ÉDITION FRANÇAISE

SQL : AU CŒUR DES PERFORMANCES

COUVRE TOUTES LES BASES DE DONNÉES MAJEURES

TOUT CE QUE LES DÉVELOPPEURS DOIVENT SAVOIR À PROPOS DES PERFORMANCES EN SQL

MARKUS WINAND

Editeur (Medieninhaber/Verleger) :
Markus Winand

Maderspergerstasse 1-3/9/11
1160 Wien
AUSTRIA
<office@winand.at>

Conception de la couverture :
tomasio.design — Mag. Thomas Weninger — Wien — Austria

Photo couverture :
Brian Arnold — Turriff — UK

Traduction française :
Guillaume Lelarge — Longjumeau — France

Titre original :
SQL Performance Explained

Relecture :
Audrey Chiron — Vienne — Autriche

2013-03-03

SQL : Au cœur des performances

Tout ce que les développeurs doivent savoir à propos des performances en SQL

Markus Winand
Vienne, Autriche

Table des matières

Les développeurs ont besoin d'indexer

Les problèmes de performances en SQL sont aussi vieux que le langage SQL lui-même. Certains n'hésitent pas à dire que SQL est intrinsèquement lent. Même s'il a pu y avoir une part de vérité au tout début du SQL, ce n'est plus du tout vrai. Et pourtant, les problèmes de performances en SQL sont toujours d'actualité. Comment cela est-il possible ?

Le langage SQL est certainement le langage de programmation de quatrième génération ayant le plus de succès. Son principal intérêt se révèle dans sa capacité à séparer le *« quoi »* et le *« comment »*. Une requête SQL décrit le *besoin* sans donner d'instructions sur la *manière* de l'obtenir. Prenez cet exemple :

```
SELECT date_de_naissance
  FROM employes
 WHERE nom = 'WINAND'
```

La requête SQL se lit comme une phrase écrite en anglais qui explique les données réclamées. Écrire des requêtes SQL ne demande aucune connaissance sur le fonctionnement interne du système de base de données ou du système de stockage (disques, fichiers, etc.) Il n'est pas nécessaire d'indiquer à la base de données les fichiers à ouvrir ou la manière de trouver les lignes demandées. Beaucoup de développeurs ayant des années d'expérience avec SQL n'ont que très peu de connaissance sur le traitement qui s'effectue au niveau du système de bases de données.

La séparation du besoin et de la façon de l'obtenir fonctionne très bien en SQL mais cela ne veut pas dire pour autant que tout est parfait. Cette abstraction atteint ses limites quand des performances importantes sont attendues : la personne ayant écrit une requête SQL n'accorde pas d'importance sur la *façon* dont la base de données exécute la requête. En conséquence, cette personne ne se sent pas responsable d'une exécution lente. L'expérience montre le contraire. Afin d'éviter des problèmes de performance, cette personne doit connaître un peu le système de bases de données.

Il s'avère que la seule chose que les *développeurs* doivent connaître est l'indexation. En fait, l'indexation d'une base de données est un travail de développeurs car l'information la plus importante pour une bonne indexation ne se situe ni au niveau de la configuration du système de stockage ni dans la configuration du matériel, mais plutôt au niveau de l'application : « comment l'application cherche ses données ». Cette information n'est pas facilement accessible aux administrateurs de bases de données ou aux consultants externes. Il est parfois nécessaire de récupérer cette information en surveillant l'application. Par contre, les développeurs ont cette information facilement.

Ce livre couvre tout ce que les développeurs doivent savoir sur les index, et rien de plus. Pour être plus précis, ce livre couvre seulement le type d'index le plus important : l'*index B-tree*.

L'index B-tree fonctionne de façon pratiquement identique sur la plupart des bases de données. Ce livre utilise uniquement la terminologie de la base de données Oracle®, mais les principes s'appliquent tout aussi bien aux autres moteurs de bases de données. Des notes fournissent des informations spécifiques aux bases de données MySQL, PostgreSQL et SQL Server®.

La structure de ce livre est conçue spécifiquement pour les développeurs ; la plupart des chapitres correspond à une partie distincte d'une requête SQL.

CHAPITRE 1 - Anatomie d'un index

Le premier chapitre est le seul qui ne couvre pas SQL. Il se focalise sur la structure fondamentale d'un index. Une compréhension de la structure des index est essentielle pour continuer. Ne sous-estimez pas ce chapitre, il est essentiel !

Bien que ce chapitre soit plutôt court, environ huit pages, vous verrez après l'avoir parcouru que vous comprendrez mieux le phénomène des index lents.

CHAPITRE 2 - La clause Where

Ce chapitre explique tous les aspects de la clause **`where`**, des recherches simples sur une seule colonne aux recherches complexes avec des intervalles et des cas spéciaux comme `LIKE`.

Ce chapitre est le cœur de ce livre. Une fois que vous aurez appris à utiliser ces techniques, vous devriez écrire des requêtes SQL bien plus performantes.

CHAPITRE 3 - Performance et scalabilité
: Ce chapitre est une petite digression sur les mesures des performances et sur la scalabilité des bases de données. Cela vous permettra de comprendre pourquoi ajouter plus de matériel n'est pas la meilleure solution aux requêtes lentes.

CHAPITRE 4 - L'opération de jointure
: Retour au SQL : ici, vous trouverez une explication sur l'utilisation des index pour obtenir des jointures de tables rapides.

CHAPITRE 5 - Regrouper les données
: Vous êtes-vous déjà demandé s'il y avait une différence entre sélectionner une seule colonne ou toutes les colonnes ? Ce chapitre vous apportera cette réponse, ainsi qu'une astuce pour obtenir encore plus de performances.

CHAPITRE 6 - Trier et grouper
: Même **order by** et **group by** peuvent utiliser des index.

CHAPITRE 7 - Résultats partiels
: Ce chapitre explique comment bénéficier d'une exécution sérialisée si vous n'avez pas besoin de l'ensemble complet des résultats.

CHAPITRE 8 - INSERT, DELETE et UPDATE
: Comment les index influencent-ils les performances en écriture ? Les index ne sont pas gratuits, utilisez-les avec précaution !

ANNEXE A - Plans d'exécution
: Demandez à la base de données comment elle exécute une requête.

