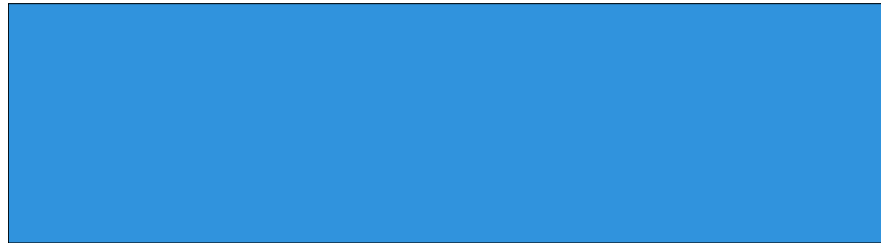
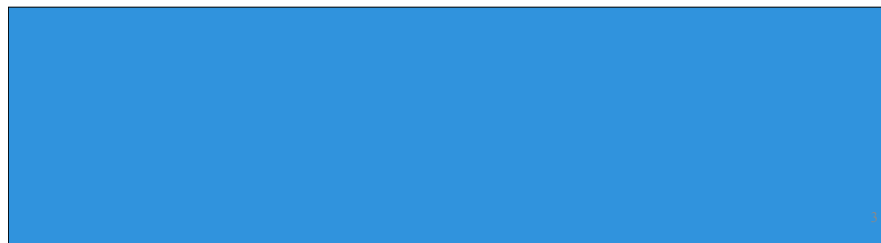


Techniques d'implémentation d'OLAP



MOLAP



Introduction

- On a vu qu'il existait 2 grandes alternatives :
 - MOLAP = structure de données adhoc, pour le multidimensionnel.
 - ROLAP = implémentation à l'aide d'un SGBD relationnel (schéma en étoile/flocon).
- On peut mélanger les 2 techniques (HOLAP) pour combiner leurs avantages.

2

Principes

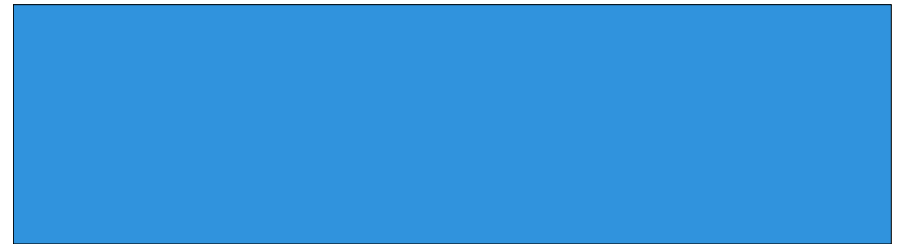
- Stockage des données de manière multidimensionnelle, dans des tableaux à n dimensions (hypercubes)
- Pré-calcul des mesures donc des agrégations, sur tous les niveaux des hiérarchies
- Pas de cadre technologique commun : chaque produit a ses stratégies de stockage
- Langage de requête MDX (*Multidimensional expressions*)

Avantages/Inconvénients

- Avantage :
 - Bonnes performances car pré-calculs, indexation adaptée
- Inconvénients :
 - Comme on stocke tous les calculs, ça génère un grand volume d'informations (mais parfois stratégies de compression)
 - La phase de chargement est longue (parfois maintenance incrémentale possible)

5

ROLAP



Principes

- Utilisation d'un SGBD relationnel pour stocker un SID multidimensionnel : modèle en étoile / flocon
- Langage SQL étendu à OLAP
- Nécessité d'optimiser les requêtes : indexation, partitionnement, matérialisation des vues.

Avantages/Inconvénients

- Avantage :
 - Les données de production (le SIO) utilisent un SGBD relationnel, on reste dans les mêmes technologies.
 - Modèle relativement simple
- Inconvénients :
 - Modèle pas toujours adapté aux requêtes OLAP complexes

Indexation

En plus des index « traditionnels » (B-arbres, tables de hachage ... cf cours de F. Bossut), il existe des index adaptés au décisionnel :

- Bitmap index
- (Bitmap) join index

9

Index Bitmap

- Intéressant quand on utilise comme clé d'indexation une colonne qui peut prendre peu de valeurs différentes
 - Situation familiale $\in \{\text{marié, divorcé, veuf, pacsé, célibataire}\}$, sexe $\in \{M, F\}$
- Tableau avec autant de colonnes que de valeurs possibles de la clé et autant de lignes que dans la table à indexer.
- Chaque case (x,y) contient 1 bit qui indique si la ligne x a pour valeur de clé y , la ligne ne comporte que des 0 si la clé vaut null.

