SGBD-R : traitement de requêtes

Tuning de bases de données

Guillaume Raschia — Nantes Université

Dernière mise-à-jour : 13 mars 2023

Plan de la session

La recherche de performance

Poser des index

Construire des vues

Reformuler la requête SQL

Bidouiller le schéma

La recherche de performance

Le réglage (tuning) des bases de données

Une promenade au niveau de l'organisation « physique » de la base de données

Dans cette section, on étudie les leviers d'optimisation de requêtes aux bases de données.

Compétences visées

- Définir une exigence de performance sur les BD
- Connaître les grandes classes de solutions technologiques
- Appliquer des techniques de modélisation pour optimiser une BD.

L'énoncé du problème

Étant donné le profil de la charge de travail :

- La liste des lectures (`select`) avec leurs fréquences
- La liste des écritures (`update`, `insert`, `delete`) avec leurs fréquences
- Des critères de performance attendue pour chaque requête

Le tuning de base de données

C'est le réglage des détails d'implémentation dans le but d'atteindre la performance souhaitée pour le traitement des données, relativement au profil de charge

F.A.Q. à propos du profil de charge

- Quelles tables sont concernées ?
- Quels attributs au sein de ces tables sont examinés ?
- Quels sont ceux qui participent aux critères de sélection ou de jointure ?
- Quelle est la sélectivité de ces critères ?
- Quel est le volume de données traité ?
- À quelle fréquence et selon quelle répartition temporelle ?
- Quelle est la complexité des requêtes ? (nombre de jointures, niveaux d'imbrication, etc.)
- *etc.*

Réponses à formuler pour chaque requête de la charge de travail…

Poser des index

Du choix des index

Premier levier d'optimisation, avec une rentabilité souvent élevée !

Inconvénients

- Les modifications sont plus coûteuses (modification simultanée de l'index)
- Le poids de l'index peut parfois dépasser celui des données

Objectif

Déterminer l'ensemble minimal d'index qui maximise la performance de la charge de travail

C'est **a priori** un problème difficile

La clé d'index

Un index est construit sur un attribut K si les conditions suivantes sont satisfaites :

- une sélection ou une jointure opère sur K
- la fréquence des requêtes sur K est élevée
- l'attribut est très discriminant (peu de nuplets avec la même valeur)
- la table correspondante est rarement modifiée

Les index : cas numéro 1

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

100 000 requêtes :

```
SELECT *
FROM R
WHERE A=?
```

100 requêtes :

```
SELECT *
FROM R
WHERE C=?
```

Quels index ?

Les index : cas numéro 1

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

100 000 requêtes :

```
SELECT *
FROM R
WHERE A=?
```

100 requêtes :

```
SELECT *
FROM R
WHERE C=?
```

Quels index ?

$R(A)$ et $R(C)$, par hachage ou arbre B

Les index : cas numéro 2

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

100 000 requêtes :

```
SELECT *
FROM R
WHERE A BETWEEN ? AND ?
```

100 requêtes :

```
SELECT *
FROM R
WHERE C=?
```

100 000 requêtes :

```
INSERT INTO R
VALUES (?,?,?)
```

Quels index ?

Les index : cas numéro 2

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

100 000 requêtes :

```
SELECT *
FROM R
WHERE A BETWEEN ? AND ?
```

100 requêtes :

```
SELECT *
FROM R
WHERE C=?
```

100 000 requêtes :

```
INSERT INTO R
VALUES (?,?,?)
```

Quels index ?

impérativement un arbre B sur $R(A)$, et probablement rien sur $R(C)$

Les index : cas numéro 3

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

100 000 requêtes :

```
SELECT *
FROM R
WHERE A=?
```

1 000 000 requêtes :

```
SELECT *
FROM R
WHERE A=? AND C>?
```

100 000 requêtes :

```
INSERT INTO R
VALUES (?,?,?)
```

Quels index ?

Les index : cas numéro 3

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

100 000 requêtes :

```
SELECT *
FROM R
WHERE A=?
```

1 000 000 requêtes :

```
SELECT *
FROM R
WHERE A=? AND C>?
```

100 000 requêtes :

```
INSERT INTO R
VALUES (?,?,?)
```

Quels index ?

$R(A, C)$ – sset non $R(C, A)$

Les index : cas numéro 4

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

1 000 requêtes :

```
SELECT *
FROM R
WHERE A BETWEEN ? AND ?
```

100 000 requêtes :

```
SELECT *
FROM R
WHERE C BETWEEN ? AND ?
```

Quels index ?