Chapitre 1

Anatomie d'un index

« Un index accélère la requête » est l'explication la plus simple que j'ai pu voir pour un index. Bien qu'elle décrive parfaitement l'aspect le plus important d'un index, cela n'est malheureusement pas suffisant pour ce livre. Ce chapitre décrit la structure d'un index d'une façon moins superficielle, mais sans trop entrer dans les détails. Il fournit suffisamment d'informations pour comprendre tous les aspects relatifs aux performances en SQL, aspects qui seront détaillés tout au long de ce livre.

Un index est une structure séparée dans la base de données, construite en utilisant l'instruction **`create index`**. Il nécessite son propre espace sur le disque et détient une copie des données de la table. Cela signifie qu'un index est une redondance. Créer un index ne modifie pas les données de la table. Cela crée une nouvelle structure de données faisant référence à la table. En fait, un index de base de données ressemble très fortement à l'index d'un livre : il occupe de la place, il est redondant et il fait référence aux informations réelles stockées ailleurs.

Index clusterisés

SQL Server et MySQL (en utilisant InnoDB) ont une vue plus large de ce qu'est un « *index* ». Ils font références à des tables qui ont une structure d'index en tant que *index clusterisé*. Ces tables sont appelées des tables organisées en index (en anglais, « Index-Organized Tables » ou IOT) dans la base de données Oracle.

Chapitre 5, « *Regrouper les données* » les décrit avec plus de détails et explique leur avantages et inconvénients.

Rechercher dans un index de base de données ressemble à rechercher dans un annuaire téléphonique. L'idée est que toutes les informations sont rangées dans un ordre bien défini. Trouver des informations dans un ensemble de données triées est rapide et simple car l'ordre de tri détermine la position de chaque donnée.

Néanmoins, un index de base de données est plus complexe qu'un annuaire car l'index est constamment modifié. Mettre à jour un annuaire imprimé pour chaque changement est impossible car il n'y a pas d'espace entre chaque information pour en ajouter de nouvelles. Un annuaire imprimé contourne ce problème en gérant des mises à jours accumulées lors de la prochaine impression. Une base de données ne peut pas attendre aussi longtemps. Elle doit traiter les commandes **INSERT**, **DELETE** et **UPDATE** immédiatement, tout en conservant l'ordre de l'index sans déplacer de gros volumes de données.

La base de données combine deux structures de données pour parvenir à ce résultat : une liste doublement chaînée et un arbre de recherche. Ces deux structures expliquent la plupart des caractéristiques de performances des bases de données.

LES NŒUDS FEUILLES D'UN INDEX

Le but principal d'un index est de fournir une représentation ordonnée des données indexées. Néanmoins, il n'est pas possible de stocker les données séquentiellement car une commande **INSERT** nécessiterait le déplacement des données suivantes pour faire de la place à la nouvelle donnée. Déplacer de gros volumes de données demande beaucoup de temps, ce qui causerait des lenteurs importantes pour une commande **INSERT**. La solution à ce problème revient à établir un ordre logique qui est indépendant de l'ordre physique en mémoire.

L'ordre logique est établi grâce à une liste doublement chaînée. Chaque nœud a un lien vers les deux nœuds voisins, un peu comme une chaîne. Les nouveaux nœuds sont insérés entre deux nœuds existants en mettant à jour leurs liens pour référencer le nouveau nœud. L'emplacement physique du nouveau nœud n'a aucune importance car la liste doublement chaînée maintient l'ordre logique.

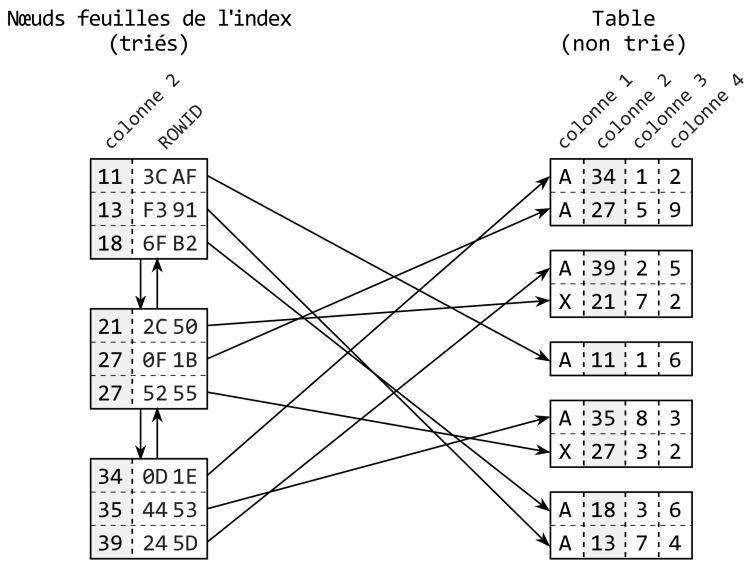
Cette structure de données est appelée une *liste doublement chaînée* car chaque nœud fait référence aux nœuds précédent et suivant. La base de données peut ainsi lire l'index en avant et en arrière suivant le besoin. Il est du coup possible d'insérer de nouvelles entrées sans déplacer de gros volumes de données, seuls les pointeurs sont changés.

Les listes doublement chaînées sont aussi utilisées pour les collections (conteneurs) dans de nombreux langages de développement.

Langage de programmation	Nom
Java	java.util.LinkedList
Framework .NET	System.Collections.Generic.LinkedList
C++	std::list

Les bases de données utilisent des listes doublement chaînées pour connecter des *nœuds feuilles* d'index. Chaque nœud feuille est stocké dans un *bloc* de la base de données (aussi appelé *page*), autrement dit, la plus petite unité de stockage de la base de données. Tous les blocs d'index sont de la même taille, généralement quelques kilo-octets. La base de données utilise l'espace dans chaque bloc du mieux possible, et stocke autant d'entrées d'index que possible dans chaque bloc. Cela signifie que l'ordre de l'index est maintenu sur deux niveaux différents : les enregistrements de l'index à l'intérieur de chaque nœud feuille, et les nœuds feuilles entre elles en utilisant une liste doublement chaînée.

