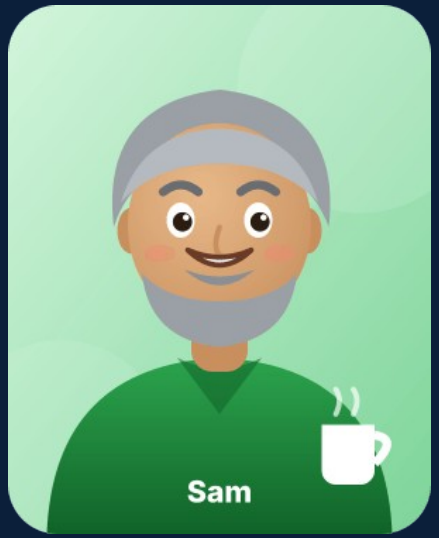
HEF-070 · MISSION 07 / 10

Index et plans d'exécution

Un index, c'est le sommaire d'un livre : il évite de tout relire pour trouver une page.





Badge : Accélérateur de requêtes

85 min · intermédiaire

L'histoire

L'histoire

La page « mes inscriptions » de HashBox met 8 secondes pour certains membres. La table `inscriptions` contient 5 millions de lignes. La requête est correcte, mais la base lit **toute la table** à chaque fois.

> *« Demande à la base comment elle exécute ta requête, dit Sam. Elle te le dira. Et ensuite, donne-lui un index. »*



Tu apprends à lire un plan d'exécution et à choisir les bons index.

Ta mission

À la fin, je serai capable de…

1. expliquer ce qu'est un index (B-tree) et ce qu'il accélère
2. lire un plan d'exécution (SCAN ou SEARCH)
3. créer un index simple, composite et couvrant
4. expliquer pourquoi un index ralentit les écritures
5. mesurer le gain d'un index

Défi d'échauffement

Tu cherches le mot « abeille » dans un dictionnaire de 1 000 pages. Option A : tu lis page par page depuis le début. Option B : tu ouvres vers le début (lettre A), tu ajustes. Option C : tu utilises les onglets alphabétiques. Combien de pages lis-tu environ pour chaque option ? Quel prix à payer si le dictionnaire est mis à jour chaque jour (ajouter des mots, refaire les onglets) ?

Le concept

Sans index : le balayage complet

Pour répondre à `WHERE ville = 'Lyon'`, la base lit **chaque ligne** de la table : c'est un **balayage complet** (*full table scan*). Pour n lignes : O(n).

Avec un index : la recherche

Un **index** est une structure supplémentaire, triée, qui permet de retrouver rapidement les lignes qui ont une valeur donnée. La plupart des index sont des **arbres B** (*B-tree*) : des arbres équilibrés larges, optimisés pour le disque. Une recherche coûte O(log n) : quelques lectures, même pour des millions de lignes.

```
Index sur ville :
  Lille  → lignes 7, 12
  Lyon   → lignes 1, 3, 9, 15
  Nice   → ligne 4
```

Le concept

Avec un index : la recherche (suite)

```
(trié par ville ; on cherche 'Lyon' par recherche dichotomique)
```

Créer un index

```
CREATE INDEX idx_membres_ville ON membres(ville);
CREATE INDEX idx_insc_membre_cours ON inscriptions(membre_id, cours_code);   -- co
   mposite
CREATE UNIQUE INDEX idx_membres_email ON membres(email);                    -- +
   unicité
```

Une clé primaire et une contrainte `UNIQUE` créent automatiquement un index.

Lire un plan

Le concept

Lire un plan (suite)

```
EXPLAIN QUERY PLAN SELECT * FROM membres WHERE ville = 'Lyon';
```

Plan	Signification
`SCAN membres`	balayage complet
`SEARCH membres USING INDEX idx_membres_ville (ville=?)`	recherche par l'index
`SEARCH ... USING COVERING INDEX`	l'index suffit, la table n'est pas lue

(PostgreSQL : `EXPLAIN` / `EXPLAIN ANALYZE`, avec `Seq Scan`, `Index Scan`, `Index Only Scan`.)

Index composite : l'ordre compte

Un index sur `(a, b)` sert pour : `WHERE a = ?` et `WHERE a = ? AND b = ?` (**préfixe gauche**). Il ne sert **pas** pour `WHERE b = ?` seul.

