

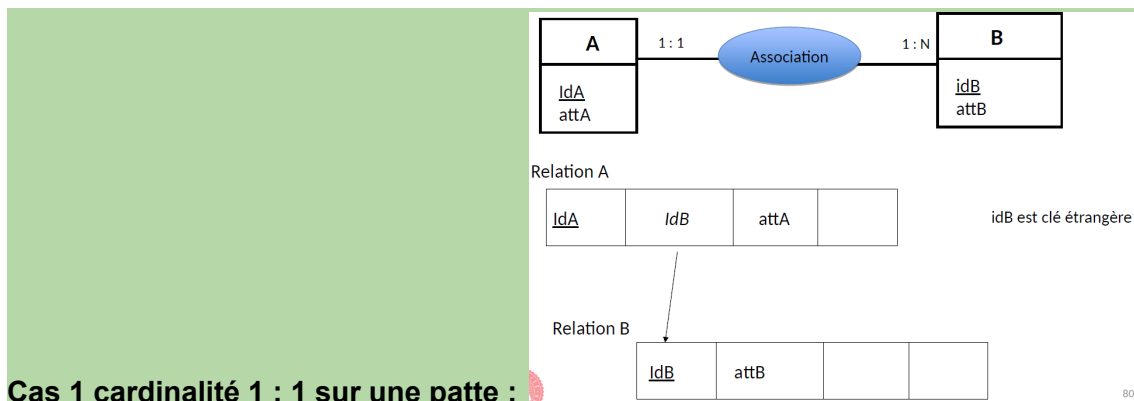
BIG-DATA :

Rappels modèle relationnels, UML, Stockage

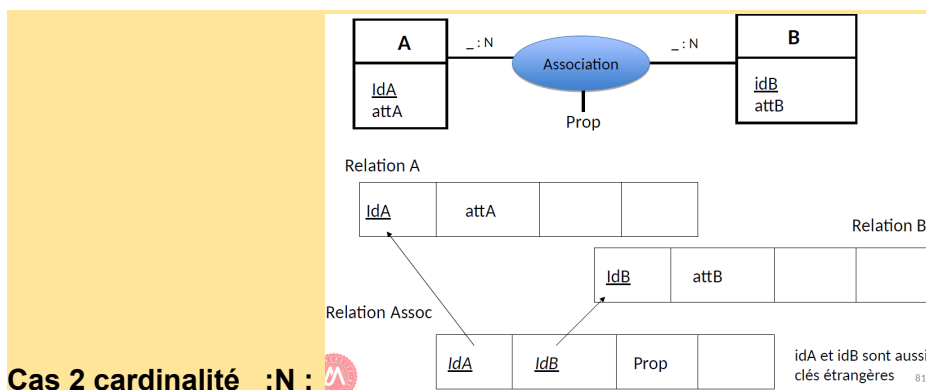
Modèle Conceptuel	UML, EA, Merise
Modèle Logique	Modèle relationnel, modèle object, graph
Modèle Physique	SQL, OQL, XML

Transformations pour les Types d'Associations

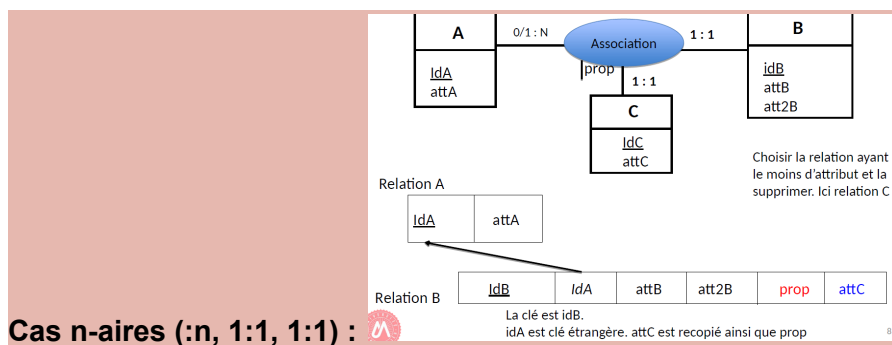
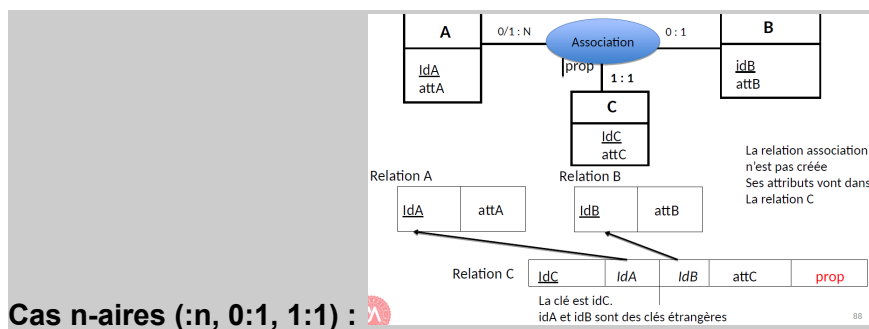
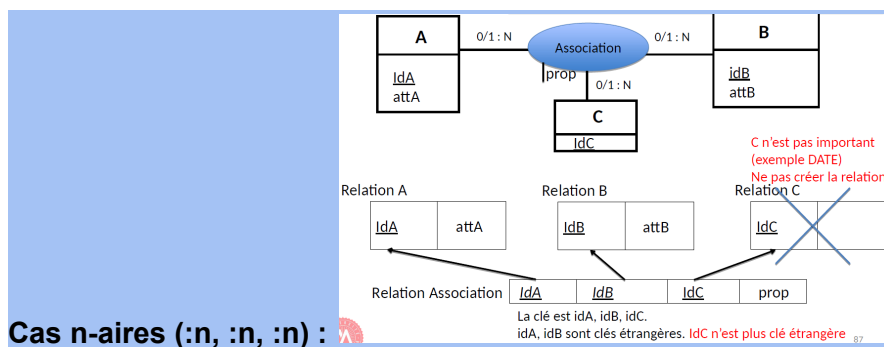
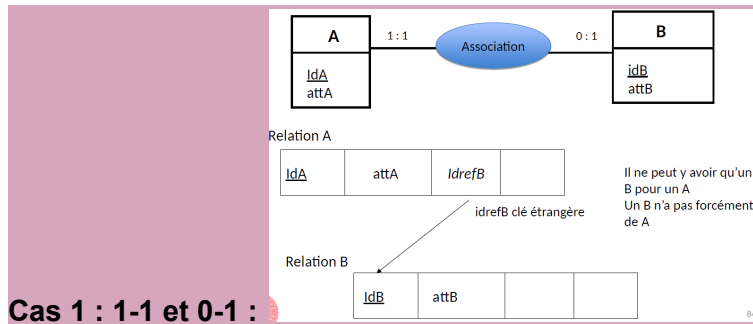
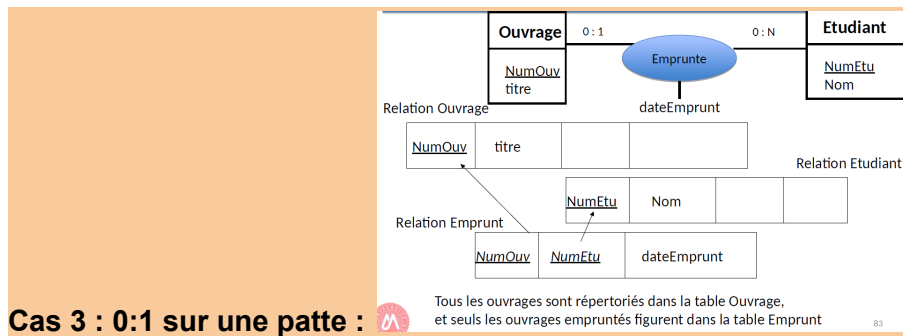
- **cas 1 : cardinalité 1,1 sur l'une des pattes** : compléter la relation concrétisant l'entité avec la patte 1,1 en y ajoutant une propriété qui référence l'identifiant de l'autre entité
- **cas 2 : toutes les cardinalités maximale = n** : créer une relation pour l'association, dont la clé se compose des clés des entités liées, avec éventuellement les propriétés portées par l'association
- **cas 3 : cardinalité 0,1 sur l'une des pattes** : choisir entre une traduction selon le cas 1 ou le cas 2 en veillant toutefois : dans le cas 1, si l'association a des propriétés, ne pas oublier de les adjoindre à la relation correspondant à l'entité avec la patte 0,1 et dans le cas 2 de simplifier l'identifiant.



