ms (x{t1 / t2:.0f})")
    assert c1 == c2 == 400 and t1 > 3 * t2
```

Lecture :

1. **Plan** : avant l'index, `SCAN inscriptions` (parcours des 300 000 lignes pour chaque requête) ; après, `SEARCH ... USING INDEX`. Le gain est de plusieurs ordres de grandeur sur 40 requêtes.
2. L'index **composite couvrant** `(membre_id, jour, montant)` sert la requête sans toucher la table : `COVERING INDEX`. Mais un filtre sur `jour` **seul** ne peut pas l'utiliser efficacement (préfixe gauche manquant).
3. Une fonction sur la colonne (`lower(cours)`) **désactive** l'index ; l'égalité simple l'utilise.

Exemple pas à pas (suite)

4. Le **N+1** coûte 101 requêtes pour 100 lignes ; la jointure en coûte 1, avec les **mêmes** résultats.
5. La pagination par **clé** reste rapide à la page 5000, alors que le **décalage** doit parcourir 250 000 lignes.
6. Grouper les insertions dans **une transaction** supprime les 400 commits durables.

À toi, avec aide

On a la table `commandes(id, client_id, statut, date, total)` avec 5 millions de lignes. Voici trois requêtes fréquentes :

1. `SELECT * FROM commandes WHERE client_id = ? AND statut = 'payee' ORDER BY date DESC LIMIT 20`
2. `SELECT COUNT(*) FROM commandes WHERE date >= ? AND date < ?`
3. `SELECT client_id, SUM(total) FROM commandes WHERE statut = 'payee' GROUP BY client_id`

Pour chacune : quel index proposer (colonnes dans l'ordre, et pourquoi) ? Vérifie les plans avec SQLite sur 100 000 lignes générées. Quel coût les index imposent-ils sur les écritures, et lequel garderais-tu si tu devais en limiter le nombre ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Requête 1 : égalité sur `client_id` et `statut`, tri sur `date`.

Indice 2

Requête 2 : un intervalle sur `date` seul.

Indice 3

Requête 3 : un index couvrant `(statut, client_id, total)` évite de lire la table.

À toi, avec aide · réponse

Réponse

```
import random, sqlite3
bd = sqlite3.connect(":memory:")
bd.execute("CREATE TABLE commandes (id INTEGER PRIMARY KEY, client_id INT, statut
   TEXT, date INT, total INT)")
rng = random.Random(1)
bd.executemany("INSERT INTO commandes (client_id, statut, date, total) VALUES (?,?
   ,?,?)",
               ((rng.randint(1, 3000), rng.choice(["payee", "annulee", "en_cours"]
   ), rng.randint(1, 1000), rng.randint(5, 500)) for _ in range(100000)))
def plan(q, p=()): return " | ".join(r[3] for r in bd.execute("EXPLAIN QUERY PLAN
   " + q, p))
Q1 = "SELECT * FROM commandes WHERE client_id = ? AND statut = 'payee' ORDER BY da
   te DESC LIMIT 20"
Q2 = "SELECT COUNT(*) FROM commandes WHERE date >= ? AND date < ?"
Q3 = "SELECT client_id, SUM(total) FROM commandes WHERE statut = 'payee' GROUP BY
   client_id"
bd.execute("CREATE INDEX i1 ON commandes (client_id, statut, date)")
bd.execute("CREATE INDEX i2 ON commandes (date)")
bd.execute("CREATE INDEX i3 ON commandes (statut, client_id, total)")
```

À toi, avec aide · réponse

Réponse (suite)

```
for nom, q, p in (("Q1", Q1, (42,)), ("Q2", Q2, (100, 200)), ("Q3", Q3, ())):
    print(nom, "→", plan(q, p))
assert "i1" in plan(Q1, (42,)) and "i2" in plan(Q2, (100, 200)) and "COVERING INDE
   X i3" in plan(Q3)
```

1. `(client_id, statut, date)` : égalités d'abord, puis la colonne de **tri** — la base lit directement les 20 dernières lignes sans trier.
2. `(date)` : un intervalle unique.
3. `(statut, client_id, total)` couvrant : la table n'est pas lue, le groupement s'appuie sur l'ordre de l'index.

Chaque index ralentit les `INSERT`/`UPDATE` (mise à jour de l'arbre) et consomme de l'espace. Si je ne devais en garder qu'un : celui de la requête **la plus fréquente et la plus coûteuse** — a priori la 1 si elle sert chaque page client. Le choix se fait en mesurant la fréquence et le coût réels (journal des requêtes lentes), pas à l'intuition.

À toi, en solo

Détecteur de N+1. Écris un petit outil qui instrumente une connexion SQLite et détecte les motifs de N+1 :

1. un enveloppeur qui enregistre chaque requête avec ses paramètres (via `set_trace_callback`) ;
2. normalise les requêtes (remplace les valeurs littérales et paramètres par `?`) et compte les occurrences de chaque forme ;
3. signale comme suspecte une forme exécutée plus de 20 fois dans une même « unité de travail » (un bloc `with`) ;
4. propose une correction en texte (JOIN ou `IN (...)`) ;
5. teste sur une fonction `afficher_inscriptions()` N+1 et sur sa version corrigée ;
6. ajoute un test automatique qui échoue si une unité de travail dépasse un budget de requêtes (utile en CI).

À toi, en solo · réponse

Pistes de correction