Le concept

Index couvrant

Si l'index contient **toutes les colonnes** demandées, la base n'a pas besoin de lire la table.

```
CREATE INDEX idx_couvrant ON inscriptions(membre_id, cours_code);
SELECT cours_code FROM inscriptions WHERE membre_id = 42;     -- répondu par l'ind
   ex seul
```

Sélectivité

Un index est utile quand il **élimine beaucoup de lignes**. Un index sur un booléen (50 % / 50 %) apporte peu. Sur un identifiant unique, il est excellent.

Le concept

Le coût des index

Avantage	Coût
lectures plus rapides	**écritures plus lentes** (chaque INSERT/UPDATE/DELETE met à jour les index)
	espace disque

Ne pas indexer partout : indexer ce que les requêtes **réelles** utilisent.

Quand un index n'est pas utilisé

- Fonction sur la colonne : `WHERE LOWER(nom) = 'ada'` (sauf index sur l'expression).
- `LIKE '%ada'` (joker au début).
- Types incompatibles, très peu sélectif, ou table minuscule (balayer est plus rapide).

Le concept

Mesurer

```
import sqlite3, time

bd = sqlite3.connect(":memory:")
bd.execute("CREATE TABLE t (id INTEGER PRIMARY KEY, ville TEXT)")
bd.executemany("INSERT INTO t (ville) VALUES (?)", [(f"ville{i % 1000}",) for i in
    range(200_000)])
t = time.perf_counter(); bd.execute("SELECT COUNT(*) FROM t WHERE ville = 'ville7'
   ").fetchone(); print(round(time.perf_counter() - t, 4))
bd.execute("CREATE INDEX idx ON t(ville)")
t = time.perf_counter(); bd.execute("SELECT COUNT(*) FROM t WHERE ville = 'ville7'
   ").fetchone(); print(round(time.perf_counter() - t, 4))
```

Exemple pas à pas

Problème : accélérer la page « mes inscriptions » avec des index, en lisant le plan avant et après.

```
import random, sqlite3, time

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE inscriptions (id INTEGER PRIMARY KEY, membre_id INTEGER, cours_code T
   EXT, date TEXT);
''')
rng = random.Random(1)
bd.executemany("INSERT INTO inscriptions (membre_id, cours_code, date) VALUES (?,
   ?, ?)",
               [(rng.randint(1, 50_000), f"HEF-{rng.randint(0, 160):03d}", "2025-0
   1-01") for _ in range(300_000)])

def plan(sql):
    return [ligne[3] for ligne in bd.execute("EXPLAIN QUERY PLAN " + sql)]
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def mesurer(sql, n=200):
    t = time.perf_counter()
    for i in range(n):
        bd.execute(sql, (1 + i * 7,)).fetchall()
    return (time.perf_counter() - t) / n * 1000

requete = "SELECT cours_code FROM inscriptions WHERE membre_id = ?"
print("sans index :", plan(requete.replace("?", "42")), round(mesurer(requete, 20)
   , 3), "ms")

bd.execute("CREATE INDEX idx_membre ON inscriptions(membre_id)")
print("index simple :", plan(requete.replace("?", "42")), round(mesurer(requete),
   3), "ms")

bd.execute("CREATE INDEX idx_couvrant ON inscriptions(membre_id, cours_code)")
print("index couvrant :", plan(requete.replace("?", "42")), round(mesurer(requete)
   , 3), "ms")
```

Exemple pas à pas (suite)

Résultats typiques :

Étape	Plan	Temps par requête
Sans index	`SCAN inscriptions`	une dizaine de ms
Index simple	`SEARCH ... USING INDEX idx_membre`	moins d'une ms
Index couvrant	`SEARCH ... USING COVERING INDEX idx_couvrant`	encore un peu moins

Lecture : on passe d'un balayage de 300 000 lignes à quelques lectures. Avec l'index couvrant, la table n'est même plus consultée.