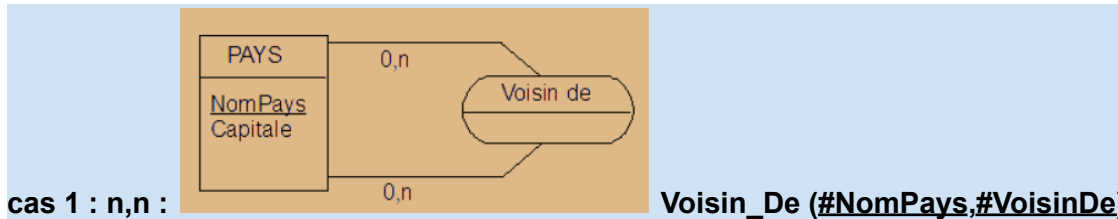
Cas 1 cardinalité 1 : 1 sur une patte :



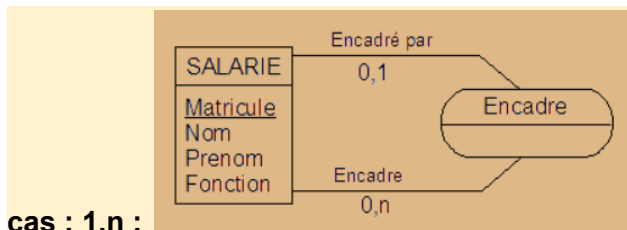
Cas 2 cardinalité _:N :



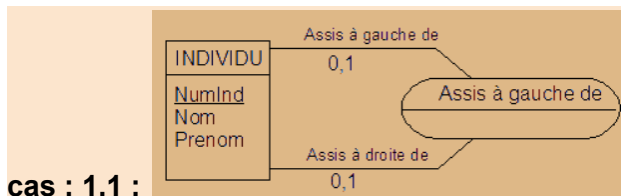
Associations réflexives en relationnel :



Une association réflexive $[n,n]$ sur une entité E est traduite en une relation de même nom avec deux clés étrangères. L'une d'elles porte le nom de l'identifiant de l'entité et l'autre, le nom de l'association. La clef primaire de cette relation est constituée de ces deux attributs.

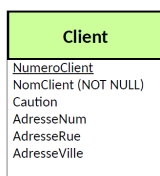


Une association réflexive $[1,n]$ sur une entité est traduite en une clef étrangère dans la relation représentant cette entité. Le nom de cette clef étrangère est celui de l'association si c'est le nom de l'association qui étiquette le trait de cardinalité maximale 1, sinon c'est le nom de l'association réciproque.



Une association réflexive $[1,1]$ sur une entité est traduite en une clef étrangère dans la relation représentant cette entité. Le nom de cette clef étrangère est celui de l'association.

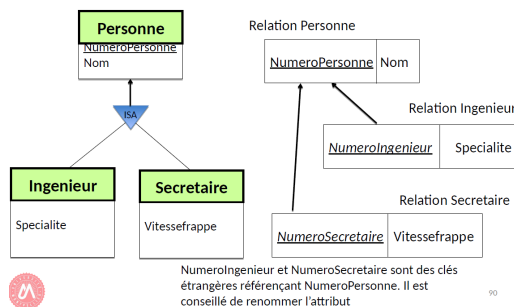
Les héritages :



Relation Client
(NumeroClient,
NomClient, AdresseNum,
AdresseRue,
AdresseVille)

```
/*
Creation de la table client avec les contraintes
*/
CREATE TABLE CLIENT (
    NUMEROCLIENT INT,
    NOMCLIENT VARCHAR(15) NOT NULL,
    CAUTION NUMERIC(3,0),
    ADRESSENUM NUMERIC(3,0),
    ADRESSE RUE VARCHAR(25),
    ADRESSEVILLE VARCHAR(15),
    CONSTRAINT PK_Client (NUMEROCLIENT)
);

/*
Creation de l'index sur la cle primaire pour
optimiser les jointures
*/
CREATE INDEX ICLIENT ON CLIENT(NUMEROCLIENT);
```



Première forme normale :

Une relation est en 1FN ssi tous ses attributs sont atomiques (mono-valués). Créer autant d'attributs que le nombre maximum de valeurs (stockage horizontal) Personne (Num, Nom, Prenom1, Prenom2, Prenom3). Créer une nouvelle relation comprenant la CP de la relation initiale et l'attribut multi-valué. Attention : éliminer l'attribut de la relation initiale.