Figure 1.1. Nœuds feuilles de l'index et données de la table

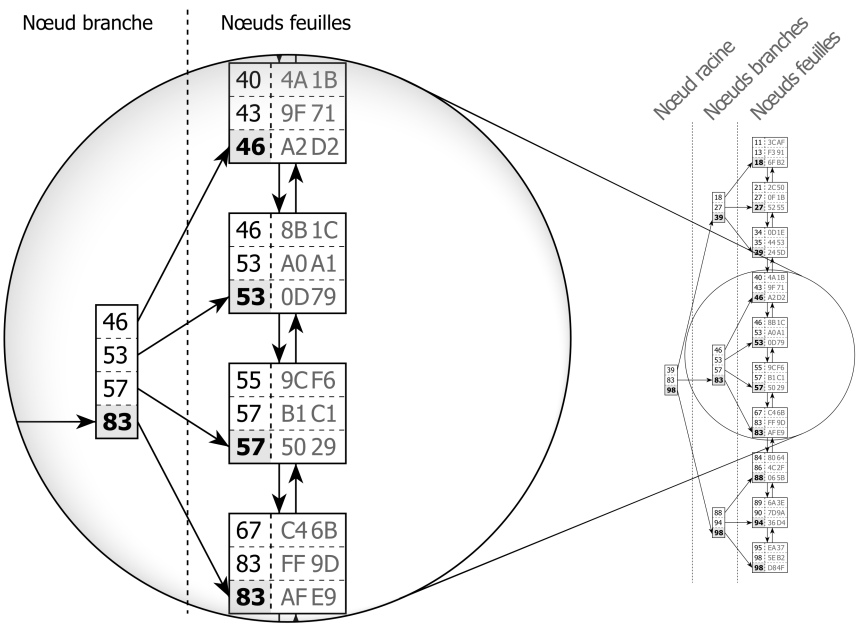


La Figure 1.1 illustre les nœuds feuilles de l'index et leur connexion aux données de la table. Chaque enregistrement de l'index consiste en des colonnes indexées (la clé, la colonne 2) et fait référence à la ligne correspondante dans la table (via `ROWID` ou `RID`). Contrairement à l'index, les données de la table sont stockées dans une structure appelée heap et n'est pas du tout triée. Il n'existe aucune relation entre les lignes stockées dans le même bloc de table, pas plus qu'il n'y a de connexions entre les blocs.

L'ARBRE DE RECHERCHE (B-TREE)

Les nœuds feuilles de l'index sont stockés dans un ordre arbitraire en ce sens que la position sur le disque ne correspond pas à la position logique suivant l'ordre de l'index. C'est comme un annuaire téléphonique avec les pages mélangées. Si vous recherchez « Dupont » mais que vous ouvrez l'annuaire à « Canet », il n'est pas garanti que Dupont suive Canet. Une base de données a besoin d'une deuxième structure pour trouver rapidement une donnée parmi des pages mélangées : un *arbre de recherche équilibré* (en anglais, un « Balanced Tree Search ») ou, plus court, un B-tree.

Figure 1.2. La structure du B-tree



La Figure 1.2 montre un exemple d'index avec 30 entrées. La liste doublement chaînée établit l'ordre logique entre les nœuds feuilles. La racine et les nœuds branches permettent une recherche rapide parmi les nœuds feuilles.

Le graphique distingue le nœud branche et les nœuds feuilles auxquels il fait référence. Chaque entrée de nœud branche correspond à la valeur la plus grosse dans le nœud feuille référencé. Par exemple, la valeur 46 dans

le premier nœud feuille pour que la première entrée du nœud branche soit aussi 46. Ceci est toujours vrai pour les autres nœuds feuilles. À la fin, le nœud branche a les valeurs 46, 53, 57 et 83. Suivant cette méthode, un niveau de branche est construit jusqu'à ce que tous les nœuds feuilles soient couverts par un nœud branche.

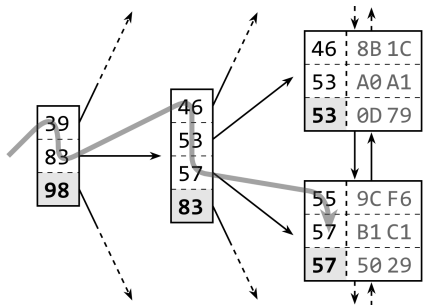
Le prochain niveau est construit de façon similaire, mais au-dessus du premier niveau de branche. La procédure se répète jusqu'à ce que toutes les clés remplissent un nœud seul, le *nœud racine*. La structure est un *arbre de recherche équilibré* car la profondeur de l'arbre est identique à chaque position. La distance entre le nœud racine et les nœuds feuilles est identique partout.

Note

Un B-tree est un arbre équilibré — pas un arbre binaire.

Une fois créée, la base de données maintient automatiquement l'index. Elle applique chaque commande **INSERT**, **DELETE** et **UPDATE** à l'index et conserve la propriété équilibrée de l'arbre, ce qui cause une surcharge de maintenance pour les opérations d'écriture. Le Chapitre 8, « *Modifier les données* » explique cela en détails.

Figure 1.3. Parcours d'un B-Tree



La Figure 1.3 montre un fragment d'index pour illustrer une recherche sur la clé 57. Le parcours de l'arbre commence au nœud racine du côté gauche. Chaque entrée est traitée dans l'ordre ascendant jusqu'à ce qu'une valeur identique ou plus grande (>=) que la valeur recherchée (57) soit trouvée. Dans ce graphique, il s'agit de la valeur 83. La base de données suit la référence au nœud branche correspondant et répète la même procédure jusqu'à ce que le parcours atteigne un nœud feuille.

IMPORTANT

L'index B-tree permet de trouver un nœud feuille rapidement.

Le parcours de l'arbre est une opération très efficace. C'est même si efficace que j'en parle comme de la *première puissance de l'indexation*. C'est presque instantané, y compris sur de très gros volumes de données. Ceci est principalement dû à la propriété équilibrée d'un arbre, ce qui permet d'accéder à tous les éléments avec le même nombre d'étapes, mais aussi au grossissement logarithmique de la profondeur de l'arbre. Cela signifie que la profondeur de l'arbre grossit très lentement en comparaison au nombre de nœuds feuilles. De vrais index avec des millions d'enregistrements ont une profondeur d'arbre de quatre ou cinq. Il est très rare de rencontrer une profondeur de cinq ou six dans un arbre. L'encart « Complexité logarithmique » décrit cela en détails.

INDEX LENTS, PARTIE I

Malgré l'efficacité du parcours de l'arbre, il existe des cas où une recherche via l'index ne sera pas aussi rapide que souhaité. Cette contradiction est la raison d'être du mythe de l'*index dégénéré*. Le mythe affirme qu'une reconstruction d'index est la solution miracle. La vraie raison pour laquelle des requêtes basiques peuvent être lentes, même en utilisant un index, peut se trouver sur les bases des sections précédentes.