Les index : cas numéro 4

Soit le schéma $R(A, B, C)$;
et le profil de charge suivant :

1 000 requêtes :

```
SELECT *
FROM R
WHERE A BETWEEN ? AND ?
```

100 000 requêtes :

```
SELECT *
FROM R
WHERE C BETWEEN ? AND ?
```

Quels index ?

$R(C)$ plaçant – non-dense – et $R(A)$ non plaçant – dense

Principes directeurs

Pour choisir les index

- traiter les transactions par ordre d'importance décroissante
- Se concentrer uniquement sur les tables sollicitées par les requêtes
- Examiner les clauses `WHERE` et `JOIN` pour trouver d'éventuelles clés d'index
- Privilégier les index à même d'accélérer plusieurs requêtes

À la création de table, un index primaire – sur la clé primaire – et des index sur les attributs uniques sont autmatiquement construits, pour optimiser la vérification de la contrainte d'unicité

Attention : rien de tel sur les clés étrangères

Les clés composées

Conditions pour un index composite sur $(K_1, \ldots, K_n)$

- La clause `WHERE` porte sur un préfixe de $(K_1, \ldots, K_n)$
- La clause `SELECT` contient exclusivement K_1, ..., K_n
- Un tri (`ORDER BY`, `DISTINCT`) est requis sur un préfixe de $(K_1, \ldots, K_n)$

Index couvrant

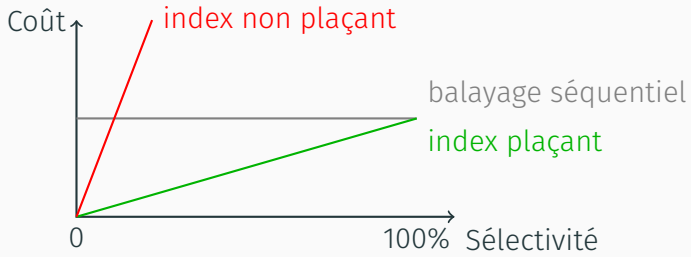
Il permet de donner la réponse à une requête, sans accès aux données

Exemple

```
CREATE INDEX idx_couvrant ON R (K1, K2);
SELECT K2 FROM R WHERE K1=55;
```

Index plaçant ou non ?

Les requêtes d'intervalle tirent profit du regroupement des données selon la clé de recherche



Les index couvrants sont insensibles au regroupement : ils fonctionnent parfaitement avec ou sans

Table de hachage vs. arbre B+

Règle n°1

Toujours construire un arbre B+ 😎

Règle n°2

Envisager une table de hachage sur K si :

- il existe une requête importante avec un prédicat d'égalité (`WHERE K=?`) et
- aucune requête d'intervalle sur K !
- aucun tri sur K !!
- aucune recherche sur un préfixe de K !!!
- aucun autre index plaçant plus avantageux !!!!
- le fichier de données n'est pas trop gros (pour éviter les collisions)

Équilibrer lectures et écritures

- Les index accélèrent les lectures
  - `SELECT FROM WHERE`
- Mais la plupart du temps, ils pénalisent les écritures
  - `INSERT`, `DELETE`, `UPDATE`

Néanmoins, certaines écritures tirent profit des index !

Exemple

```
UPDATE R SET A=7 WHERE K=55
```

Construire des vues

Les vues matérialisées

Le concept

Pré-calcul et stockage du résultat d'une requête sous la forme d'une table

Exemple

```
CREATE MATERIALIZED VIEW mv_agr AS
    SELECT P.nom AS nom, AVG(P.prix) AS prix
    FROM Produit P NATURAL JOIN Commande C
    GROUP BY P.nom HAVING COUNT(*)>10
```

Usage

```
SELECT DISTINCT M.nom FROM mv_agr M, Produit P
WHERE M.nom=P.nom AND M.prix<P.prix
```

Scénarios possibles

Il existe une vue matérialisée $V = R \bowtie S$

Requête $Q = R \bowtie S \bowtie T$

- Peut être remplacée par $Q = V \bowtie T$
- En fonction de l'estimation du coût

Requête $\sigma_{A=a}(V)$

- Il existe un index sur $R.A$, et
- un autre sur l'attribut de jointure $S.B$
- Le plan avantageux : $Q = \sigma_{A=a}(R) \bowtie S$

Les vues matérialisées : Pros & Cons

Avantages

- Définition simple
- Accélération des requêtes coûteuses
  - Production instantanée du résultat
- Réglages (index, regroupement, etc.) au même titre qu'une table

Inconvénients

- Mise à jour !
  - réévaluation périodique intégrale, ou
  - maintenance incrémentale, automatisée ou à définir à la main

Performant pour des requêtes coûteuses sur données stables

Reformuler la requête SQL

La réécriture de requête SQL

Tâche complexe

Interaction de :

- valeurs `NULL`
- doublons
- agrégation
- requêtes imbriquées

Bonne nouvelle

L'optimiseur fait une bonne part du boulot !