Seconde forme normale :

Une relation est en seconde forme normale (2FN) ssi

- Elle est en 1FN
- Tout attribut n'appartenant pas à la clé primaire est en DF totale avec la clé

Isoler la DF responsable dans une nouvelle relation. Elle devient CP dans la relation initiale

• Éliminer l'attribut cible de la DF dans la relation initiale.

• Attention : la seconde forme normale implique que :

« Tout attribut n'appartenant pas à la clé primaire est en DF totale avec la clé ». Ceci doit être vrai pour les clés candidates.

Troisième Forme Normale :

Une relation est en 3FN ssi

- Elle est en 2FN
- Elle ne contient pas de DF transitive entre attributs non clés

Isoler la DF transitive dans une nouvelle relation

• Eliminer l'attribut cible de la DF dans la nouvelle relation

Une relation est en **BCFN** (Boyce and Codd Normal Form) ssi elle est en 3FN et qu'aucun attribut de la clé ne dépend d'un attribut non clé. $R(A1, A2, A3)$ est en BCFN s'il n'existe pas $A3 \rightarrow A1$. Toute relation admet une décomposition en BCFN sans perte d'information. Une décomposition BCFN ne conserve pas les DF.

Souvent au niveau 3NF et BCNF :

- Plus d'anomalie de stockage
- Modification : modification du salaire des pilotes pour tous
- Insertion : on peut stocker le salaire d'un contrôleur sans avoir un employé de cette catégorie
- Suppression : si DURANT est supprimé on conserve l'information sur le salaire des mécaniciens

- 3NF = 2NF + no indirect dependencies (simplification)
- 2NF = 1NF + no partially dependent attributes (simplification)
- 1NF = only atomic values in a tuple

DDL (Data Definition Language)	CREATE/ALTER structures (table, views, index)
DML (Data Manipulation Language)	UPDATE/INSERT/DELETE content
DQL (Data Query Language)	SELECT data

<> = différent : SELECT DISTINCT business_key FROM memory WHERE concept <> 'case' or a_rib <> 'status' or value <> 'closed'

Null : WHERE att IS NULL ou bien WHERE att IS NOT NULL

Alter Table :

- **Add a new column**
 - ALTER TABLE Employee ADD (office VARCHAR2(20));
- **Modify an existing column**
 - ALTER TABLE Employee MODIFY (office NUMBER);
- **Define a default value for the new column**
 - ALTER TABLE Employee MODIFY office DEFAULT 'Corridor';
- **Drop a column**
 - ALTER TABLE Employee DROP (office);

Truncate = DROP + CREATE TABLE

TO_DATE = converts a character/numeric to a date

TO_CHAR = converts a numeric to a character

Mises à jours :

- Le pilote DUPONT change d'adresse et son salaire est augmenté de 10 % :
UPDATE PILOTE SET Adr='PARIS', Sal=Sal*1.1 WHERE PInom = 'DUPONT';

Insertion :

INSERT INTO nomrelation (list_att) VALUES (list_val) ;

Suppression :

DELETE FROM nomrelation WHERE condition;

DELETE FROM PILOTE WHERE PInum=206;

• **Les attributs mentionnées dans le SELECT doivent être indiquées dans le GROUP BY**

Requêtes corrélées :

SELECT * FROM PILOTE WHERE Sal =
(SELECT Sal FROM PILOTE WHERE PInum=110)

- **Quels sont les pilotes qui effectuent des vols ?**

SELECT * FROM PILOTE LesPilotes
WHERE NOT EXISTS (SELECT * FROM Vol WHERE LesPilotes.PInum=Vol.PInum) ;

- **Les pilotes conduisant tous les airbus :**

SELECT * FROM PILOTE WHERE NOT EXISTS (SELECT * FROM AVION
WHERE Avnom LIKE 'AIRBUS%' AND NOT EXISTS (SELECT *
FROM VOL WHERE VOL. PInum=PILOTE.PInum AND VOL.Avnum=AVION.Avnum));

- **PRIMARY KEY** : pour spécifier une clé primaire
- **FOREIGN KEY** : pour spécifier une clé étrangère
- **REFERENCES** : pour les contraintes d'inclusion
- **CHECK** : contrainte générale
- **UNIQUE** : oblige le fait d'avoir une valeur unique
- **NOT NULL** : pour obliger à saisir une valeur
- **CONSTRAINT** : pour nommer une contrainte – Très utile

Les ON DELETE :

• ON DELETE CASCADE

- Objectif : supprimer automatiquement toutes les valeurs des attributs qui référencent la valeur.
- Conséquences : les tuples dans la relation associée sont supprimés.

• ON DELETE SET NULL

- Objectif : mettre à NULL toutes les valeurs de clés étrangères associées.
- Conséquences : possible si la clé étrangère associée est NULL. Impossible si la clé a la contrainte NOT NULL.

Les ON UPDATE :

- Cela veut dire que l'on change une clé primaire. Quelles en sont les conséquences ?

• ON UPDATE CASCADE

- Objectif : répercuter les modifications dans la relation associée
- Conséquences : les tuples associés sont modifiés

• ON UPDATE SET NULL

- Objectif : mettre à NULL toutes les valeurs associées
- Conséquences : NULL pour la clé étrangère associée

Les indexs :

- Les **index** sont utilisés pour accélérer les accès à une relation
- Ils sont là pour optimiser les requêtes :

CREATE INDEX nom_index ON nom_table (nomdesattributs);

CREATE INDEX IPILOTE ON PILOTE (PLNUM);

Suppression d'une base :

DROP SCHEMA nom_base [CASCADE|RESTRICT];

- CASCADE : effacer toute la base
- RESTRICT : effacer seulement si la base n'a plus de tuples

Suppression d'une relation :

- DROP TABLE nom de la relation [CASCADE|RESTRICT];

DROP TABLE AVION;

Clés étrangères :

CREATE TABLE Employee (id NUMBER, department NUMBER FOREIGN KEY department REFERENCES Dept(dept_id))

Enums avec Check :

CREATE TABLE Employee (id NUMBER, department NUMBER, office VARCHAR2(10)
CHECK (office IN ('DALLAS','BOSTON', 'PARIS','TOKYO')))

La notion de vues :

• Rôle de sécurité

- L'utilisateur ne peut accéder qu'aux données des vues auxquelles il a le droit d'accès

• Contraintes d'intégrité

- Mise à jour au travers de vues

Vue : Base de données virtuelle dont le schéma et le contenu sont dérivés de la base réelle par un ensemble de requêtes. Une vue est donc un ensemble de relations déduites d'une base de données, par composition des relations de la base.

