L'informatique des entrepôts de données

Daniel Lemire

SEMAINE 4
Les techniques d'indexation

4.1. Présentation de la semaine

Les entrepôts de données exploitent diverses techniques d'indexation pour optimiser les performances des requêtes. Après avoir abordé la sélection des vues la semaine précédente, nous nous concentrons cette semaine sur les index traditionnels tels que les arbres B et les tables de hachage. L'objectif est de comprendre leur utilisation dans le contexte des entrepôts de données, sans nécessiter leur implémentation.

Nous explorerons également l'indexation de données hétérogènes, comme le texte et les fichiers XML, qui dépassent le cadre des tables relationnelles. Enfin, nous aborderons l'indexation des jointures, introduisant les index multidimensionnels qui seront étudiés la semaine prochaine.

4.2. Arbres B et tables de hachage

Dans une colonne contenant n éléments, il est courant de vouloir identifier rapidement les éléments correspondant à un critère spécifique, comme `nom=Jean`. Sans index, une recherche exhaustive nécessite de parcourir tous les n éléments, mais des structures comme les arbres B et les tables de hachage permettent d'améliorer les performances.

4.2.1. Arbres B

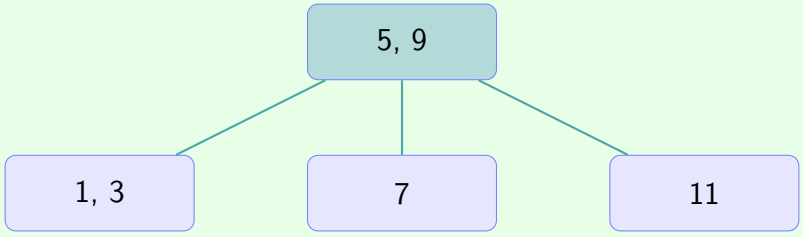
Les arbres B sont des structures arborescentes équilibrées, optimisées pour les accès disques dans les bases de données. Introduits en 1972 par Rudolf Bayer et Edward M. McCreight, ils offrent une recherche en $O(\log n)$, comparable à la recherche binaire, tout en permettant un accès ordonné aux valeurs. En Java, la classe `java.util.TreeMap` illustre une telle structure.

Un arbre B d'ordre m ($m \geq 2$) respecte les propriétés suivantes :

(a) Chaque nœud contient entre $m-1$ et $2m-2$ clés, sauf la racine (1 à $2m-2$ clés).
(b) Un nœud interne avec k clés possède $k+1$ enfants.
(c) Toutes les feuilles sont au même niveau.
(d) Les clés sont stockées en ordre croissant.
(e) Les clés des enfants respectent un ordre strict par rapport aux clés parentes.

Exemple 1: Exemple d'arbre B

Voici un exemple d'arbre B d'ordre 3 avec les clés $\{1, 3, 5, 7, 9, 11\}$:



Les opérations principales incluent :

- **Recherche** : Navigation arborescente similaire à un arbre binaire.
- **Insertion** : Ajout dans une feuille, avec division des nœuds pleins.
- **Suppression** : Suppression et rééquilibrage si nécessaire.

4.2.2. Tables de hachage

Les tables de hachage offrent une recherche en $O(1)$ (amorti), mais ne permettent pas un accès ordonné. En Java, `java.util.HashMap` est un exemple. Leur fonctionnement repose sur :

(a) Une **fonction de hachage** $h(k)$ mappant une clé à un indice.
(b) La gestion des **collisions** par chaînage ou sondage.
(c) Le **facteur de charge**, influençant les performances.

Pour une table de taille $m = 7$ avec $h(k) = k \mod 7$ et les clés $\{15, 22, 8, 29\}$, toutes tombent à l'indice 1, gérées par chaînage.

4.2.3. Comparaison des performances

Considérons un programme Java comparant `TreeMap` et `HashMap` pour sélectionner des éléments uniques :

```java
import java.util.*;

public class Comparaison {
    public static void main(String[] args) {
        TreeMap<Integer, String> treeMap = new
            TreeMap<>();
        HashMap<Integer, String> hashMap = new
            HashMap<>();
        // Code de test
    }
}
```

La Figure 1 montre que `HashMap` reste stable avec l'augmentation des données, contrairement à `TreeMap`, qui ralentit. Pour un million d'éléments, `HashMap` est jusqu'à 20 fois plus rapide. Cependant, pour des recherches par plage (ex. : âge > 20 ans), les arbres B sont préférables. Sur disque, les arbres B minimisent les accès aléatoires, rendant leur performance compétitive.