Le coût en écriture :

Exemple pas à pas (suite)

```
import sqlite3, time

def temps_insertion(avec_index):
    bd = sqlite3.connect(":memory:")
    bd.execute("CREATE TABLE t (id INTEGER PRIMARY KEY, a INTEGER, b INTEGER, c IN
   TEGER)")
    if avec_index:
        bd.execute("CREATE INDEX i1 ON t(a)"); bd.execute("CREATE INDEX i2 ON t(b)
   "); bd.execute("CREATE INDEX i3 ON t(c)")
    donnees = [(i % 1000, i % 777, i % 333) for i in range(150_000)]
    t = time.perf_counter()
    bd.executemany("INSERT INTO t (a, b, c) VALUES (?, ?, ?)", donnees)
    return time.perf_counter() - t

print(round(temps_insertion(False), 3), "s sans index ;", round(temps_insertion(Tr
   ue), 3), "s avec 3 index")
```

Les insertions sont plus lentes avec trois index : chacun doit être maintenu.

À toi, avec aide

Table `membres(id, nom, ville, score)` de 100 000 lignes.

1. Pour les requêtes suivantes, dis quel index créer (ou pourquoi aucun) :
   2. `WHERE ville = ?`
   3. `WHERE ville = ? AND score > ?`
   4. `WHERE score > ?`
   5. `WHERE LOWER(nom) = ?`
   6. `WHERE nom LIKE '%ada'`
7. Vérifie avec `EXPLAIN QUERY PLAN` avant/après.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pour `ville = ? AND score > ?`, un index composite `(ville, score)` est idéal : égalité d'abord, plage ensuite.

Indice 2

Pour `LOWER(nom) = ?`, il faut un index sur l'expression.

Indice 3

`LIKE '%ada'` ne peut pas utiliser un index B-tree classique.

À toi, avec aide · réponse

Réponse

```
import random, sqlite3

bd = sqlite3.connect(":memory:")
bd.execute("CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT, ville TEXT, sc
   ore INTEGER)")
rng = random.Random(0)
bd.executemany("INSERT INTO membres (nom, ville, score) VALUES (?, ?, ?)",
               [(f"Nom{i}", f"Ville{rng.randint(1, 300)}", rng.randint(0, 100)) fo
   r i in range(100_000)])

def plan(sql):
    return [l[3] for l in bd.execute("EXPLAIN QUERY PLAN " + sql)]

cas = {
    "ville": "SELECT * FROM membres WHERE ville = 'Ville5'",
    "ville+score": "SELECT * FROM membres WHERE ville = 'Ville5' AND score > 50",
    "score": "SELECT * FROM membres WHERE score > 98",
    "lower": "SELECT * FROM membres WHERE LOWER(nom) = 'nom7'",
    "like": "SELECT * FROM membres WHERE nom LIKE '%7'",
```

À toi, avec aide · réponse

Réponse (suite)

```
}
print("avant :", {k: plan(v)[0] for k, v in cas.items()})
bd.execute("CREATE INDEX idx_ville_score ON membres(ville, score)")
bd.execute("CREATE INDEX idx_score ON membres(score)")
bd.execute("CREATE INDEX idx_lower_nom ON membres(LOWER(nom))")
print("après :", {k: plan(v)[0] for k, v in cas.items()})
```

Résumé : `idx_ville_score` sert aux deux premières requêtes (préfixe gauche), `idx_score` à la troisième, l'index sur expression à la quatrième, et la cinquième (`LIKE '%…'`) reste un balayage.

À toi, en solo

Audit d'index. Voici les 5 requêtes les plus fréquentes d'une application :

1. `SELECT * FROM commandes WHERE client_id = ? ORDER BY date DESC LIMIT 20`
2. `SELECT COUNT(*) FROM commandes WHERE statut = 'en_attente'`
3. `SELECT * FROM commandes WHERE date BETWEEN ? AND ?`
4. `SELECT produit_id, SUM(quantite) FROM lignes WHERE commande_id = ? GROUP BY produit_id`
5. `UPDATE commandes SET statut = ? WHERE id = ?`

La table `commandes` reçoit 200 insertions par seconde.

1. Crée les tables avec 200 000 commandes et 600 000 lignes.
2. Propose un jeu d'index **minimal** et justifie chacun.
3. Mesure avant/après et le coût en insertion.