Création d'une vue :

CREATE VIEW <NOM DE VUE> [(LISTE D'ATTRIBUT)] AS <REQUETE> [WITH CHECK OPTION]

- La clause **WITH CHECK OPTION** permet de spécifier que les tuples de la vue insérés ou mis à jour doivent satisfaire aux conditions de la requête.

Destruction d'une vue :

– DROP VIEW <NOM DE VUE>

Une vue n'a pas d'existence physique. Une destruction n'entraîne pas la suppression de tuples de la base.

Exemple création d'une vue pour les pilotes niçois :

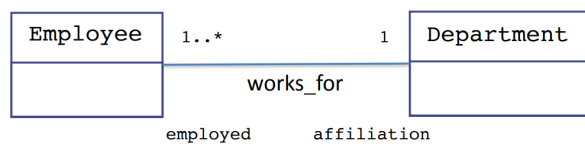
```
CREATE VIEW PILOTESNICOIS (Pnum, Pnom, Sal) AS SELECT Pnum, Pnom, Sal
FROM PILOTE WHERE Adr = 'NICE';
```

Mise à jour des vues :

- En pratique : seuls les attributs d'une table de la base doivent apparaître dans la vue
- Imposer que la clé de la table soit présente

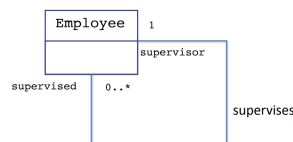
UML :

Ici un employé travaille pour un seul et unique département, et 1 à n employés travaillent dans un département. Un employé est employé dans un département. Un département est affilié un employé.

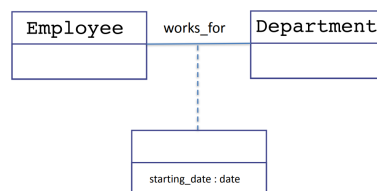


à

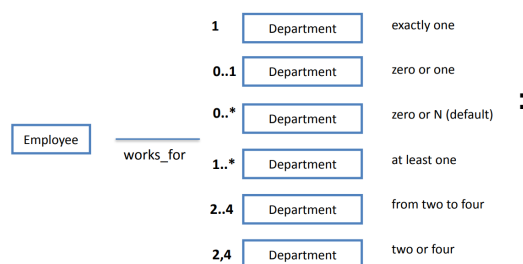
Associations réflexives :



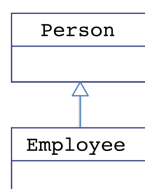
Attributs d'association :



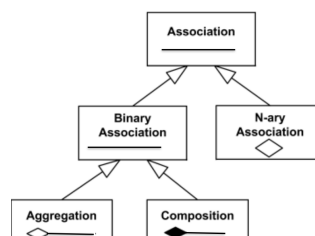
Les différents types de liaisons

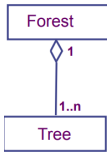
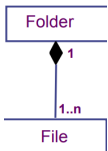


Sous classe et hiérarchie :



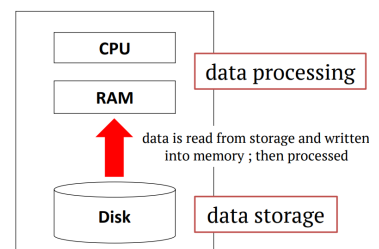
Les types de liaisons UML :



Agrégation : un arbre peut exister sans forêt.	
Composition : Un fichier ne peut pas exister sans répertoire File ne peut pas avoir de composition sur une autre classe, car interdit par UML	

Stockage base de données :

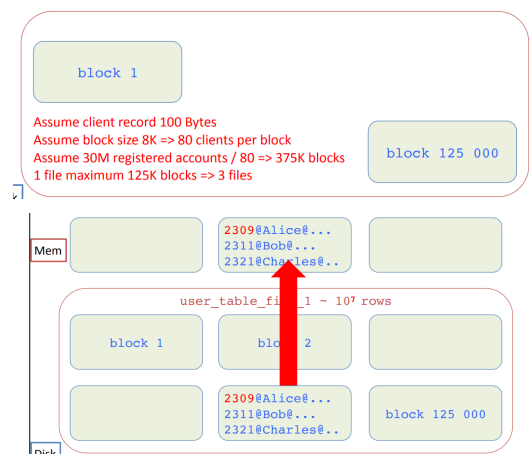
Quand nous faisons une requête les données sont lues depuis un disque, ou les datas sont persistantes. La requête est compilée dans le CPU, les datas passent ensuite des registres au CPU.



Dans l'ordre nous avons : Cache > RAM > Disque

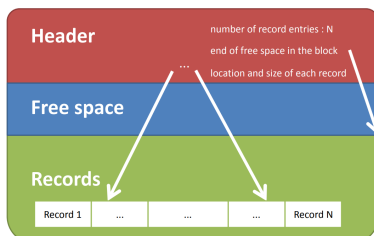
Postgres stocke les données dans plusieurs fichiers. Ces datas sont partitionnées en blocs, de tailles fixes.

- default size (tunable) : 8KB (maximum Postgres 32K)
- each block-id have a 32-bit integer ID (allows ~2 billion blocks)
- max table size : #blocks x block_size (16TB to 64TB)



Lors d'une requête, on va parcourir tous les blocs jusqu'à trouver la donnée recherchée.

Un bloc est composé de 3 parties, le header, l'espace libre et les enregistrements.

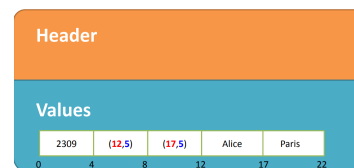


- Les enregistrements sont stockés par colonne de manière séquentielle.
- Ces enregistrements peuvent être déplacés de manière à les garder contigus et de ne pas avoir d'espaces libres entre les enregistrements.

