Bases de données

Ecole Marocaine des Sciences de l'Ingénieur

© Yousra Lembachar

Optimisation des requêtes SQL

Optimisation des requêtes

- Gagner en temps d'exécution
- Gagner en ressources mémoire
- Eviter les bugs

Utilisation d'indexes dans SQL

Indexes

- Un index est une structure de données stockée avec la base de données qui permet de scanner rapidement les données et améliorer la performance des requêtes SQL

Pourquoi utiliser des indexes?

SELECT nom FROM ETUDIANT
WHERE nom like 'S%'

ETUDIANT

Filter: Edit

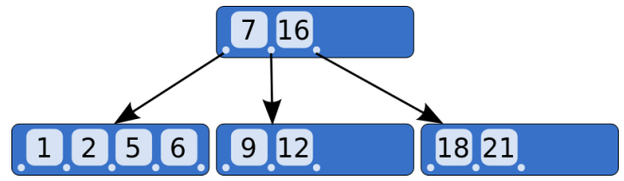
idEtudiant	nom
1	Samantha
2	Francis
3	Charles
4	Penelope
5	Craig
6	Steve
7	Dave
	NULL

Avec un index sur le nom, le SGBD ne va pas scanner toute la table et retournera le résultat rapidement

=> Amélioration significative des performances (surtout lorsqu'il s'agit de grandes BDs)

Implémentation

- B-trees (Balanced trees)
  - complexité log
  - pour des clauses col =,>,< val
    - Utilisé par la plupart des indexes sous MySQL



- Table de hashage
  - complexité constante
  - pour des clauses col = val
- B+trees, R-trees, etc.

Clés primaires et uniques

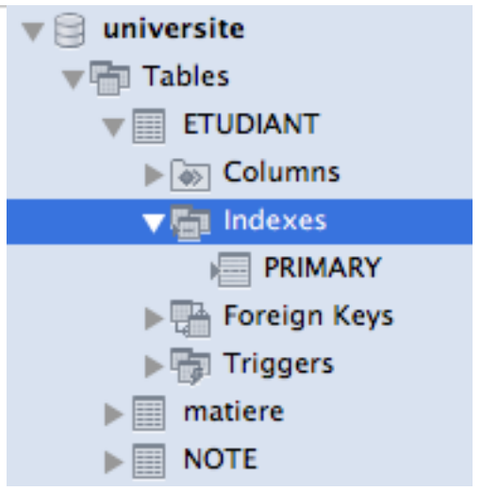
- La plupart des SGBD construisent un index sur les clés primaires et des fois même sur les clés uniques

```sql
SELECT * FROM ETUDIANT
WHERE
     idEtudiant = 7;

-- plus rapide que

SELECT * FROM ETUDIANT
WHERE
     nom = 'Dave'

-- car idEtudiant est clé primairé => indexé
```



Indexes

```sql
SELECT
    *
FROM
    NOTE
WHERE
    idEtudiant = 1 and note > 14
```

Indexes possibles:

- Index sur idEtudiant
- Index sur note
- Index sur (idEtudiant, note)

Indexes

Pour ne pas utiliser des indexes sur toutes les colonnes?

- La création d'indexes prend beaucoup de temps, surtout pour des BDs très larges
- Les indexes doivent être mis à jour à chaque fois qu'il y a une mise à jour des colonnes indexées
- Les indexes utilisent de la mémoire

Comment choisir les indexes?

- Etant donnés:
  - Une instance de la BD
  - Les types de requêtes sur la BD
  - Indexes possibles
- Optimiser les requêtes revient à utiliser les indexes les moins couteux en termes de performances (temps et mémoire)
- Un utilisateur peut soit utiliser un optimisateur de requêtes qui va retourner les indexes recommandés, soit les choisir manuellement

Syntaxe SQL

- CREATE INDEX nom_index ON nom_table(col1 [, col2...])

- CREATE UNIQUE INDEX nom_index ON nom_table(col1 [, col2...])