Certaines bases de données, comme SQLite, utilisent principalement des index de type B-tree pour leurs index. En effet, les arbres B sont adaptés aux systèmes de fichiers, car ils minimisent les accès disques en regroupant les données de manière hiérarchique. Ils sont particulièrement efficaces pour les requêtes nécessitant un accès ordonné ou des plages de valeurs.

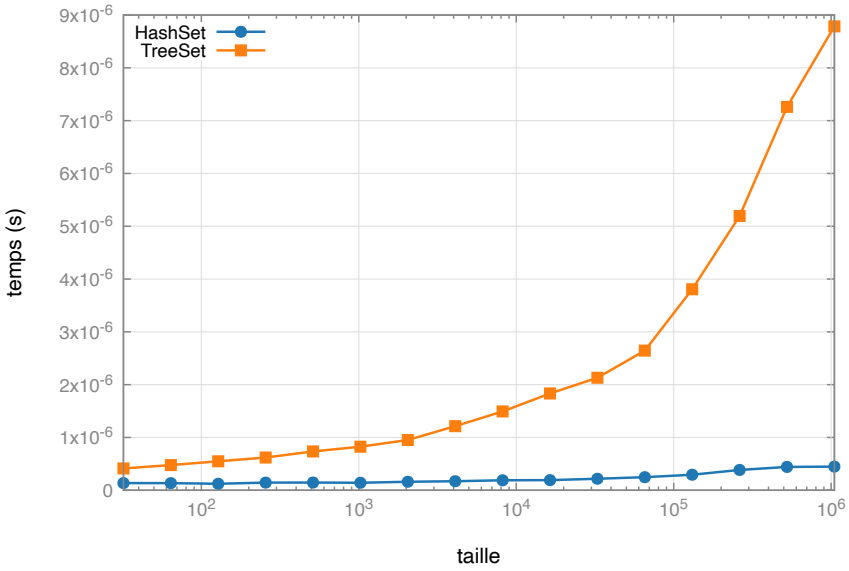


Figure 1. Comparaison des performances entre `TreeMap` et `HashMap` pour la sélection aléatoire d'éléments.

4.3. Indexation du texte

L'indexation textuelle repose sur l'*index inversé*, qui associe un mot à la liste des documents où il apparaît, souvent avec ses positions. Par exemple, pour les textes :

- D1 = "La vie est belle"
- D2 = "Belle belle"

l'index inversé est :

- la → D1,1
- vie → D1,2
- est → D1,3
- belle → D1,4; D2,1; D2,2

Avec une table de hachage, la recherche est en $O(1)$, mais la construction ou mise à jour peut être en $O(n)$. La compression réduit l'espace requis, souvent à 10 % des documents originaux. Un exemple d'implémentation avec `HashMap` :

```
1 import java.util.*;
2
3 public class IndexInverse {
4     HashMap<String, List<String>> index = new
          HashMap<>();
5     public void indexer(String docId, String texte) {
6         String[] mots = texte.split(" ");
```

```
        for (int i = 0; i < mots.length; i++) {
            index.computeIfAbsent(mots[i], k -> new
                ArrayList<>()).add(docId + "," + (i+1));
        }
    }
}
```

Les stratégies d'indexation incluent :

- Exclusion des mots vides via un *antidictionnaire*.
- *Lemmatisation* pour regrouper les variantes d'un mot (ex. : "courais" → "courir").
- Pondération par *TF-IDF* pour classer les résultats par pertinence.

4.4. Indexation XML

Les bases de données comme IBM DB2 ou Oracle indexent les documents XML, interrogés via XPath ou XQuery. XPath sélectionne des nœuds, tandis que XQuery permet des transformations complexes avec des expressions FLWOR.