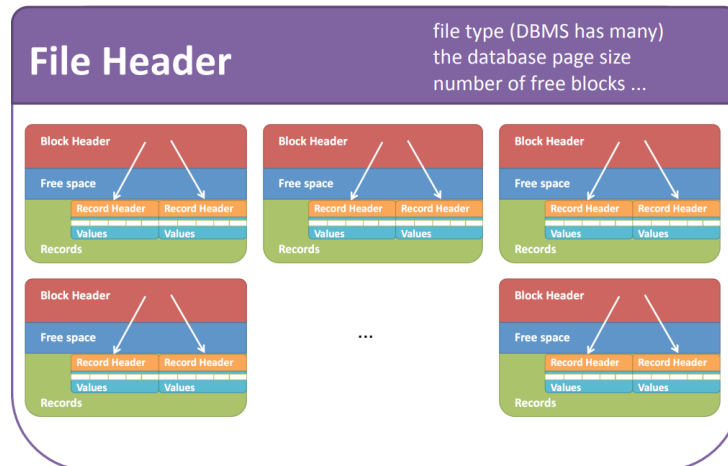
- La plupart des enregistrements sont de taille variable.
- Les attributs sont stockés dans l'ordre.

Un enregistrement est stocké de la manière suivante :

- Les attributs de taille variables sont représentés sous la forme (offset, longueur).



Et donc chaque fichiers est de la forme suivante :



Optimisation

La possibilité d'influer sur l'optimisation faite par le SGBD implique une meilleure optimisation.

- **Index non utilisé si :**
 - fonction ou d'opérateur utilisés sur une colonne indexée
 - comparaison des colonnes indexées avec la valeur null
- **Éviter les opérations inutiles**
 - le select *
 - le tri
 - filtre sur les données le plus tôt possible dans le cas de requêtes imbriquées et de jointures
- **Favoriser les opérations les moins coûteuses**
 - Favoriser les UNION/UNION ALL aux OR
 - Favoriser le EXISTS par rapport au IN lorsque la liste à parcourir est issue d'une sous-requête et pas d'une liste statique
 - Attention au IN/NOT IN lorsqu'il y a des valeurs nulles : il ne peut pas les comparer et considère qu'elles n'existent pas.

Requête SQL → Plan d'exécution logique (l'algèbre)

Plan d'exécution physique – PEP (opérateurs)

- Trouver les expressions équivalentes

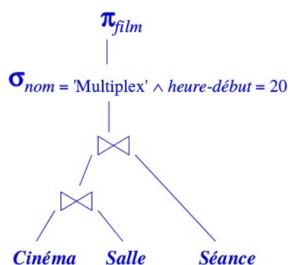
- l'algèbre permet d'obtenir une version opératoire de la requête
- les équivalences algébriques permettent d'explorer un ensemble de plans
- l'optimiseur évalue le coût (entrée / sortie) de chaque plan : différentes fonctions / modèles de coût existantes

Les règles de réécriture :

1. **Commutativité des jointures** : $R \bowtie S \equiv S \bowtie R$
2. **Regroupement des sélections** : $\sigma_{A='a' \wedge B='b'}(R) \equiv \sigma_{A='a'}(\sigma_{B='b'}(R))$
3. **Commutativité de σ et de π** : $\pi_{A_1, A_2, \dots, A_p}(\sigma_{A_i='a'}(R)) \equiv \sigma_{A_i='a'}(\pi_{A_1, A_2, \dots, A_p}(R))$
4. **Commutativité de σ et de \bowtie** : $\sigma_{A='a'}(R[\dots A \dots] \bowtie S) \equiv \sigma_{A='a'}(R) \bowtie S$

- **Arbre algébrique de requête**

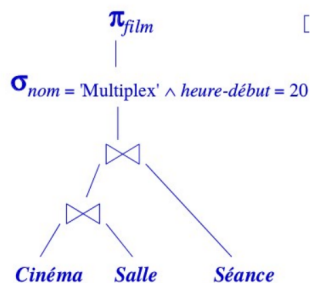
$\pi_{film}(\sigma_{nom = 'Multiplex' \wedge heure-début = 20}((Cinéma \bowtie Salle) \bowtie Séance))$



- **Hypothèses (en nombre de lignes) :**
 - Cinéma : 4 lignes dont 20 % de Multiplex
 - Salle : 6 lignes dont 50 % des salles de Cinéma
 - Séance : 50 lignes et 50 % des séances après 20h

- **Plan 1 :**
 - Jointure : on lit $4 * 6 = 24$ lignes et on produit $50 \% * 6 = 3$ lignes
 - Jointure : on lit $3 * 50 = 150$ lignes et on produit 50 lignes
 - Sélection : on lit 50 lignes et on produit $50 \% * 50 = 25$ lignes
 - Sélection : on lit 25 lignes et on produit $20 \% * 25 = 5$ lignes
 - On laisse de côté la projection (même coût dans les deux cas et même nombre de lignes)

➡ Coût (E/S) : $24E + 3S + 150E + 50S + 50E + 25S + 25E + 5S = 332$ lignes E/S



- **Facteur de sélectivité S**

- Proportion de n-uplets du produit cartésien des relations touchées qui satisfont une condition

- **Exemple :**

SELECT * FROM R1, R2

➡ $S = 1$

SELECT * FROM R1 WHERE A = valeur

➡ $S = 1 / \text{CARD}(A)$ avec un modèle uniforme

- **$TAILLE(R1 \bowtie_{A=B} R2) = p * TAILLE(R1) * TAILLE(R2)$**
 - p dépend du type de jointure et de la corrélation des colonnes :
 - $p = 0$ si aucun n-uplet n'est joint
 - $p = 1 / \text{MAX}(\text{CARD}(A), \text{CARD}(B))$ si distribution uniforme équiprobable des attributs A et B sur un même domaine (col. Join.)
 - $p = 1$ si produit cartésien
- **Cas particulier :**
 - Si A est clé de R1 et B est clé étrangère de R2 alors
 $TAILLE(R1 \bowtie_{A=B} R2) = TAILLE(R2)$

La réécriture algébrique est nécessaire, mais pas suffisante

Il faut tenir compte d'autres critères :

- Les chemins d'accès aux données (selon l'organisation physique)
 - On peut accéder aux données d'une table par accès séquentiel, par index, par hachage, etc.
- Les différents algorithmes possibles pour réaliser un opérateur
 - Il existe par exemple plusieurs algorithmes pour la jointure
 - Souvent ces algorithmes dépendent des chemins d'accès disponibles
- Les propriétés statistiques de la base de données
 - Taille des tables
 - Sélectivité des attributs
 - etc.