Le premier ingrédient rendant une recherche lente via l'index est la chaîne de nœuds feuilles. Considérez de nouveau la recherche de la valeur 57 dans la Figure 1.3. Il existe deux entrées correspondantes dans l'index. Au moins deux entrées sont identiques, pour être plus précis : le nœud feuille suivant pourrait contenir de nouvelles entrées 57. La base de données *doit* lire le prochain nœud feuille pour savoir s'il existe des entrées correspondantes. Cela signifie qu'une recherche d'index doit non seulement faire le parcours de l'arbre, mais il doit aussi suivre la chaîne des nœuds feuilles.

Le deuxième ingrédient pour une recherche d'index lente est d'avoir à accéder à la table. Même un nœud feuille simple peut contenir plusieurs fois la valeur recherchée, parfois même des centaines de fois. Les données correspondantes de la table sont généralement réparties sur un grand nombre de blocs (voir la Figure 1.1, « Nœuds feuilles de l'index et données de la table »). Cela signifie qu'il y a un accès supplémentaire à la table pour chaque valeur trouvée dans l'index.

Complexité logarithmique

En mathématique, le logarithme d'un nombre sur une base donnée est la puissance ou l'exposant avec laquelle la base doit être élevée pour produire le résultat [Wikipedia[1]].

Dans un arbre de recherche, la base correspond au nombre d'entrées par nœud branche et l'exposant à la profondeur de l'arbre. L'index en exemple dans Figure 1.2 contient jusqu'à quatre entrées par nœud et a une profondeur d'arbre de 3. Cela signifie que l'index peut contenir jusqu'à 64 entrées (4^3). S'il grossit d'un niveau, il peut déjà contenir 256 entrées (4^4). À chaque *ajout* d'un niveau, le nombre maximum d'entrées *quadruple*. Le logarithme inverse cette fonction. La profondeur de l'arbre est donc $\log_4$(nombre-d-entrées-index).

L'augmentation logarithmique permet de rechercher parmi un million d'entrées dans dix niveaux de l'arbre, mais un index réel est encore plus efficace. Le facteur principal qui influe sur la profondeur de l'arbre, et du coup ces performances, est le nombre d'entrées dans chaque nœud de l'arbre. Ce nombre correspond, mathématiquement, à la base du logarithme. Plus grande est la base, plus large sera l'arbre et plus rapide sera le parcours.

Profondeur de l'arbre	Entrées d'index
3	64
4	256
5	1,024
6	4,096
7	16,384
8	65,536
9	262,144
10	1,048,576

Les bases de données exposent ce concept le plus possible et placent autant d'entrées que possible dans chaque nœud, souvent des centaines. Cela signifie que chaque nouveau niveau d'index supporte une centaine de fois plus d'entrées.

[1] http://en.wikipedia.org/wiki/Logarithm

Une recherche dans un index suit trois étapes : (1) le parcours de l'arbre ; (2) la suite de la chaîne de nœuds feuilles ; (3) la récupération des données de la table. Le parcours de l'arbre est la seule étape qui accède à un nombre limité de blocs, qui correspond à la profondeur de l'index. Les deux autres étapes doivent avoir accès à de nombreux blocs. Elles sont la cause des lenteurs lors d'une recherche par index.

L'origine du mythe des index lents est la croyance erronée qu'une recherche d'index ne fait que parcourir l'arbre, et donc l'idée qu'un index lent est causé par un arbre « cassé » ou « non équilibré ». En fait, vous pouvez demander à la plupart des bases de données comment elles utilisent un index. La base de données Oracle est assez verbeuse sur cet aspect. Elle a trois opérations distinctes qui décrivent une recherche basique par index :

INDEX UNIQUE SCAN
: Un `INDEX UNIQUE SCAN` réalise seulement le parcours de l'arbre. La base de données Oracle utilise cette opération sur une contrainte unique qui assure que le critère de recherche correspondra à une seule entrée.

INDEX RANGE SCAN
: Un `INDEX RANGE SCAN` fait le parcours de l'arbre *et* suit la chaîne de nœuds feuilles pour trouver toutes les entrées correspondantes. C'est l'opération la plus sûre si le critère peut se révéler vrai pour plusieurs entrées.

TABLE ACCESS BY INDEX ROWID
: Une opération `TABLE ACCESS BY INDEX ROWID` récupère la ligne de la table. Cette opération est (souvent) réalisée pour chaque enregistrement récupéré à partir d'un précédent parcours d'index.

Le point important est qu'un `INDEX RANGE SCAN` peut potentiellement lire une large part d'un index. S'il existe plus d'un accès de table pour chaque ligne, la requête peut devenir lente même en utilisant un index.

CHAPITRE 2

LA CLAUSE WHERE

Le chapitre précédent a décrit la structure des index et a expliqué la cause des mauvaises performances avec les index. Dans ce nouveau chapitre, nous apprendrons comment trouver et éviter ces problèmes dans les requêtes SQL. Nous allons commencer en regardant la clause **`where`**.

La clause **`where`** définit la condition de recherche d'une requête SQL et, de ce fait, elle tombe dans le domaine fonctionnel principal d'un index : trouver des données rapidement. Bien que la clause **`where`** ait un impact important sur les performances, elle est souvent mal écrite, si bien que la base de données doit parcourir une grande partie de l'index. Le résultat : une clause **`where`** mal écrite est la première raison d'une requête lente.

Ce chapitre explique comment les différents opérateurs affectent l'utilisation d'un index et comment s'assurer qu'un index est utilisable pour autant de requêtes que possible. La dernière section montre des contre-exemples et proposent des alternatives qui donnent de meilleures performances.

L'OPÉRATEUR D'ÉGALITÉ

L'opérateur d'égalité est l'opérateur SQL le plus trivial et le plus fréquemment utilisé. Les erreurs d'indexation qui affectent les performances sont toujours très communes et les clauses **`where`** qui combinent plusieurs conditions sont particulièrement vulnérables.

Cette section montre comment vérifier l'utilisation de l'index et explique comment les index concaténés peuvent optimiser les conditions combinées. Pour faciliter la compréhension, nous allons analyser une requête lente pour voir l'impact réel des causes expliquées dans le Chapitre 1.