À toi, en solo

À toi, en solo (suite)

4. Identifie un index que tu **n'as pas** créé et dis pourquoi.

À toi, en solo · réponse

Pistes de correction

```
import random, sqlite3, time

rng = random.Random(3)
bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE commandes (id INTEGER PRIMARY KEY, client_id INTEGER, date TEXT, stat
   ut TEXT);
CREATE TABLE lignes (commande_id INTEGER, produit_id INTEGER, quantite INTEGER);
''')
statuts = ["livree"] * 95 + ["en_attente"] * 3 + ["annulee"] * 2
bd.executemany("INSERT INTO commandes (client_id, date, statut) VALUES (?, ?, ?)",
               [(rng.randint(1, 20000), f"2025-{rng.randint(1,12):02d}-{rng.randin
   t(1,28):02d}", rng.choice(statuts)) for _ in range(200_000)])
bd.executemany("INSERT INTO lignes VALUES (?, ?, ?)", [(rng.randint(1, 200000), rn
   g.randint(1, 500), rng.randint(1, 5)) for _ in range(600_000)])

def temps(sql, params, n=30):
    t = time.perf_counter()
    for _ in range(n):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        bd.execute(sql, params).fetchall()
    return (time.perf_counter() - t) / n * 1000

R = {
    "client": ("SELECT * FROM commandes WHERE client_id = ? ORDER BY date DESC LIM
   IT 20", (500,)),
    "date": ("SELECT * FROM commandes WHERE date BETWEEN ? AND ?", ("2025-03-01",
   "2025-03-03")),
    "lignes": ("SELECT produit_id, SUM(quantite) FROM lignes WHERE commande_id = ?
    GROUP BY produit_id", (777,)),
}
avant = {k: round(temps(*v), 3) for k, v in R.items()}
bd.execute("CREATE INDEX idx_cmd_client_date ON commandes(client_id, date)")
bd.execute("CREATE INDEX idx_cmd_date ON commandes(date)")
bd.execute("CREATE INDEX idx_lignes_cmd ON lignes(commande_id, produit_id, quantit
   e)")
apres = {k: round(temps(*v), 3) for k, v in R.items()}
print("avant (ms):", avant)
print("après (ms):", apres)
```

Pistes de correction (suite)

Index créés : `(client_id, date)` sert la requête 1 (filtre + tri), `date` la 3, `lignes(commande_id, …)` la 4 (couvrant). La requête 5 utilise la clé primaire (déjà indexée). **Index non créé** : sur `statut` pour la requête 2 : seulement ~3 % de lignes sont « en_attente » mais la table est en écriture intensive et le comptage est rare ; un index **partiel** (`WHERE statut = 'en_attente'`) serait une meilleure option s'il devenait critique.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Indexer toutes les colonnes	Écritures lentes, espace gaspillé	Indexer ce que les requêtes réelles utilisent
Mauvais ordre dans un index composite	L'index ne sert pas	Égalités d'abord, plage ensuite
Fonction sur la colonne indexée	L'index est ignoré	Index sur l'expression ou réécrire la condition
Ne pas vérifier le plan	On croit qu'il est utilisé	`EXPLAIN` avant et après

Mini-boss

Mini-boss

Dossier d'optimisation d'une base. Construis une base de 500 000 lignes pour une plateforme de e-commerce (clients, produits, commandes, lignes). Prends cinq requêtes lentes de ton choix (recherche par client, par période, top produits, recherche par préfixe de nom, comptage par statut).

1. Mesure chaque requête sans index et relève le plan.
2. Propose des index et justifie-les (sélectivité, ordre des colonnes, couvrant ou non).
3. Mesure à nouveau, calcule le gain.
4. Mesure le coût sur 100 000 insertions avec tous les index.
5. Écris une recommandation : quels index garder, lesquels abandonner, pourquoi.

Mini-boss · réponse

Pistes de correction

Un bon dossier : tableau avant/après (ms) pour chaque requête et son plan, avec le facteur d'accélération ; un index composite justifié par « égalité puis plage » ; un index couvrant pour le top produits ; une explication de ce que `LIKE 'abc%'` peut utiliser (en SQLite, avec `COLLATE NOCASE` ou `PRAGMA case_sensitive_like`, ou un index avec collation adaptée) ; une mesure honnête de l'insertion (elle peut être plusieurs fois plus lente avec quatre index) ; une décision finale qui abandonne l'index peu sélectif ou redondant (un index `(a)` est redondant si `(a, b)` existe).