Pour le document XML :

```
<livres>
    <livre>
        <titre>Le Petit Prince</titre>
        <auteur>Antoine de Saint-Exupery</auteur>
        <annee>1943</annee>
    </livre>
    <livre>
        <titre>1984</titre>
        <auteur>George Orwell</auteur>
        <annee>1949</annee>
    </livre>
</livres>
```

Une requête XPath `/livres/livre[annee > 1945]/titre` retourne `1984`. Une requête XQuery trie les livres après 1940 :

```
for $livre in /livres/livre
where $livre/annee > 1940
order by $livre/annee
return
    <resultat>
        <titre>{$livre/titre/text()}</titre>
```

```
        <auteur>{$livre/auteur/text()}</auteur>
    </resultat>
```

L'indexation XML avec ORDPATH attribue des identifiants hiérarchiques (ex. : 1, 1.1, 1.1.1). Pour le document :

```
<liste temps="janvier">
    <bateau />
    <bateau>
        <canard />
    </bateau>
</liste>
```

la table ORDPATH est :

ORDPATH	Nom	Type	Valeur
1	liste	élément	aucune
1.1	temps	attribut	janvier
1.2	bateau	élément	aucune
1.3	bateau	élément	aucune
1.3.1	canard	élément	aucune

Table 1. Représentation ORDPATH du document XML.

Un index sur la colonne `valeur` accélère les requêtes comme `@temps="janvier"`.

4.5. Indexation des jointures

Les jointures SQL sont coûteuses sans index. L'algorithme Sort-Merge (voir Algorithme 1) trie les tables en $O(n \log n)$, puis fusionne en $O(n)$. Considérons les tables suivantes.

Nom	Profit
Jean	531.00
Pierre	132.00
Marie	32.00

Entrées : Deux relations R et S à joindre sur l'attribut A
Sorties : La jointure de R et S
Trier R par ordre croissant sur A; Trier S par ordre croissant sur A; Initialiser $i \leftarrow 1$, $j \leftarrow 1$; Résultat $\leftarrow$ ensemble vide;
while $i \leq |R|$ *et* $j \leq |S|$ **do**
  if $R[i].A < S[j].A$ **then**
    $i \leftarrow i + 1$;
  else
    if $R[i].A > S[j].A$ **then**
      $j \leftarrow j + 1$;
    else
      $k \leftarrow i$; **while** $k \leq |R|$ *et* $R[k].A = R[i].A$ **do**
        $l \leftarrow j$; **while** $l \leq |S|$ *et* $S[l].A = S[j].A$ **do**
          Ajouter le tuple joint $(R[k], S[l])$ à Résultat;
          $l \leftarrow l + 1$;
        end
        $k \leftarrow k + 1$;
      end
      $i \leftarrow k$; $j \leftarrow l$;
    end
  end
end
return *Résultat*;

Algorithme 1 : Algorithme Sort-Merge Join

Nom	Ville
Jean	Montréal
Marie	Québec
Pierre	Roberval

Après tri, la jointure produit le résultat suivant.

Nom	Profit	Nom	Ville
Jean	531.00	Jean	Montréal
Marie	32.00	Marie	Québec
Pierre	132.00	Pierre	Roberval

Entrées : Deux relations R et S (supposons S plus petite que R), jointure sur l'attribut A
Sorties : La jointure de R et S
Créer une table de hachage H vide; **pour** *chaque tuple* s *dans* S **faire**
  Calculer $h = hash(s.A)$; Ajouter s à la liste dans $H[h]$
fin
Résultat ← ensemble vide; **pour** *chaque tuple* r *dans* R **faire**
  Calculer $h = hash(r.A)$; **pour** *chaque* s *dans* $H[h]$ *tel que* $s.A = r.A$ **faire**
    Ajouter le tuple joint (r, s) à Résultat;
  fin
fin
retourner *Résultat*;

Algorithme 2 : Algorithme Hash Join

L'algorithme Hash Join (voir Algorithme 2) utilise une table de hachage pour associer les enregistrements en $O(n)$, mais nécessite des accès aléatoires. Dans les entrepôts de données, minimiser les jointures non-indexées est préférable.

Certains moteurs de bases de données, comme PostgreSQL, utilisent des algorithmes de jointure adaptatifs, combinant les approches Sort-Merge et Hash Join selon les statistiques des données. Par exemple, si une table est indexée sur l'attribut de jointure, le moteur peut choisir la jointure par hachage.

Heureusement, plusieurs jointures sont naturellement indexées, comme les jointures sur les clés primaires et étrangères. Par exemple, dans une base de données de ventes, la table des ventes peut être jointe à la table des clients via l'ID client, qui est souvent indexé.

4.6. Activité pratique : indexation dans une base de données relationnelle

Nous vous invitons maintenant à mettre en pratique les concepts d'indexation, de normalisation et d'optimisation de requêtes SQL qui ont été abordés dans les sections précédentes.