Conclusion

- Utiliser des index
- Créer les indexes après l'insertion des données
- Créer des indexes sur les colonnes qui servent lors de la jointure
- Utiliser des indexes pour les colonnes fréquemment utilisées dans le WHERE

Autres moyens d'optimisation

Utilisation de *

- Utiliser les noms de colonnes au lieu de *
- Certains SGBD cachent les noms des colonnes retournées par un SELECT * FROM T => Si ajout d'une colonne dans T, cette colonne peut ne pas figurer lors de l'exécution de la requête à nouveau
- Perte de ressources (mémoire) lorsqu'on n'a pas besoin de lister toutes les colonnes (surtout dans le cas d'une jointure où les attributs en commun sont dupliqués)

Utilisation de *

```
SELECT
    ETUDIANT.idEtudiant, nom, idMatiere, note
FROM
    ETUDIANT,
    NOTE
WHERE
    ETUDIANT.idEtudiant = NOTE.idEtudiant;
```

au lieu

```
SELECT
    *
FROM
    ETUDIANT,
    NOTE
WHERE
    ETUDIANT.idEtudiant = NOTE.idEtudiant;
```

idEtudiant	nom	idMatiere	note
1	Samantha	1	12
1	Samantha	4	9
1	Samantha	7	9
2	Francis	1	20
3	Charles	1	13
4	Penelope	1	14
5	Craig	1	10
6	Steve	1	9

idEtudiant	nom	idEtudiant	idMatiere	note
1	Samantha	1	1	12
1	Samantha	1	4	9
1	Samantha	1	7	9
2	Francis	2	1	20
3	Charles	3	1	13
4	Penelope	4	1	14
5	Craig	5	1	10
6	Steve	6	1	9

duplication de la colonne idEtudiant

Utilisation de HAVING

- HAVING est utilisée pour créer un filtre sur les colonnes sélectionnées
- WHERE crée un filtre avant de sélectionner les colonnes => plus rapide
- Ne pas utiliser HAVING avec des conditions qui peuvent être utilisées dans le WHERE

Utilisation de HAVING

```sql
SELECT
    *
FROM
    NOTE
WHERE
    idMatiere = 1
;
-- au lieu de

SELECT
    *
FROM
    NOTE
HAVING idMatiere = 1;
```

Minimiser les sous-requêtes

- Chaque requête est un accès à la BD
- Minimiser le nombre de sous requêtes, si possible
- Convertir les sous-requêtes en jointures, si possible

Minimiser les sous-requêtes

```sql
SELECT
    nom
FROM
    ETUDIANT,
    DETAILS_ETUDIANT
WHERE
    ETUDIANT.idEtudiant = DETAILS_ETUDIANT.idEtudiant
        AND (age , moyenne) = (SELECT
            max(age), max(moyenne)
        FROM
            DETAILS_ETUDIANT);

-- au lieu


SELECT
    nom
FROM
    ETUDIANT,
    DETAILS_ETUDIANT
WHERE
    ETUDIANT.idPersonne = DETAILS_ETUDIANT.idEtudiant
        AND age = (SELECT
            max(age)
        FROM
            DETAILS_ETUDIANT)
        AND moyenne = (SELECT
            max(moyenne)
        FROM
            DETAILS_ETUDIANT);
```

Conversion des sous-requêtes en jointures

```sql
-- Utiliser
SELECT
    nom
FROM
    ETUDIANT, NOTE
WHERE
    NOTE.idEtudiant = ETUDIANT.idEtudiant;

-- au lieu de

SELECT
    nom
FROM
    ETUDIANT
WHERE
    idEtudiant IN (SELECT
            idEtudiant
        FROM
            NOTE);
```

EXISTS and IN

- EXISTS est plus rapide que IN si le résultat de la sous-requête est large
- IN est plus rapide que EXISTS si le résultat de la sous-requête est petit
- IN est souvent moins performant que EXISTS

UNION vs UNION ALL

- Union élimine les lignes dupliquées alors que UNION ALL rend toutes les lignes, même celles dupliquées
- Utiliser UNION ALL en unifiant des lignes non dupliquées, ou quand il n'y a pas besoin d'éliminer les lignes dupliquées

UNION vs UNION ALL

```
1 • SELECT idEtudiant FROM ETUDIANT
2   UNION
3   SELECT idEtudiant FROM NOTE
```

160% 6:2

Filter: Edit: File:

idEtudiant
1
2
3
4
5
6
7

```
1 • SELECT idEtudiant FROM ETUDIANT
2   UNION ALL
3   SELECT idEtudiant FROM NOTE
```

160% 28:3

Filter: Edit: File:

idEtudiant
1
2
3
4
5
6
7
1
2
3
4
5
6
1
1