Les opérations de l'algèbre relationnelle peuvent être évaluée à l'aide de plusieurs algorithmes : une même expression d'algèbre relationnelle peut être évaluée de plusieurs façons différentes

Rôle de l'optimiseur, les opérateurs (PEP)

Tout opérateur est implanté sous forme d'un itérateur

- Trois fonctions :
 - open : initialise les ressources et positionne le curseur
 - next : ramène l'enregistrement courant et se place sur l'enregistrement suivant
 - close : libère les ressources
- Un plan d'exécution est un arbre d'itérateurs
 - un itérateur consomme des nuplets d'autres itérateurs source ou de données
 - un itérateur produit des nuplets à la demande
- **Rôle des itérateurs : principes essentiels**
 - production à la demande : le serveur n'envoie un enregistrement au client que quand ce dernier le **demande**
 - Pipeline : on essaie d'éviter le stockage en mémoire de résultats intermédiaires (le résultat est calculé au fur et à mesure)
 - ➡ évite d'avoir à stocker des résultats intermédiaire
 - ➡ temps de réponse minimisé mais attention aux opérateurs bloquants
 - temps de réponse : temps pour obtenir le premier nuplet
 - temps d'exécution : temps pour obtenir tous les nuplets.
- **Opérateur bloquant**
 - ➡ on additionne le temps d'exécution et le temps de d'exécution
`| select min(date) from T`

3 principaux types d'opérateurs et plusieurs algorithmes pour chaque opération:

Accès aux données (via les tables et les index)

- parcours séquentiel de la table (FullScan)
- parcours d'index (Index Scan)
- accès par adresse (DirectAccess)
- test de la condition (Filter)

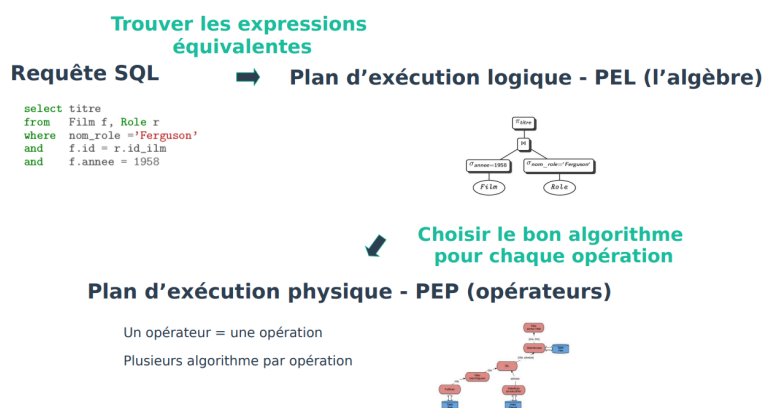
Jointures (sans / avec index)

- jointure par boucles imbriquées (Nested loop join) indexée ou non indexée
- jointure par tri-fusion (Merge sort join)
- jointure par hachage (Hash join)

Tri et regroupement

- tri externe (même type d'algorithme que pour les jointures)

Résumé optimiseur :



Un plan d'exécution (physique) :

C'est un programme combinant des opérateurs physiques (chemins d'accès et traitements de données).

Il a la forme d'un arbre : chaque nœud est un opérateur qui

- prend des données en entrée
- applique un traitement
- produit les données traitées en sortie

La phase d'optimisation proprement dite :

- Pour une requête, le système a le choix entre plusieurs plans d'exécution (logiques puis physiques).
 - Ils diffèrent par l'ordre des opérations, les algorithmes, les chemins d'accès.
 - Pour chaque plan on peut estimer le coût de chaque opération et la taille du résultat
- Objectif : diminuer le plus vite possible la taille des données manipulées.

L'optimiseur ORACLE :

L'optimiseur ORACLE suit une approche classique :

- Génération de plusieurs plans d'exécution.
- Estimation du coût de chaque plan généré.
- Choix du meilleur et exécution.

Tout ceci est automatique, mais il est possible d'influer, voire de forcer le plan d'exécution

Paramètres pour l'estimation du coût :

- Les chemins d'accès disponibles.
- Les opérations physiques de traitement des résultats intermédiaires.
- Des statistiques sur les tables concernées (taille, sélectivité) : appel explicite à l'outil ANALYZE. – Les ressources disponibles.

Les chemins d'accès de l'optimiseur ORACLE :

- Parcours séquentiel :
 - TABLE ACCESS FULL
- Accès direct par adresse :
 - TABLE ACCESS BY (INDEX|USER|...) ROWID
- Accès par index :
 - INDEX (UNIQUE|RANGE|...) SCAN
- Accès par hachage :
 - TABLE ACCESS HASH
- Accès par cluster :
 - TABLE ACCESS CLUSTER

Directives pour l'optimiseur ORACLE :

- Directive (hint) utilisée pour influencer l'optimiseur

- imposer un opérateur spécifique,
- faire le choix sur l'exploitation d'un index ou non, ...

➡ **Utile en mode de conception ou lorsque l'optimiseur ne choisit pas un plan optimal (ex: mauvaises statistiques)**

- Insertion des directives dans la requête à exécuter

- Exemples de directives (doc ORACLE)
 - Ne pas exploiter l'index d'une table : /*+ NO_INDEX(nom_table) */
`select /*+ NO_INDEX(Commune) */ * from Commune ;`
 - Utiliser l'opérateur FullScan sur une table : /*+ full(nom_table) */
`select /*+ full(f) */ * from f where nom_f like 'd%';`

Obtenir le plan d'exécution d'une requête :

explain plan for select * from emp where num=33000 ;

Comment lire un plan d'exécution ?

- Parcours des étapes de haut en bas jusqu'à en trouver une qui n'a pas de fille (pas d'étape indentée en dessous)
 - Traitement de cette étape sans fille ainsi que de ses sœurs (étapes de même indentation)
 - Traitement de toutes les étapes mères jusqu'à trouver une étape qui a une sœur
 - Traitement de la sœur conformément à l'étape 1.
-
- Id : Identifiant de l'opérateur ;
 - Operation : type d'opération utilisée
 - Name : nom de la relation utilisée ;
 - Rows : le nombre de lignes qu'Oracle pense transférer. ,

- Bytes : nombre d'octets qu'oracle pense transférer.
- Cost : coût estimé par oracle

Obtenir les statistiques des requêtes : set autotrace on.

- Accès mémoire : nb de pages/blocs logiques lus
 - consistent gets : nb d'accès à une donnée consistante en RAM (non modifiée)
 - db block gets : nb d'accès à une donnée en RAM
- Accès physiques :
 - physical reads : nb total de pages/blocs lues sur le disque
 - recursive calls : nb d'appels à un sous-plan (requête imbriquée, tris)
- redo size : taille du fichier de log produit (écriture, mis à jour,...)
- sorts (disk) : nb d'opérations de tri avec au moins une écriture
- sorts (memory) : nb d'opérations de tri en mémoire

Entrepôt de données

Les requêtes analytiques permettent de faire des analyses sur de grands volumes de données.