L'activité s'appuie sur une base de données réelle et publique : les données salariales du secteur public de l'Ontario (Sunshine list). Ces données, bien qu'anonymisées dans certains contextes, permettent d'observer concrètement l'impact des index sur les performances des requêtes, en particulier sur des opérations de jointure, de filtrage et d'agrégation.

Rendez-vous à l'adresse suivante pour accéder au dépôt contenant les scripts et les instructions détaillées :

https://github.com/lemire/sunshine

4.6.1. Objectifs de l'activité

L'objectif principal est de vous permettre d'expérimenter directement l'effet des index sur les temps d'exécution des requêtes. Plus précisément, vous allez :

- charger un fichier CSV volumineux dans une base SQLite bien normalisée ;
- observer le plan d'exécution des requêtes avant et après la création d'index ;
- mesurer l'impact d'un index sur une colonne fréquemment filtrée (par exemple le nom de famille) ;
- comparer les performances d'une jointure sur plusieurs tables avec et sans index appropriés.

4.6.2. Déroulement suggéré

(a) Clonez le dépôt ou téléchargez-le directement depuis GitHub.
(b) Installez Python (version 3.8 ou supérieure recommandée) si ce n'est pas déjà fait sur votre machine.
(c) Téléchargez le fichier CSV des données Sunshine (le lien est normalement indiqué dans le README du dépôt).
(d) Exécutez le script de création de la base de données :

```
python create.py
```

Ce script normalise les données et produit un fichier `database.bin`.
(e) Lancez le script de benchmark pour observer les performances sans index supplémentaire :

```
python benchmark.py
```

(f) Ajoutez un ou plusieurs index (par exemple sur la colonne `last_name`) en suivant les exemples du README ou en modifiant directement le script, puis relancez le benchmark.
(g) Testez une requête ciblée (par nom de famille) avec et sans index grâce au script `query.py`.

4.6.3. nterprétation des plans d'exécution

Les **plans d'exécution** (query execution plans ou explain plans) repose sur un principe assez simple : quand vous écrivez une requête SQL, vous dites *quoi* calculer, mais vous ne dites presque jamais *comment* le

faire de manière efficace. C'est le moteur de base de données qui prend cette responsabilité.

Le moteur dispose d'un composant appelé **optimiseur de requêtes** (query optimizer). Cet optimiseur analyse la requête, lit les statistiques disponibles (nombre de lignes, distribution des valeurs, cardinalité des colonnes, taille des tables, etc.), examine les index existants et construit plusieurs stratégies d'exécution possibles. Ensuite il estime le coût de chacune (principalement en termes d'entrées-sorties disque et de CPU) et choisit celle qu'il juge la moins chère. Le résultat de ce choix est précisément le plan d'exécution.

Ce plan est une sorte de recette détaillée : il indique l'ordre dans lequel les tables seront accédées, le type d'accès (balayage complet ou recherche dans un index), la méthode de jointure (nested loop, hash join, merge join...), les tris éventuels, les regroupements, les filtres appliqués à quel moment, etc. En d'autres termes, il montre le chemin que les données vont réellement emprunter pour arriver jusqu'au résultat final.

Dans la grande majorité des moteurs relationnels (PostgreSQL, MySQL, SQL Server, Oracle, SQLite...), on peut demander ce plan sans exécuter la requête (`EXPLAIN`) ou après l'avoir exécutée avec des mesures réelles (`EXPLAIN ANALYZE` ou *Include Actual Execution Plan*). Dans SQLite, la commande la plus lisible est `EXPLAIN QUERY PLAN`.

4.6.4. Comment lire un plan d'exécution

Un plan se lit généralement de bas en haut et de droite à gauche : on commence par les opérations les plus profondes (accès aux tables) et on remonte vers les opérations finales (projection, tri, limite...).

Quelques motifs très courants à repérer :

- `SCAN` ou `TABLE SCAN` → balayage séquentiel complet de la table. C'est souvent le signal le plus alarmant quand la table contient des dizaines ou centaines de milliers de lignes. Cela signifie que le moteur lit presque tout le contenu, ligne par ligne.
- `SEARCH ...USING INDEX` → recherche ciblée dans un index. C'est exactement ce qu'on veut voir quand on filtre sur une colonne indexée. Si l'index est *covering* (toutes les colonnes nécessaires sont dans l'index), on évite même de retourner dans la table principale.
- `USE TEMP B-TREE FOR ORDER BY` ou `USE TEMP B-TREE FOR GROUP BY` → le moteur crée une structure temporaire en mémoire (ou sur disque) pour trier ou grouper. C'est coûteux si le volume est important.