CLÉS PRIMAIRES

Nous commençons avec la clause **where** la plus simple et la plus commune : la recherche d'une clé primaire. Pour les exemples de ce chapitre, nous utilisons la table EMPLOYES qui se définie ainsi :

```
CREATE TABLE employes (
   id_employe        NUMBER         NOT NULL,
   prenom            VARCHAR2(1000) NOT NULL,
   nom               VARCHAR2(1000) NOT NULL,
   date_de_naissance DATE           NOT NULL,
   numero_telephone  VARCHAR2(1000) NOT NULL,
   CONSTRAINT employes_pk PRIMARY KEY (id_employe)
);
```

La base de données crée automatiquement un index pour la clé primaire. Cela signifie qu'il existe un index sur la colonne ID_EMPLOYE, même si nous n'avons pas exécuté de commande **create index**.

La requête suivante utilise la clé primaire pour récupérer le nom d'un employé :

```
SELECT prenom, nom
  FROM employes
 WHERE id_employe = 123
```

La clause **where** ne peut pas correspondre à plusieurs enregistrements car la clé primaire assure l'unicité des valeurs de la colonne ID_EMPLOYE. La base de données n'a pas besoin de suivre les nœuds feuilles de l'index. Il suffit de parcourir l'arbre de l'index. Nous pouvons utiliser le *plan d'exécution* pour vérifier :

```
-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYES     |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN          | EMPLOYES_PK  |    1 |    1 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ID_EMPLOYE"=123)
```

Le plan d'exécution Oracle affiche un `INDEX UNIQUE SCAN`, ce qui correspond à l'opération qui ne fait que parcourir l'arbre de l'index. Il utilise complètement la complexité logarithmique de l'index pour trouver l'entrée très rapidement, pratiquement indépendamment de la taille de la table.

Astuce

Le *plan d'exécution* (quelque fois appelé *plan explain* ou *plan de la requête*) montre les étapes réalisées par la base pour exécuter une requête SQL. L'Annexe A à la page 165 explique comment récupérer et lire les plans d'exécution avec d'autres bases de données.

Après avoir accédé à l'index, la base de données doit réaliser une autre étape pour récupérer les données demandées (`PRENOM`, `NOM`) à partir du stockage de la table : il s'agit de l'opération `TABLE ACCESS BY INDEX ROWID`. Cette opération peut devenir un goulet d'étranglement, comme expliqué dans la « Index lents, partie I », mais ce risque est inexistant avec un `INDEX UNIQUE SCAN`. Cette opération ne peut pas renvoyer plus d'une entrée donc elle ne peut pas déclencher plus d'un accès à la table. Cela signifie que les ingrédients d'une requête lente ne sont pas présents avec un `INDEX UNIQUE SCAN`.

Clés primaires sans index unique

Une clé primaire n'a pas nécessairement besoin d'un index unique. Vous pouvez aussi utiliser un index non unique. Dans ce cas, la base de données Oracle n'utilise pas l'opération `INDEX UNIQUE SCAN` mais utilise à la place l'opération `INDEX RANGE SCAN`. Néanmoins, la contrainte maintiendra l'unicité des clés pour que la recherche dans l'index renvoie au plus une entrée.

Les *contraintes différables* sont une des raisons pour utiliser des index non uniques pour les clés primaires. En opposition aux contraintes standards, qui sont validées lors de l'exécution de la requête, la base de données repousse la validation des contraintes différables jusqu'au moment où la transaction est validée. Les contraintes différées sont requises pour insérer des données dans des tables ayant des dépendances circulaires.

INDEX CONCATÉNÉS

Même si la base de données crée automatiquement l'index pour la clé primaire, il est toujours possible de réaliser des améliorations manuelles si la clé contient plusieurs colonnes. Dans ce cas, la base de données crée un index sur toutes les colonnes de la clé primaire, ce qu'on appelle un index *concaténé* (aussi connu sous le nom d'index *multi-colonnes*, *composite* ou *combiné*). Notez que l'ordre des colonnes d'un index concaténé a un grand impact sur ses capacités à être utilisé. Donc l'ordre des colonnes doit être choisi avec attention.

Pour les besoins de la démonstration, supposons qu'il y ait une fusion de sociétés. Les employés de l'autre société sont ajoutés à notre table `EMPLOYES` qui devient alors dix fois plus grosses. Il y a ici un problème : la colonne `ID_EMPLOYE` n'est pas unique entre les deux sociétés. Nous avons besoin d'étendre la clé primaire par un identifiant supplémentaire. Du coup, la nouvelle clé primaire a deux colonnes : la colonne `ID_EMPLOYE` comme auparavant et la colonne `ID_SUPPLEMENTAIRE` pour rétablir l'unicité.

L'index pour la nouvelle clé primaire est donc défini de la façon suivante :

```
CREATE UNIQUE INDEX employe_pk
    ON employes (id_employe, id_supplementaire);
```

Une requête pour un employé en particulier doit prendre en compte la clé primaire complète. Autrement dit, la colonne `ID_SUPPLEMENTAIRE` doit aussi être utilisée :

```
SELECT prenom, nom
  FROM employes
 WHERE id_employe        = 123
   AND id_supplementaire = 30;

-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYES     |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN          | EMPLOYES_PK  |    1 |    1 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ID_EMPLOYE"=123 AND "ID_SUPPLEMENTAIRE"=30)
```

Quand une requête utilise la clé primaire complète, la base de données peut utiliser une opération INDEX UNIQUE SCAN, quel que soit le nombre de colonnes de l'index. Mais qu'arrive-t-il quand seulement une des colonnes clés, par exemple, est utilisée pour rechercher tous les employés d'une société ?

```
SELECT prenom, nom
  FROM employes
 WHERE id_supplementaire = 20;

-------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
-------------------------------------------------
|  0 | SELECT STATEMENT  |           |  106 |  478 |
|* 1 |  TABLE ACCESS FULL| EMPLOYES  |  106 |  478 |
-------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ID_SUPPLEMENTAIRE"=20)
```

Le plan d'exécution révèle que la base de données ne doit pas utiliser l'index. À la place, il réalise un FULL TABLE SCAN. En résultat, la base de données lit la table entière et évalue chaque ligne par rapport à la clause **where**. La durée d'exécution grandit avec la taille de la table : si la table décuple, l'opération FULL TABLE SCAN prendra dix fois plus longtemps. Le danger de cette opération est qu'elle est souvent suffisamment rapide dans un petit environnement de développement, mais elle cause de sérieux problèmes de performance en production.

PARCOURS COMPLET DE TABLE

L'opération TABLE ACCESS FULL, aussi connue sous le nom de *parcours complet de table*, peut être l'opération la plus efficace dans certains cas, en particulier quand il faut récupérer une large portion de la table.