Huggy les bons tuyaux 1/3

Idée n°1

1 seul bloc d'optimisation partout où c'est a priori possible

Exemple

requête corrélée

```
SELECT E.nom FROM Enseignant E
WHERE EXISTS ( SELECT * FROM Enseigne N
               WHERE E.id=N.id AND N.annee=2016 )
```

requête aplatie

```
SELECT E.nom FROM Enseignant E, Enseigne N
WHERE E.id=N.id AND N.annee=2016
```

✎ Et avec un `NOT EXISTS` ?

Huggy les bons tuyaux 2/3

Idée n°2

Limiter l'usage du `DISTINCT`, notamment si le résultat contient une clé

Idée n°3

Limiter l'usage de la construction `HAVING` avec le `GROUP BY`. Préférer filtrer en amont dans la clause `WHERE` là où c'est possible

Idée n°4

Pour des requêtes imbriquées, préférer la formulation avec `EXISTS` plutôt que `IN` ou `COUNT()`

Huggy les bons tuyaux 3/3

Idée n°5

Proscrire la création de tables temporaires

Idée n°6

En présence d'index, écrire des blocs d'`UNION` (`ALL`) plutôt que des conditions `OR` dans la clause `WHERE`

Idée n°7

Énumérer la liste des attributs dans la clause `SELECT` plutôt que `*`

…

Bidouiller le schéma

Révision du schéma

Premier réflexe

Examen des formes normales alternatives du schéma

Rappel

Un schéma admet souvent plusieurs FNBC/3FN

Bricolages supplémentaires

- Fragmentation horizontale ou verticale
- Dé-normalisation : dégrader la forme normale de la BdD

Fragmentation verticale

Curriculum=

NSS	Nom	Adresse	Cv	Photo
123123	Alice	Nantes	Clob1...	Blob1...
234234	Bob	Paris	Clob2...	Blob2...
345345	Carole	Lyon	Clob3...	Blob3...
456456	David	Nantes	Clob4...	Blob4...

▼

T_1

NSS	Nom	Adresse
123123	Alice	Nantes
234234	Bob	Paris
...	...	...

T_2

NSS	Cv
123123	Clob1...
234234	Clob2...

T_3

NSS	Photo
123123	Blob1...
234234	Blob2...

Sur-décomposition au-delà de la FNBC/3FN !

Que devient la table originale ?

Besoin de préserver la table Cv pour certaines applications/requêtes ?

vue virtuelle

```
CREATE VIEW Cv AS
    SELECT T1.NSS, T1.Nom, T1.Adresse,
           T2.Cv,
           T3.Photo
    FROM T1
         NATURAL JOIN T2
         NATURAL JOIN T3
```

Du bon usage de la fragmentation verticale

Exemple

requête sur la vue

```
SELECT Adresse
FROM Cv
WHERE Nom='Alice'
```

Quelles sont les tables parmi T_1, T_2 et T_3 qui sont concernées ?

- Reformulation de la requête avec T_1 uniquement

requête sur un fragment

```
SELECT Adresse
FROM T1
WHERE Nom='Alice'
```

Les motivations 1/2

Quand doit-on utiliser la fragmentation verticale ?

1. La performance des requêtes

Lorsque peu d'attributs sont concernés :

- Accès disque uniquement pour ces colonnes
- Économie I/O substantielle pour des tables « larges »

Inconvénients

- Surcoût de stockage pour les répliques de la clé
- Jointures coûteuses pour reconstruire les n-uplets

Les motivations 2/2

2. Les « gros » attributs rarement sollicités
   - texte long, document
   - image, son
   - etc.
3. Les bases de données réparties
   - Infos personnelles sur un site,
   - Activité et profil sur un autre
4. L'intégration de données
   - T_1 provient d'une source
   - T_2 d'une autre

Fragmentation horizontale

Client=

NSS	Nom	Ville	Dpt
123123	Alice	Nantes	44
234234	Bob	Paris	75
345345	Carole	Lyon	69
456456	David	Rezé	44
567567	Eva	Paris	75
678678	Franck	Nantes	44

▶

ClientParis=

NSS	Nom	Ville	Dpt
234234	Bob	Paris	75
567567	Eva	Paris	75

ClientLyon=

NSS	Nom	Ville	Dpt
345345	Carole	Lyon	69

Client44=

NSS	Nom	Ville	Dpt
123123	Alice	Nantes	44
456456	David	Rezé	44
678678	Franck	Nantes	44

Recomposition et usage

vue virtuelle par union

```
CREATE VIEW Client AS
    ( ClientParis
        UNION ALL
      ClientLyon
        UNION ALL
      Client44 )
```

requête sur la vue

```
SELECT Nom
FROM Client
WHERE Ville='Paris'
```

Quelles sont les tables qui sont sollicitées ?!