Systèmes transactionnels vs Analytiques

Aspect	Operational DB	DW
User	clerk	manager
Interaction	short (s)	long analyses (min,h)
Type of interaction	Insert, Update, Delete	Read,periodically (bulk) inserts
Type of query	many simple queries	few, but complex queries (typically drill-down, slice...)
Query scope	a few tuples (often 1)	many tuples (range queries)
Concurrency	huge (thousands)	limited (hundreds)
Data source	single DB	multiple independant DB...
Schema	query-independant (3NF)	based on queries
Data	original, detailed, dynamic	derived,consolidated, inte- grated,historicized,partially aggregated stable
Size	MB,GB	TB,PB
Availability	crucial	not so crucial
Architecture	3-tier (ANSI-SPARC)	adapted to data integration

Les datawarehouse utilisent des schémas relationnels redondants, et violent en conséquence la clause 3NF.

Un fait est un enregistrement d'un événement se passant dans le monde réel.

Une dimension est un set d'attributs décrivant le fait.

Les mesures sont des éléments calculés des tables de faits, comme le chiffre d'affaires etc.

Les mesures peuvent être **Additives, Semi-Additives ou Non-Additives.**

- Additif : Mesures métier pouvant être agrégées sur toutes les dimensions
- Semi-Additif : Une mesure semi-additive est une mesure qui doit être additionnée pour certaines dimensions, mais pas toutes.

- Non-Additif : Mesures métier qui ne peuvent pas être agrégées dans n'importe quelle dimension (par exemple la marge en pourcentage, ou les ratios comme les étoiles des avis)

La liste des dimensions définissent **la granularité** de la table de fait. Toutes les mesures dans la table de faits **ont la même granularité**.

Les tables de **dimensions** ont une simple clé primaire. Ils contiennent autant d'attributs que possible. Ces tables de dimensions sont les points d'entrée dans la table des faits. Ils représentent **10%** de l'entrepôt de données. Elle contient plus de colonne que la table de fait.

Les opérateurs ROLLUP et CUBE :

Ils se placent dans le GROUP BY, et agissent comme une extension.

ROLLUP produit des agrégations cumulées. CUBE fait la même chose mais en allant plus loin. Exemple : **ROLLUP (YEAR, MONTH, DAY)**

Avec un ROLLUP il aura les résultats suivants :

```
YEAR, MONTH, DAY
YEAR, MONTH
YEAR
()
```

Avec CUBE il aura les caractéristiques suivantes :

```
YEAR, MONTH, DAY
YEAR, MONTH
YEAR, DAY
YEAR
MONTH, DAY
MONTH
DAY
()
```

La principale différence entre ROLLUP et CUBE est que ROLLUP regroupe les données et produit des résultats agrégés sur une seule dimension, tandis que CUBE produit des résultats agrégés sur plusieurs dimensions en même temps. ROLLUP calcule des sommes et des moyennes sur une seule dimension et CUBE calcule des sommes et des moyennes sur toutes les dimensions. L'utilisation de ROLLUP est plus limitée car il ne peut être utilisé que sur une seule dimension, tandis que CUBE peut être utilisé sur plusieurs dimensions.

Les étapes pour réaliser un entrepôt de donnée :

- Choisir le business à modéliser
- Trouver la granularité (moins de dimensions possible)
- Choisir les dimensions
- Réaliser le schéma en étoile
- Identifier les mesures

La dimension **date** est la seule dimension se trouvant dans tous les dataWarehouse.

La **date** est différente du **temps**.

- Les dimensions **causales** sont des dimensions qui contiennent des attributs qui sont susceptibles de changer d'autres attributs, par exemple une table promotion est une table causale, car elle peut changer des valeurs dans la table vente.
- Une dimension **corrélée** comprend d'autres dimensions, ou attributs qui pourraient être dans des dimensions séparées. Par exemple, la table promotion est une dimension corrélée si elle contient des réductions de prix, des coupons, et des pubs.
- Une dimension **dégénérée** est une dimension qui ne contient pas d'attributs utiles à part un id, par exemple les tickets de caisse.
- Une dimension **poubelle** est une dimension regroupant des dimensions non corrélées.

La normalisation des tables (snowflaking) de dimension permet de réduire les redondances dans les données, et permet aussi une meilleure maintenance des valeurs des tables de dimensions. Mais permettant de très faibles gains, on ne réalise des fois pas cette normalisation.

Snapshot périodique : Enregistrement régulièrement l'état complet d'un inventaire (base de données), tous les jours, toutes les semaines, toutes les deux heures ... L'inconvénient est que les tables sont denses (1 ligne pour chaque produit, jours, magasin..)

Des requêtes analytiques sur des snapshots peuvent être :

- La quantité totale d'un produit donné aujourd'hui
- La quantité totale d'un magasin aujourd'hui

Mais on ne peut pas répondre à des requêtes du types :

- Quantité totale d'un produit en juillet (on ne peut pas additionner des stocks)

Faits transactionnels : Enregistre toutes les transactions, par exemple chaque vente dans un magasin etc. Solution très coûteuse, utilisée par amazon. il faut une dimension spéciale pour les types de transactions.

Enregistrement Updated : Permet de mettre à jour des données, par exemple l'attribut "status" d'une commande peut prendre les valeurs "pending", "ok", "delivered" etc. Ces enregistrements update sont aussi appelés snapshots cumulés.

CHARACTERISTIC	TRANSACTION GRAIN	updated records	
		PERIODIC SNAPSHOT GRAIN	ACCUMULATING SNAPSHOT GRAIN
Time period represented	Point in time	Regular, predictable intervals	Indeterminate time span, typically short-lived
Grain	One row per transaction event	One row per period	One row per life
Fact table loads	Insert	Insert	Insert and update
Fact row updates	Not revisited	Not revisited	Revisited whenever activity
Date dimension	Transaction date	End-of-period date	Multiple dates for standard milestones
Facts	Transaction activity	Performance for predefined time interval	Performance over finite lifetime

Les **dimensions partagées** sont des dimensions partagées entre plusieurs datamart.

Une **vue** dans une base de données est une synthèse d'une requête d'interrogation de la base. On peut la voir comme une table virtuelle, définie par une requête.