Ceci est dû en partie à la surcharge pour une recherche dans l'index, ce qui n'arrive pas pour une opération TABLE ACCESS FULL. En fait, quand une recherche par index lit un bloc après l'autre, la base de données ne sait pas quel bloc sera lu après tant que le traitement du bloc courant n'est pas terminé. Un FULL TABLE SCAN doit lire la table complète de toute façon, donc la base de données peut lire de plus gros morceaux à la fois (*lecture multi-blocs*). Bien que la base de données lise plus de données, il pourrait être nécessaire d'exécuter moins d'opérations de lecture.

La base de données n'utilise pas l'index car elle ne peut pas utiliser une colonne à partir d'un index concaténé. C'est beaucoup plus clair en faisant plus attention à la structure de l'index.

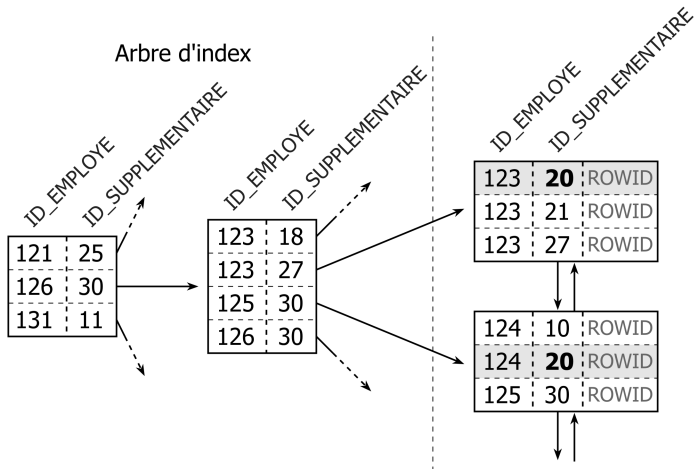
Un index concaténé est un index B-tree comme n'importe quel autre, qui conserve les données indexées dans une liste triée. La base de données considère chaque colonne suivant sa position dans la définition de l'index pour trier les enregistrements. La première colonne est le critère principal de tri et la deuxième colonne détermine l'ordre seulement si deux enregistrements ont la même valeur dans la première colonne. Et ainsi de suite.

IMPORTANT

Un index concaténé est *un index sur plusieurs colonnes*.

L'ordre d'un index à deux colonnes ressemble à l'ordre d'un annuaire téléphonique : il est tout d'abord trié par le nom, puis par le prénom. Cela signifie qu'un index à deux colonnes ne permet pas une recherche sur la deuxième colonne seule. Cela reviendrait à rechercher dans un annuaire en ayant seulement le prénom.

Figure 2.1. Index concaténé



L'exemple d'index dans la Figure 2.1 montre que les enregistrements pour l'identifiant supplémentaire 20 ne sont pas stockés les uns à côté des autres. On remarque également qu'il n'y a pas d'enregistrements pour lesquels `ID_SUPPLEMENTAIRE = 20` dans l'arbre, bien qu'ils existent dans les nœuds feuilles. Du coup, l'arbre est inutile pour cette requête.

ASTUCE

Visualiser un index aide à comprendre les requêtes supportées par cet index. Vous pouvez exécuter une requête sur la base de données pour retrouver tous les enregistrements dans l'ordre de l'index (syntaxe SQL:2008, voir la page 144 pour des solutions propriétaires utilisant `LIMIT`, `TOP` ou `ROWNUM`) :

```
SELECT <COLONNES LISTE D'INDEX>
  FROM <TABLE>
 ORDER BY <COLONNES LISTE D'INDEX>
 FETCH FIRST 100 ROWS ONLY;
```

Si vous placez la définition de l'index et le nom de la table dans la requête, vous obtiendrez un extrait de l'index. Demandez-vous si les lignes demandées sont groupées en un point central. Dans le cas contraire, l'arbre de l'index ne peut pas vous aider à trouver cette place.

Bien sûr, nous pouvons ajouter un autre index sur `ID_SUPPLEMENTAIRE` pour améliorer la rapidité de la requête. Néanmoins, il existe une meilleure solution, tout du moins si nous supposons que la recherche sur `ID_EMPLOYE` seul n'a pas de sens.

Nous pouvons profiter du fait que la première colonne de l'index est toujours utilisable pour la recherche. Encore une fois, cela fonctionne comme un annuaire téléphonique : vous n'avez pas besoin de connaître le prénom pour chercher le nom. L'astuce revient donc à inverser l'ordre des colonnes de l'index pour que `ID_SUPPLEMENTAIRE` soit en première position :

```
CREATE UNIQUE INDEX EMPLOYES_PK
    ON EMPLOYES (ID_SUPPLEMENTAIRE, ID_EMPLOYE);
```

Les deux colonnes ensemble sont toujours uniques. Ainsi les requêtes sur la clé primaire complète peuvent toujours utiliser un `INDEX UNIQUE SCAN` mais la séquence des enregistrements d'index est complètement différente. La colonne `ID_SUPPLEMENTAIRE` est devenue le critère principal de tri. Cela signifie que tous les enregistrements pour une société sont consécutifs dans l'index, et la base de données peut utiliser l'index B-tree pour trouver leur emplacement.

IMPORTANT

Le point le plus important à considérer lors de la définition d'un index concaténé est le choix de l'ordre des colonnes pour qu'il puisse supporter autant de requêtes SQL que possible.

Le plan d'exécution confirme que la base de données utilise l'index inversé. La colonne `ID_SUPPLEMENTAIRE` seule n'est plus unique, donc la base de données doit suivre les nœuds feuilles pour trouver tous les enregistrements correspondants : du coup, il utilise une opération `INDEX RANGE SCAN`.

```
-------------------------------------------------------------
|Id |Operation                    | Name        | Rows | Cost |
-------------------------------------------------------------
| 0 |SELECT STATEMENT             |             |  106 |   75 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYES    |  106 |   75 |
|*2 |  INDEX RANGE SCAN           | EMPLOYE_PK  |  106 |    2 |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ID_SUPPLEMENTAIRE"=20)
```

En général, une base de données peut utiliser un index concaténé lors de la recherche des colonnes les plus à gauche. Un index à trois colonnes peut être utilisé pour une recherche sur la première colonne, pour une recherche sur les deux premières colonnes et pour une recherche sur toutes les colonnes.