Un peu mieux

requête révisée avec marqueurs

```
CREATE VIEW Client AS
    ( SELECT * FROM ClientParis WHERE Ville='Paris' )
        UNION ALL
    ( SELECT * FROM CLientLyon WHERE Ville='Lyon' )
        UNION ALL
    ( SELECT * FROM Client44 WHERE Dpt='44' OR
                                   Ville='Nantes' OR
                                   Ville='Rezé' )
```

- Besoin de « marqueurs » de fragmentation
- Techniques différentes selon le SGBD

Retour à l'usage

requête initiale

```
SELECT Nom
FROM Client
WHERE Ville='Paris'
```

▼

requête reformulée sur fragment

```
SELECT Nom
FROM ClientParis
```

Motivations pour la fragmentation horizontale

1. Performance

- Données d'activité, enregistrées par accumulation uniquement
  - 1 fragment par mois, *etc.*
- Données historisées et données actives
- Données volumineuses, avec une clé de répartition fréquemment invoquée dans les requêtes
  - Zone géographique, *etc.*

2. BdD réparties et parallèles

- Clé de fragmentation obtenue par hachage

3. Intégration de données

Révision du schéma

- Fragmentation
- Dé-normalisation

Les variantes de la dé-normalisation 1/2

1. Réifier un attribut calculé
   - `Disque(titre, dateSortie, label, durée)`
   - somme des durées de chaque piste
2. Ré-introduire la transitivité
   - `Employé(nom, fonction, batId, dptId)`
   - dépendance fonctionnelle `batId` $\rightarrow$ `dptId`
3. Promouvoir un attribut
   - `Employé(nom, fNom, batId)` à partir de `Fonction(fId, fNom)`
   - `Fonction(fNom)` comporte peu de modalités de grande taille

Les variantes de la dé-normalisation 2/2

4. Fusionner des tables
   - `Employé(nom, fonction, batId, bNom, bAdresse)`
   - pré-calcul de la jointure $\texttt{Employé} \bowtie \texttt{Bâtiment}$

5. Dupliquer un attribut
   - `Employé(nom, fonction, batId, bNom)`
   - forme faible de la fusion

Étude de cas

Exemple

Produit(**pId**, pNom, prix, fId)
Fournisseur(**fId**, fNom, ville)

Une requête très fréquente

```
SELECT P.pId, P.pNom
FROM Produit P
     NATURAL JOIN Fournisseur F
WHERE P.prix<? AND F.ville=?
```

- Quelles optimisations ?

Révision du schéma

Rappel

```
Produit(pId, pNom, prix, fId)
Fournisseur(fId, fNom, ville)
```

Dé-normalisation

```
ProduitFournisseur(pId, pNom, prix, fNom, ville)
```

Retour à la requête d'exemple

requête originale

```
SELECT P.pId, P.pNom
FROM Produit P
     NATURAL JOIN Fournisseur F
WHERE P.prix<? AND F.ville=?
```

requête sur schéma dénormalisé

```
SELECT pId, pNom
FROM ProduitFournisseur
WHERE prix<? AND ville=?
```

Les problèmes de la dé-normalisation

- entorse à la FNBC/3FN
  - fId $\rightarrow$ fNom, ville
- redondance
  - sur-poids
  - mise à jour de toutes les répliques d'une valeur
- anomalies
  - mise en œuvre de contrôles manuels (**trigger**, appli.)
- requêtes ciblées moins efficaces
  - la table dé-normalisée est plus coûteuse à balayer

Dé-normaliser ou pas ?

Redondance volontaire justifiée sous conditions

1. la performance est un objectif et la redondance y contribue
2. les DF responsables de la dé-normalisation sont documentées
3. les mécanismes de contrôle sont en place

Aperçu de la complexité du réglage

Pour accélérer une jointure fréquente :

- création d'index
- définition d'une vue matérialisée
- dé-normalisation
- stockage conjoint des enregistements des deux tables (localité spatiale)

Réglage d'une requête complexe

Il arrive qu'une seule requête concentre les efforts de tuning

Pour régler une requête

1. Examiner le plan d'exécution de l'optimiseur
   - `EXPLAIN PLAN`
   - relever les index, les algorithmes de jointure, les méthodes d'accès, **etc.**
2. Proposer de nouveaux index et vues matérialisées
3. Réviser le schéma de la base de données
4. Reformuler la requête

Effet secondaire Attention à ne pas pénaliser les autres transactions !