- `JOIN` avec `NESTED LOOP` → jointure par boucles imbriquées. Acceptable quand une des deux tables est très petite ou très bien filtrée ; sinon on préfère souvent hash join ou merge join sur les gros volumes.
- `ROWS` / `ESTIMATED ROWS` (selon le moteur) → estimation du nombre de lignes traitées à chaque étape. Quand l'estimation est très éloignée de la réalité (surtout sur les jointures), c'est souvent un signe que les statistiques sont périmées.

4.6.5. Utiliser les plans pour optimiser

Observer le plan avant et après une modification permet de valider concrètement si l'on progresse. Voici les cas les plus fréquents où le plan guide l'optimisation :

- On voit un `SCAN` sur une table volumineuse alors qu'on filtre sur une colonne → créer un index sur cette colonne (ou sur la combinaison colonne + filtre fréquent).
- On filtre sur plusieurs colonnes avec `AND` → un index composite (multi-colonnes) dans le bon ordre est souvent plus efficace qu'un index par colonne.
- Une jointure est lente → vérifier si les colonnes de jointure sont indexées des deux côtés (surtout la table qu'on parcourt en inner loop).
- On trie fréquemment sur la même colonne → un index sur cette colonne peut parfois transformer un tri en simple parcours d'index (index-ordered scan).
- Le plan change radicalement après `ANALYZE` / `UPDATE STATISTICS` → les statistiques périmées trompent souvent l'optimiseur.

Dans le contexte SQLite (comme dans le dépôt sunshine), les gains les plus spectaculaires viennent presque toujours de la suppression d'un `SCAN` au profit d'un `SEARCH` via index. Un index sur `last_name` peut diviser le temps par 50 ou 100 pour une recherche par nom, parce qu'au lieu de lire 500 000 lignes, on lit seulement quelques dizaines d'entrées d'index + les lignes correspondantes.

En résumé, le plan d'exécution n'est pas un outil de débogage : c'est le principal moyen de dialoguer avec l'optimiseur. En regardant ce qu'il choisit de faire, on comprend rapidement pourquoi une requête est lente et on peut agir là où ça compte (index, réécriture de la requête pour aider l'optimiseur, mise à jour des statistiques, ou parfois refonte du schéma). C'est une des compétences qui sépare un utilisateur SQL moyen d'une personne qui arrive à rendre des requêtes 10–100× plus rapides sur de gros volumes.

4.6.6. Questions à explorer pendant l'activité

- Combien de fois la requête est-elle accélérée grâce à l'ajout d'un index sur `last_name`?
- Quel est l'impact d'un index sur une jointure impliquant les tables `individuals` et `salaries`?
- Dans quelles circonstances un index peut-il ne pas être utilisé par le moteur SQLite?
- Quelle est la taille ajoutée au fichier de base de données par les index créés?
- Proposez un index composite utile pour une requête qui filtre à la fois sur le secteur d'activité et l'année.

4.7. Questions d'approfondissement

(a) Quel type d'index utiliser pour identifier les événements dans une plage temporelle?
(b) Si un arbre B et une table de hachage ont des performances équivalentes pour un million de rangées, combien de rangées sont nécessaires pour que la table de hachage soit deux fois plus rapide?
(c) Pourquoi utiliser un arbre B plutôt qu'une recherche binaire sur une liste triée?
(d) Un index inversé peut-il trouver les textes contenant un mot se terminant par *ent*?
(e) Quels sont les inconvénients de la représentation ORDPATH par rapport au stockage direct d'un document XML?

4.8. Réponses suggérées

(a) Arbre B.
(b) En théorie, une table de hachage a une complexité de $O(1)$, et un arbre B de $O(\log n)$. Pour $\log_2 n = 2 \log_2 10^6$, on a $n = 10^{12}$.
(c) Un arbre B permet des mises à jour en $O(\log n)$, contrairement à une liste triée.
(d) Non, sauf avec une indexation spécifique des suffixes.
(e) La reconstruction d'un document XML à partir d'ORDPATH est coûteuse si le document entier est requis.

Votre avis compte !

Chers étudiants,
Si vous repérez une erreur dans ce document ou si vous avez une suggestion pour l'améliorer, nous vous invitons à remplir notre **formulaire anonyme**.
Votre contribution est précieuse pour nous !

Cliquez ici pour accéder au formulaire