Même si la solution à deux index délivre des performances très bonnes pour les **SELECT**, la solution à un seul index est préférable. Non seulement cela permet d'économiser de l'espace de stockage, mais aussi la surcharge due à la maintenance pour le deuxième index. Moins une table a d'index, meilleures sont les performances des commandes **INSERT**, **DELETE** et **UPDATE**.

Pour définir un index optimal, vous devez comprendre un peu plus que le simple fonctionnement des index. Vous devez également savoir comment l'application demande les données, c'est-à-dire connaître les combinaisons de colonnes qui apparaissent dans les clauses **WHERE**.

Du coup, définir un index optimal est très difficile pour des consultants externes car ils n'ont pas de vue globale des chemins d'accès de l'application. Les consultants peuvent généralement prendre en compte une seule requête. Ils n'exploitent pas les bénéfices supplémentaires qu'un index peut fournir pour les autres requêtes. Les administrateurs de bases de données sont dans une position similaire. Ils peuvent connaître le schéma de la base de données mais ils n'ont pas de connaissances étendues des chemins d'accès.

La seule personne qui doit avoir la connaissance technique de la base de données et la connaissance fonctionnelle du métier est le développeur. Les développeurs ont une idée des données et connaissent les chemins d'accès aux données. Ils peuvent indexer les données correctement de telle manière à obtenir les meilleures performances pour l'application complète sans trop d'efforts.

INDEX LENTS, PARTIE II

La section précédente a expliqué comment gagner en performance à partir d'un index existant en changeant l'ordre de ses colonnes mais l'exemple portait seulement sur deux requêtes. Néanmoins, changer un index pourrait affecter toutes les requêtes sur la table indexée. Cette section explique comment les bases de données choisissent un index et démontre les effets de bord possibles dû à un changement dans la définition d'un index.

L'index `EMPLOYE_PK` adopté améliore les performances de toutes les requêtes qui cherchent par société seulement. Il est aussi utilisable pour toutes les requêtes qui cherchent par `ID_SUPPLEMENTAIRE` et par tout autre critère de recherche supplémentaire. Cela signifie que l'index devient utilisable pour les requêtes qui utilisaient un autre index pour une autre partie de la clause **`where`**. Dans ce cas, si plusieurs chemins d'accès sont possibles, c'est à l'optimiseur de choisir le meilleur.

L'OPTIMISEUR DE REQUÊTES

L'optimiseur de requêtes, ou le planificateur de requêtes, est le composant de la base de données qui transforme une requête SQL en un plan d'exécution. Ce processus est aussi connu sous le nom de *compilation* ou *analyse*. Il existe deux types distincts d'optimiseurs.

Un *optimiseur basé sur le coût* (CBO) génère un grand nombre de plans d'exécution et calcule un *coût* pour chaque plan. Le calcul du coût est basé sur les opérations utilisées et les estimations de nombres de lignes. À la fin, la valeur du coût sert d'information de base pour choisir le « meilleur » plan d'exécution.

Un *optimiseur basé sur des règles* (RBO) génère le plan d'exécution utilisant un ensemble de règles codées en dur. Ce type d'optimiseur est moins flexible et très peu utilisé de nos jours.

Changer un index peut aussi avoir des effets de bord déplaisants. Dans notre exemple, l'application du répertoire téléphonique est devenue très lente depuis la fusion. La première analyse a identifié la requête suivante comme cause principale des lenteurs.

```
SELECT prenom, nom, id_supplementaire, numero_telephone
  FROM employes
 WHERE nom = 'WINAND'
   AND id_supplementaire = 30;
```

Le plan d'exécution est le suivant :

Exemple 2.1. Plan d'exécution avec l'index de clé primaire revu

```
-----------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-----------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |   30 |
|*1 | TABLE ACCESS BY INDEX ROWID| EMPLOYES     |    1 |   30 |
|*2 |  INDEX RANGE SCAN           | EMPLOYES_PK  |   40 |    2 |
-----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - filter("NOM"='WINAND')
  2 - access("ID_SUPPLEMENTAIRE"=30)
```

Le plan d'exécution utilise un index. Il a un coût global de 30. Pour l'instant, tout va bien. Néanmoins, il est étonnant de voir qu'il utilise l'index que nous venons de changer. C'est une raison suffisante pour suspecter que le changement d'index a causé le problème de performances, tout particulièrement si on se rappelle de l'ancienne définition de l'index. Elle commençait avec la colonne `ID_EMPLOYE` qui ne fait pas du tout partie de la clause **`where`**. La requête ne pouvait pas utiliser cet index avant.

Pour aller plus loin dans l'analyse, il serait bon de comparer le plan d'exécution avant et après changement. Pour obtenir le plan d'exécution original, nous pourrions remettre en place l'ancien index. Néanmoins, la plupart des bases de données offre un moyen simple pour empêcher l'utilisation d'un index pour une requête particulière. L'exemple suivant utilise un *marqueur pour l'optimiseur Oracle* (appelé « hint ») pour gérer ce cas.

```
SELECT /*+ NO_INDEX(EMPLOYES EMPLOYE_PK) */
       prenom, nom, id_supplementaire, numero_telephone
  FROM employes
 WHERE nom  = 'WINAND'
   AND id_supplementaire = 30;
```

Le plan d'exécution qui était utilisé avant la modification de l'index n'utilisait pas d'index du tout :

```
-----------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
-----------------------------------------------------
|  0 | SELECT STATEMENT  |           |    1 |  477 |
|* 1 |  TABLE ACCESS FULL| EMPLOYES  |    1 |  477 |
-----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("NOM"='WINAND' AND "ID_SUPPLEMENTAIRE"=30)
```

Même si l'opération `TABLE ACCESS FULL` doit lire et traiter la table entière, il semble plus rapide d'utiliser l'index dans ce cas. Ceci est plutôt inhabituel car la requête ne récupère qu'une ligne. Utiliser un index pour trouver une seule ligne devrait être bien plus rapide qu'un parcours de table complet. Mais cela ne se passe pas ainsi ici. L'index semble être lent.

Dans ce cas, il est préférable de vérifier chaque étape du plan problématique. La première étape est l'opération `INDEX RANGE SCAN` sur l'index `EMPLOYEE_PK`. Cet index ne couvre pas la colonne `NOM`. `INDEX RANGE SCAN` peut seulement traiter le filtre sur `ID_SUPPLEMENTAIRE` ; la base de données Oracle affiche cette information sur le deuxième élément dans la partie « Predicate Information » du plan d'exécution. Vous pouvez donc voir les conditions appliquées à chaque opération.