10

Index Bitmap

Exemple :

Rowid	M	F
213	1	0
234	0	1
395	1	0
423	0	0
...

Index Bitmap

Rowid	Id_Employe	sexe	age	Id_service	...
213	1	'M'	46	null	
234	2	'F'	52	13	
395	3	'M'	28	2	
423	4	null	34	2	
...			

Table RH

11

Index Bitmap

- Adapté quand on fait des selects, et pas (peu) de mises à jour.
- Opérateurs logiques pour répondre aux requêtes
- *where sexe = 'M' and situation = 'marié'*

rowid	'M'		rowid	'marié'		rowid	Req
123	0		123	1		123	0
153	1		153	1		153	1
264	1	AND	264	0	=	264	0
391	0		391	0		391	0
...

12

Join Index

- Optimise la jointure entre la table de faits et (certaines de) ses dimensions.
- Plusieurs variantes, selon les SGBD (Oracle, Teradata, ...)
- Sur l'exemple précédent, index qui donne pour chaque id_service l'ensemble des rowid des lignes de la table RH qui possèdent une référence à ce service
- Pour faire facilement des intersections d'ensembles de rowid, ces index auront une structure de **Bitmap** join index.

13

Partitionnement

- Partitionnement vertical :
 - Le tuple est coupé par sous-ensembles d'attributs.
 - C'est à la base des SGBD orientés colonnes/vectoriels
 - On peut par exemple partitionner en fonction de la fréquence d'utilisation des attributs
- Partitionnement horizontal :
 - En fonction des valeurs des attributs
 - Gestion de très grandes relations
 - Toutes les partitions ont le même schéma logique (i.e. les mêmes attributs).

14

Partitionnement horizontal

- Existe dans la majorité des SGBDR commerciaux
- Lorsqu'une table est volumineuse, on peut la partitionner, i.e. la répartir sur plusieurs tables plus petites, stockées sur des espaces physiques différents.
- Le découpage logique de la table permet un accès plus rapide aux informations (moins de lectures disques)
- C'est transparent pour l'utilisateur (requête SQL sur une table, pas besoin de savoir qu'elle est partitionnée)
- La partition se fait selon une clé de partition : c'est un sous-ensemble des colonnes qui permet à chaque ligne de la table d'être affectée sans ambiguïté à 1 partition.

15

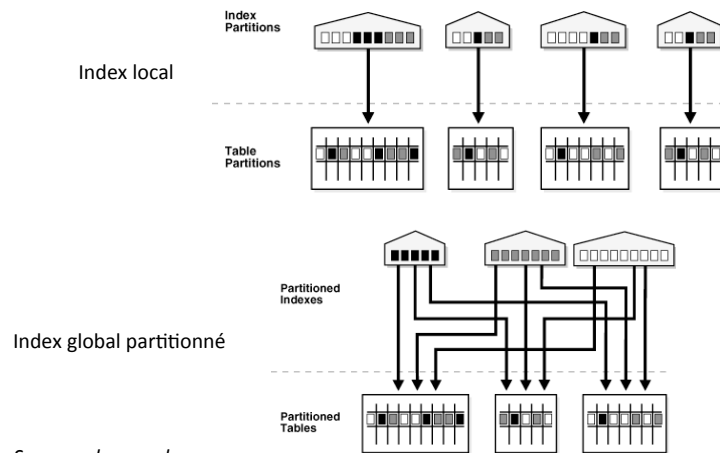
Partitionnement d'index

Les index, comme les tables, peuvent être partitionnés

- en même temps que la table (index local), dans ce cas 1 partition d'index pour 1 partition de la table
- ou indépendamment du partitionnement de la table (index global)