Quiz éclair

Quiz éclair

1. Un index sert à :

A. retrouver rapidement des lignes sans tout lire
B. chiffrer les données
C. dupliquer une table

2. `SCAN` dans un plan d'exécution signifie :

A. recherche par index
B. balayage complet de la table
C. jointure

3. Un index composite `(a, b)` sert pour :

A. WHERE b = ? seul

Quiz éclair

Quiz éclair (suite)

B. aucune requête
C. WHERE a = ? (préfixe gauche)

4. 🔁 *Révision mission 06* : Un LEFT JOIN renvoie :

A. toutes les lignes de gauche, avec NULL s'il n'y a pas de correspondance
B. uniquement les lignes avec correspondance
C. les lignes de droite seulement

5. 🔁 *Révision mission 06* : HAVING filtre :

A. les lignes avant le regroupement
B. les groupes après le GROUP BY
C. les colonnes

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Quels « index » utilises-tu dans la vie (sommaire, marque-pages, répertoire) et quel est leur coût d'entretien ?
2. Pourquoi un index mal choisi peut-il ralentir une application ?

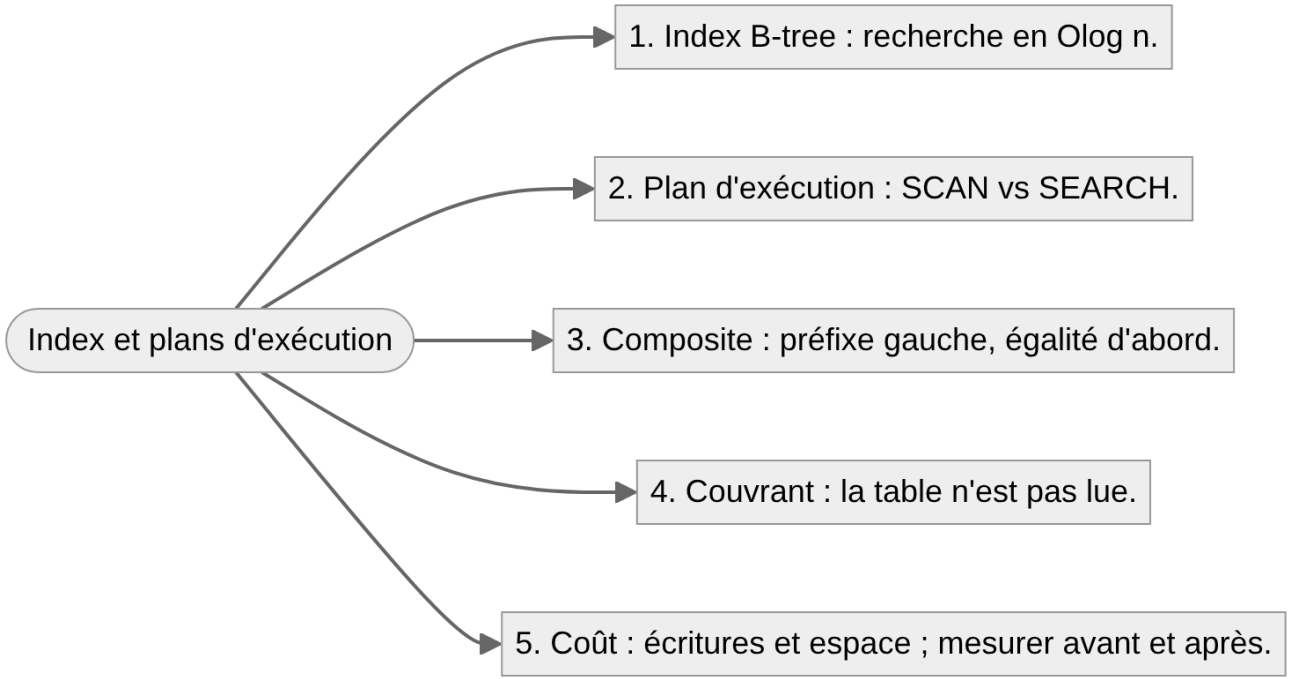
À retenir

À retenir

1. Index B-tree : recherche en O(log n).
2. Plan d'exécution : SCAN vs SEARCH.
3. Composite : préfixe gauche, égalité d'abord.
4. Couvrant : la table n'est pas lue.
5. Coût : écritures et espace ; mesurer avant et après.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Accélérateur de requêtes

Prochaine mission : Transactions et ACID