```
import re, sqlite3
from collections import Counter

class Observateur:
    def __init__(self, bd, budget=None):
        self.bd, self.budget, self.requetes = bd, budget, []
    def __enter__(self):
        self.requetes = []
        self.bd.set_trace_callback(self.requetes.append); return self
    def __exit__(self, type_, valeur, trace):
        self.bd.set_trace_callback(None)
        if type_ is None and self.budget is not None and len(self.requetes) > self
   .budget:
            raise AssertionError(f"budget de requêtes dépassé : {len(self.requetes
   )} > {self.budget}")
    @staticmethod
    def forme(sql):
        sql = re.sub(r"'[^']*'", "?", sql); sql = re.sub(r"\b\d+\b", "?", sql)
        return re.sub(r"\s+", " ", sql).strip()
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def suspectes(self, seuil=20):
        return {f: n for f, n in Counter(self.forme(r) for r in self.requetes).ite
   ms() if n > seuil}

bd = sqlite3.connect(":memory:")
bd.executescript("CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT); CREATE
   TABLE insc (id INTEGER PRIMARY KEY, membre_id INT, cours TEXT);")
bd.executemany("INSERT INTO membres VALUES (?, ?)", ((i, f"m{i}") for i in range(1
   , 201)))
bd.executemany("INSERT INTO insc (membre_id, cours) VALUES (?, ?)", ((i % 200 + 1,
    f"C{i % 9}") for i in range(500)))

def afficher_n_plus_1():
    return [(i, bd.execute("SELECT nom FROM membres WHERE id = ?", (m,)).fetchone(
   )[0], c) for i, m, c in bd.execute("SELECT id, membre_id, cours FROM insc LIMIT
    100").fetchall()]

def afficher_jointure():
    return bd.execute("SELECT i.id, m.nom, i.cours FROM insc i JOIN membres m ON m
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    .id = i.membre_id LIMIT 100").fetchall()

with Observateur(bd) as obs:
    r1 = afficher_n_plus_1()
print("suspectes :", obs.suspectes())
assert list(obs.suspectes().values()) == [100] and "SELECT nom FROM membres WHERE
   id = ?" in obs.suspectes()
with Observateur(bd) as obs2:
    r2 = afficher_jointure()
assert obs2.suspectes() == {} and len(obs2.requetes) == 1 and sorted(r1) == sorted
   (r2)