16

Partitionnement d'index



Source : doc oracle

17

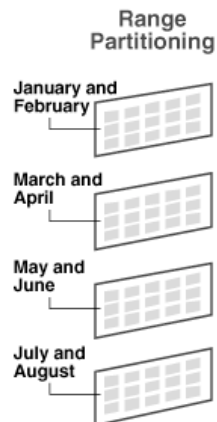
Partitionnement sous Oracle

Une table peut être partitionnée horizontalement selon 4 méthodes :

1. Range : en fonction d'intervalles de valeurs
2. List : en fonction de listes de valeurs
3. Hash : selon une fonction de hashage
4. Composite : utilise la méthode range et fabrique pour chaque partition une sous-partition (avec la méthode List ou Hash)

18

Range - exemple



```
CREATE TABLE sales_range
(salesman_id NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount NUMBER(10),
sales_date DATE) PARTITION BY
RANGE(sales_date)
(PARTITION sales_jan_feb2000
VALUES LESS THAN(TO_DATE('03/01/2000',
'MM/DD/YYYY')),
PARTITION sales_mar_apr2000
VALUES LESS THAN(TO_DATE('05/01/2000',
'MM/DD/YYYY')),
PARTITION sales_may_jun2000
VALUES LESS THAN(TO_DATE('07/01/2000',
'MM/DD/YYYY')),
PARTITION sales_jul_aug2000
VALUES LESS THAN(TO_DATE('09/01/2000',
'MM/DD/YYYY')) );
```

19

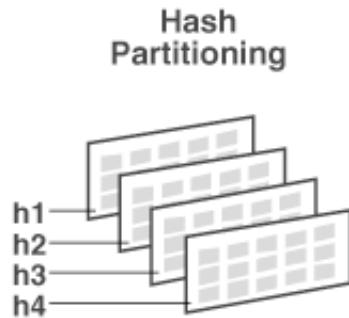
List - exemple



```
CREATE TABLE sales_list
(salesman_id NUMBER(5),
salesman_name VARCHAR2(30),
sales_state VARCHAR2(20),
sales_amount NUMBER(10),
sales_date DATE)
PARTITION BY LIST(sales_state)
(
PARTITION sales_east
VALUES ('New York', 'Virginia', 'Florida'),
PARTITION sales_west
VALUES ('California', 'Oregon', 'Hawaii'),
PARTITION sales_central
VALUES ('Texas', 'Illinois', 'Missouri')
);
```

20

Hash - exemple



```
CREATE TABLE sales_hash
(salesman_id NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount NUMBER(10),
week_no NUMBER(2))
PARTITION BY
HASH(salesman_id)
PARTITIONS 4
STORE IN (ts1, ts2, ts3, ts4);
```

La clause STORE permet de répartir les éléments de la partitions dans différents tablespaces (ts1,ts2,...).

21

Vues matérialisées

- Une vue permet de définir une relation par une requête.
- Vue « classique » : on stocke la définition de la vue, i.e. le texte d'une requête select.
- **Vue matérialisée** : on stocke la définition **et** le résultat de la requête.

22

Utilisation des vues matérialisées

- Beaucoup de requêtes similaires sur les mêmes tables de l'entrepôt.
→ définir des vues matérialisées sur l'entrepôt permet une **optimisation des performances** car pré-calcul (d'une partie) de ces requêtes fréquentes.
Difficulté de la détermination d'une réécriture logique d'une requête utilisant les vues matérialisées. *Problème NP-dur.*
- En TP : définition du schéma en étoile par des vues matérialisées sur les tables du SIO – permet de ne pas utiliser d'ETL.

23

Maintenance des vues mat.

- Dans un entrepôt, les données ne sont pas ou peu modifiées, mais beaucoup interrogées :
les vues matérialisées sont donc bien adaptées car il ne faut pas les mettre à jour très souvent (synchronisation avec les données dans les tables de base).
- Maintenance :
 - re-calcul complet de la vue matérialisée (coûteux), ou
 - maintenance incrémentale.

24

Gestion des vues matérialisées

Quand on définit une vue matérialisée, il faut donc être capable de calculer

- Le gain en terme de performance (temps de réponse aux requêtes), par rapport à la perte de temps dû à la réactualisation de la vue et de ses index.
- La volumétrie : quelle place occupe la vue matérialisée et ses index ?