ASTUCE

L'Annexe A, « *Plans d'exécution* » explique comment trouver la partie « Predicate Information » pour chaque base de données.

Le `INDEX RANGE SCAN` avec l'identifiant d'opération 2 (Exemple 2.1 à la page 19) applique seulement le filtre `ID_SUPPLEMENTAIRE=30`. Cela signifie qu'il parcourt l'arbre de l'index pour trouver le premier enregistrement pour lequel `ID_SUPPLEMENTAIRE` vaut 30. Ensuite, il suit la chaîne de nœuds feuilles pour trouver tous les enregistrements pour cette société. Le résultat de l'opération `INDEX ONLY SCAN` est une liste d'identifiants de lignes (généralement appelés `ROWID`) qui remplissent la condition `ID_SUPPLEMENTAIRE` : suivant la taille de la société, cela peut aller de quelques-uns à plusieurs centaines.

La prochaine étape est l'opération `TABLE ACCESS BY INDEX ROWID`. Elle utilise les `ROWID` trouvés à l'étape précédente pour récupérer les lignes, toutes

colonnes comprises, de la table. Une fois que la colonne NOM est disponible, la base de données peut évaluer le reste de la clause **where**. Cela signifie que la base de données doit récupérer toutes les lignes de la table pour lesquelles la clause ID_SUPPLEMENTAIRE=30 est vraie avant d'appliquer le filtre NOM.

La durée d'exécution de la requête ne dépend pas du nombre de lignes du résultat mais du nombre d'employés dans la société visée. Si cette société a peu d'employés, l'opération INDEX RANGE SCAN donnera de meilleures performances. Néanmoins, un TABLE ACCESS FULL peut se révéler plus rapide sur une grosse société car il peut lire dans ce cas de larges parties de la table en une fois (voir « Parcours complet de table » à la page 13).

La requête est lente parce que la recherche dans l'index renvoie de nombreux ROWID, un pour chaque employé de la société originale, et la base de données doit les récupérer un par un. C'est la combinaison parfaite des deux ingrédients qui rend l'index lent : la base de données lit un grand nombre d'éléments dans l'index et doit récupérer chaque ligne séparément.

Choisir le meilleur plan d'exécution dépend aussi de la distribution des données dans la table. Du coup, l'optimiseur utilise les statistiques sur le contenu de la base de données. Dans notre exemple, un histogramme contenant la distribution des employés par société est utilisé. Cela permet à l'optimiseur d'estimer le nombre de lignes renvoyées à partir de la recherche de l'index. Le résultat est utilisé pour le calcul du coût.

Statistiques

Un optimiseur basé sur les coûts utilise les statistiques sur les tables, colonnes et index. La plupart des statistiques sont récupérées au niveau des colonnes : le nombre de valeurs distinctes, la plus petite et la plus grande valeur (intervalle de données), le nombre de valeurs NULL et l'histogramme de la colonne (distribution des données). La valeur statistique la plus importante pour une table est sa volumétrie (en lignes et en blocs).

Les statistiques les plus importantes pour un index sont la profondeur de l'arbre, le nombre de nœuds feuilles, le nombre de clés distinctes et le facteur d'ordonnancement (voir le Chapitre 5, « *Regrouper les données* »).

L'optimiseur utilise ces valeurs pour estimer la sélectivité des prédicats de la clause **where**.

Si aucune statistique n'est disponible, par exemple parce qu'elles ont été supprimées, l'optimiseur utilise des valeurs par défaut. Les statistiques par défaut de la base de données Oracle font référence à un petit index avec une sélectivité moyenne. Elles amènent à supposer qu'un INDEX RANGE SCAN renverra 40 lignes. Le plan d'exécution affiche cette estimation dans la colonne « Rows » (encore une fois, voir l'Exemple 2.1 à la page 19). Évidemment, c'est très fortement sous-estimé car 1000 employés travaillent pour cette société.

Si nous fournissons des statistiques correctes, l'optimiseur fait un meilleur travail. Le plan d'exécution suivant montre la nouvelle estimation : 1000 lignes pour l'INDEX RANGE SCAN. En conséquence, il calcule un coût supérieur pour les accès suivant à la table.

```
-----------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-----------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |  680 |
|*1 | TABLE ACCESS BY INDEX ROWID| EMPLOYES     |    1 |  680 |
|*2 |  INDEX RANGE SCAN           | EMPLOYES_PK  | 1000 |    4 |
-----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - filter("NOM"='WINAND')
  2 - access("ID_SUPPLEMENTAIRE"=30)
```

Un coût de 680 est même plus important que le coût pour le plan d'exécution utilisant l'opération FULL TABLE SCAN (477, voir la page 20). Du coup, l'optimiseur préférera automatiquement l'opération FULL TABLE SCAN.

Cet exemple d'un index lent ne devrait pas cacher le fait qu'une bonne indexation est la meilleure solution. Bien sûr, chercher le nom est très bien supporté par un index sur NOM :

```
CREATE INDEX nom_emp ON employes (nom);
```

En utilisant le nouvel index, l'optimiseur calcule un coût de 3 :

Exemple 2.2. Plan d'exécution avec un index dédié

```
-----------------------------------------------------------------
| Id | Operation                    | Name      | Rows | Cost |
-----------------------------------------------------------------
|  0 | SELECT STATEMENT             |           |    1 |    3 |
|* 1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYES  |    1 |    3 |
|* 2 |   INDEX RANGE SCAN           | NOM_EMP   |    1 |    1 |
-----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ID_SUPPLEMENTAIRE"=30)
   2 - access("NOM"='WINAND')
```

L'accès à l'index ramène, suivant les estimations de l'optimiseur, une seule ligne. Du coup, la base de données doit récupérer uniquement cette ligne dans la table : ceci sera à coup sûr plus rapide qu'une opération FULL TABLE SCAN. Un index correctement défini est toujours meilleur qu'un parcours complet de table.

Les deux plans d'exécution provenant de l'Exemple 2.1 (page 19) et de l'Exemple 2.2 sont pratiquement identiques. La base de données réalise les mêmes opérations et l'optimiseur calcule des coûts similaires. Néanmoins, le deuxième plan est bien plus performant. L'efficacité d'une opération INDEX RANGE SCAN peut varier dans un grand intervalle, tout spécialement s'il est suivi d'un accès à la table. Utiliser un index ne signifie pas automatiquement qu'une requête est exécutée de la meilleure façon qu'il soit.