try:
    with Observateur(bd, budget=5):
        afficher_n_plus_1()
    raise SystemExit("le budget aurait dû être dépassé")
except AssertionError as e:
    print("garde-fou CI :", e)
with Observateur(bd, budget=5):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    afficher_jointure()
print("correction suggérée : remplacer la boucle par un JOIN ou par un SELECT ...
   WHERE id IN (...) unique")
```

Compter les requêtes par **unité de travail** est un test **déterministe** (contrairement au chronométrage) : il attrape les N+1 dès qu'ils sont introduits, même avec très peu de données de test.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Aucun index sur les colonnes filtrées/jointes	Parcours complet de grandes tables	Lire le plan, indexer selon la requête
Tester sur 100 lignes	Tout est rapide, jusqu'à la production	Volumes réalistes
N+1	Des centaines de requêtes inutiles	JOIN, IN (...), chargement groupé
Pagination par décalage profond	Plus on avance, plus c'est lent	Pagination par clé

Mini-boss

Mini-boss

Rapport d'administration lent. On te donne (ou tu génères) une base de 1 million d'inscriptions, 50 000 membres, 100 cours, et un rapport qui exécute 5 requêtes lentes : (a) inscriptions d'un membre dans un intervalle, (b) total par cours du mois, (c) liste paginée avec nom du membre (N+1 + décalage), (d) membres sans inscription depuis 6 mois (sous-requête corrélée), (e) import quotidien de 100 000 lignes en autocommit.

1. mesure chaque requête et lis son plan ;
2. corrige par index, réécriture, clés de pagination, jointures, lots ;
3. vérifie à chaque étape l'**équivalence des résultats** ;
4. tableau avant/après (temps, lignes lues, requêtes) ;
5. mesure le **coût sur les écritures** des index ajoutés ;
6. écris les garde-fous (budget de requêtes, test de plan) pour la CI.

Mini-boss · réponse

Pistes de correction

Un bon travail : tableau avant/après avec des gains de l'ordre de ×50 à ×1000 selon la requête; réécriture de la sous-requête corrélée en `NOT EXISTS`/`LEFT JOIN ... IS NULL` avec index `(membre_id, jour)`; pagination par clé `(jour, id)`; import en transaction par lots de 10 000; mesure du ralentissement des insertions avec 3 index (quelques dizaines de pourcents) et décision argumentée sur les index à conserver; garde-fous : test de plan (`SEARCH` attendu) et budget de requêtes.

Quiz éclair

Quiz éclair

1. Un `SCAN` dans un plan d'exécution signifie :

A. un parcours complet de la table
B. une recherche par index
C. une erreur de syntaxe

2. Le problème N+1 correspond à :

A. une table avec N+1 colonnes
B. une requête pour la liste puis une requête par élément
C. un index en trop

3. Dans un index composite, on place d'abord :

A. la colonne la plus longue

Quiz éclair

Quiz éclair (suite)

B. les colonnes au hasard

C. les colonnes testées par égalité, puis celle d'intervalle ou de tri

4. 🔁 *Révision mission 06* : Pour de petites requêtes lointaines, ce qui domine le temps est :

A. la latence (nombre d'allers-retours)

B. la bande passante

C. la couleur de la page

5. 🔁 *Révision mission 06* : Le keep-alive permet de :

A. chiffrer les échanges

B. réutiliser une connexion pour plusieurs requêtes

C. compresser les images

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle requête de ton application s'exécute le plus souvent, et connais-tu son plan ?
2. Quels index existent chez toi sans que personne sache plus pourquoi ?

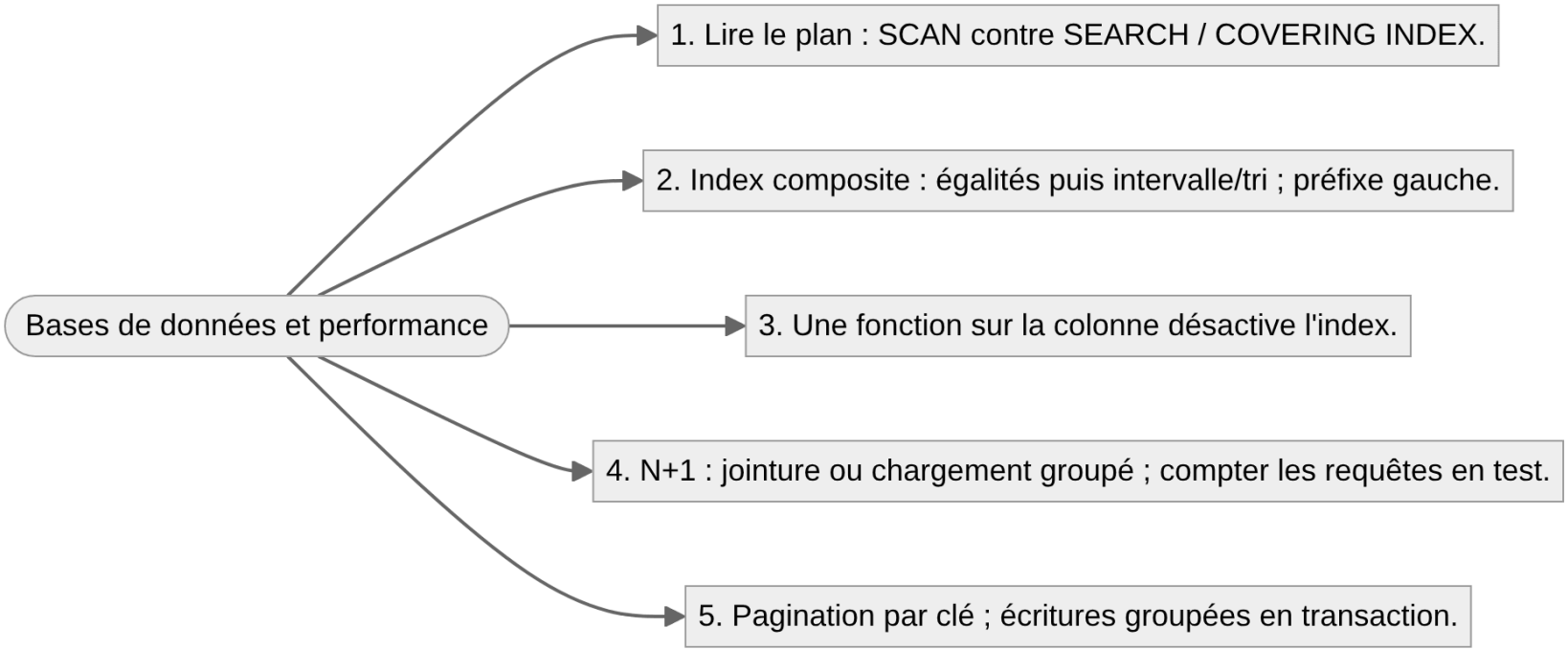
À retenir

À retenir

1. Lire le plan : SCAN contre SEARCH / COVERING INDEX.
2. Index composite : égalités puis intervalle/tri ; préfixe gauche.
3. Une fonction sur la colonne désactive l'index.
4. N+1 : jointure ou chargement groupé ; compter les requêtes en test.
5. Pagination par clé ; écritures groupées en transaction.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Dresseur de requêtes

Prochaine mission : Cache et invalidation