25

Exemple sous Oracle

```
Create materialized view costs_vm
Build immediate
Refresh fast on demand
Enable query rewrite as
Select time_id, prod_name,
sum(unit_cost) as sum_units,
count(unit_cost) as count_units,
count(*) as cnt
From costs c join products p on c.prod_id =
p.prod_id
Group by time_id, prod_name
```

26

Requête définissant la vue

- Une vue matérialisée peut inclure
 - des agrégations (sum, count, ...)
 - des jointures
 - un group by
- Il ne peut pas y avoir de sous-requête dans le select.

27

Chargement initial des données

- **Build immediate** : la définition de la vue est stockée dans le dictionnaire, la requête est exécutée et son résultat est stocké dans l'objet Vue Matérialisée
- **Build deferred** : la définition de la vue est stockée dans le dictionnaire mais elle est vide jusqu'au prochain DBMS_MVIEW.REFRESH

28

Actualisation d'une vue

- **Refresh complete** : à chaque rafraichissement, on recalcule toute la requête définissant la vue. Donc toutes les données de la vues sont supprimées puis rechargées.
- **Refresh fast** : on rafraichit la vue incrémentalement. Il faut avoir créé un objet « log » pour chaque table dont dépend la vue. *Attention, ce mode n'est pas toujours possible, ça dépend de la définition de la vue.*

Exemple de création d'un log sur la table COSTS :

```
Create materialized view log on costs
```

- **Refresh force** (par défaut) : effectue si c'est possible un refresh fast, sinon un refresh complete.

29

Fréquence de l'actualisation

- **On Demand** : on actualise la vue en appelant une procédure stockée.

Par exemple :

```
Execute dbms_mview.refresh('COSTS_VM') ;
```

- **On Commit** : la vue est automatiquement actualisée à chaque fois qu'on valide une transaction dans laquelle on modifie une table dont dépend la vue.

30

Réécriture de requête - principe

```
Select L.state, sum(s.sales)
From locations L
Join Sales S on L.locid = S.locid
Group by L.state
```

```
Select P.category, sum(s.sales)
From products P
Join Sales S on P.pid = S.pid
Group by P.category
```

La table Sales est volumineuse (table de faits) et chaque jointure est donc coûteuse

31

- Vue matérialisée :

```
Create materialized view TotalSales as
Select S.pid, S.locid, sum(S.sales)
From Sales S
Group by S.pid, S.locid
```

- La vue TotalSales peut être utilisée à la place de Sales dans les 2 requêtes précédentes. La jointure est moins volumineuse.
- On utilise le fait que s.sales est une mesure additive selon les dimensions Locations et Products.

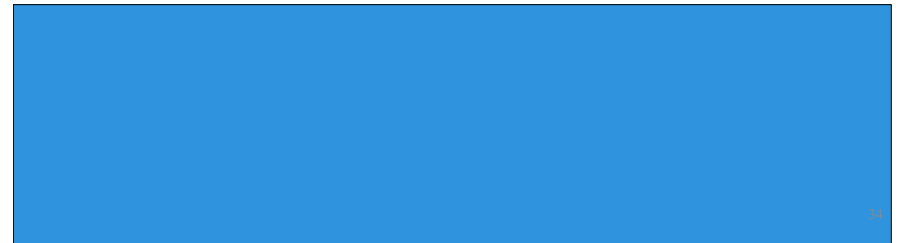
32

Réécriture de requête - Oracle

- **Enable Query Rewrite** : permet de créer une vue en autorisant son utilisation pour la réécriture de requête (elle devient un cache pour l'exécution des requêtes).
- Plusieurs procédures stockées permettent d'obtenir des informations sur la vue, et en particulier de savoir si une vue est éligible pour de la réécriture de requête.
- Mode par défaut : Disable Query Rewrite

33

HOLAP



Principes

- On combine un mode de stockage ROLAP avec un mode MOLAP.
- En général :
 - Un SGBD relationnel (approche ROLAP) pour stocker les données détaillées
 - Un SGBD multidimensionnel (approche MOLAP) pour stocker les données agrégées
 - = pour répondre plus rapidement aux requêtes
- Bon compromis : permet de gérer des grandes quantités de données, avec des temps de réponse acceptables