Les avantages des vues sont :

- d'éviter de taper une requête très longue : la vue sert à donner un nom à la requête pour l'utiliser souvent,
- de masquer certaines données à certains utilisateurs. En SQL, les protections d'une vue ne sont pas forcément les mêmes que celles des tables sous-jacentes.

Les vues matérialisées : A la différence d'une vue standard, dans une vue matérialisée les données sont dupliquées. On l'utilise à des fins d'optimisation et de performance dans le cas où la requête associée est particulièrement complexe ou lourde, ou pour faire des répliqués de table.

Les **hiérarchies** sont mises en place grâce à des tables bridges, soit entre deux dimensions, soit entre la dimension et la table de fait. Elle contient l'id de la première et de la deuxième dimension, une distance, un bottom flag et un top flag.

La dimension la plus importante est la dimension utilisateur, qui peut compter plusieurs millions de lignes, et des centaines de colonnes. Elle est aussi fréquemment mise à jour. Certaines agences marketing ont des centaines d'attributs pour la table User, et pour de grandes tables, il faut absolument snowflaking, en normalisant les dimensions, mais aussi partitionner les dimensions par colonnes, lignes ou les deux, et aussi la table de faits par ligne.

Le **snowflaking** est utilisé uniquement sur les très grosses tables.

Le partitionnement est le fait de séparer une grosse table en plusieurs petites. Cela reste différent du snowflaking.

Le partitionnement par lignes est basé sur des valeurs dans une ou plusieurs colonnes, comme la dimension temps, types de profils clients, produits ou taille de la table.

Le partitionnement par colonnes est basé sur des valeurs dans une ou plusieurs colonnes, comme la dimension temps, types de profils clients, produits ou taille de la table. Le but est d'avoir plusieurs tables de même nombre de lignes. On peut créer des tables du type, souvent modifiées, modifiées quelquefois, tout le temps modifiées etc.

Le **partitionnement hybride** se fait par lignes et par colonnes.

Pour mettre à jours des dimensions il ya 3 options :

- Réécrire la valeur (perds l'historique)
- Versionner les lignes (Mettre un numéro de version)
- Dupliquer les attributs (historique partiel)

On peut aussi utiliser les mini-dimensions qui comprennent les attributs qui changent souvent. il y a donc une clé primaire

OLAP = ON-LINE ANALYTICAL QUERY PROCESSING

Vues matérialisées :

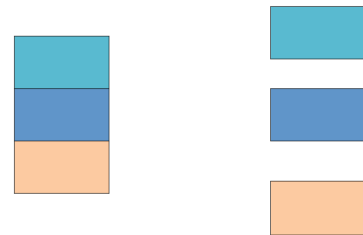
Avantages :

- réduit le temps de la requête
- Le calcul de la requête et la compilation est fait qu'une seule fois
- Peut être utilisée pour plusieurs requêtes

Inconvénients :

- Il faut un espace de stockage
- Elle doit être maintenue

1) Partitioning by-Row



- Il faut choisir la meilleure vue
- Besoin de reformuler les algorithmes

Query Reformulation Using Views

Problem

Input

- A (set of) relations **F**
- A (set of) views **V**
- A (set of) user queries **Q**

Output

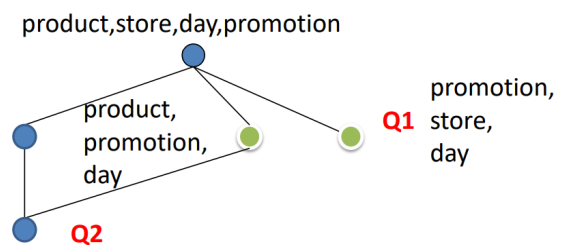
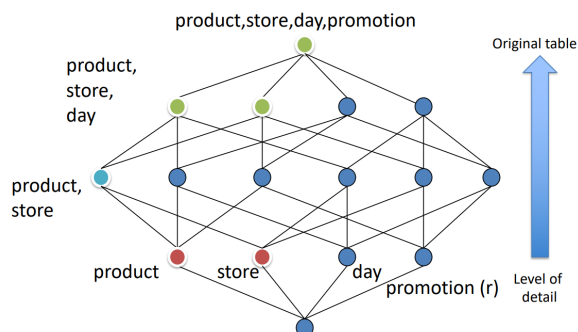
- Q^{ref} such that $Q^{ref}(V) = Q(F)$

Query Reformulation

Possible to reformulate a query with a view when :

- The view has selection-conditions that are compatible with those of the queries
- The view has dimensions that are compatible with those of the queries
- In this case, we say that the view is relevant.

Aggregation lattice



Il faut avec le treillis chercher les nœuds et donc les joins qui peuvent répondre au plus grand nombre de requêtes. On colorie en une couleur les éléments qui sont dans le GROUP BY d'une requête.

Les index : un index est une structure de données utilisée et entretenue par le système de gestion de base de données (SGBD) pour lui permettre de retrouver rapidement les données. L'utilisation d'un index simplifie et accélère les opérations de recherche, de tri, de jointure ou d'agrégation effectuées par le SGBD.

L'index placé sur une table va permettre au SGBD d'accéder très rapidement aux enregistrements, selon la valeur d'un ou plusieurs champs.

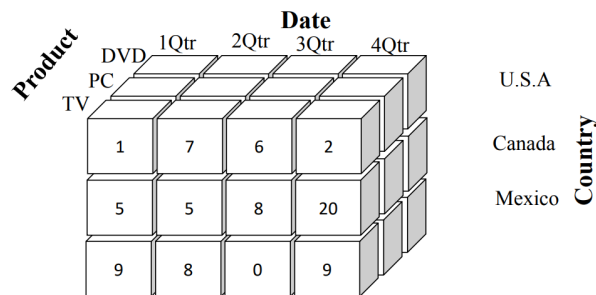
Les méthodes d'accès communes sont :

- B+ trees
- inverted lists
- bit map indexes
- join indexes

Les bases de données multidimensionnelles :

Elles sont sous forme de datacube :

A Sample Data Cube



Typical OLAP Operations

- **Roll up** (drill-up): summarize data
 - by climbing up hierarchy or by dimension reduction
- **Drill down** (roll down): reverse of roll-up
 - from higher level summary to lower level summary or detailed data, or introducing new dimensions
- **Slice and dice:**
 - project and select
- **Pivot (rotate):**
 - reorient the cube, visualization, 3D to series of 2D planes.
- Other operations
 - **drill across:** involving (across) more than one fact table
 - **drill through:** through the bottom level of the cube to its back-end relational tables

GROUP-BY
ROLL-UP
CUBE

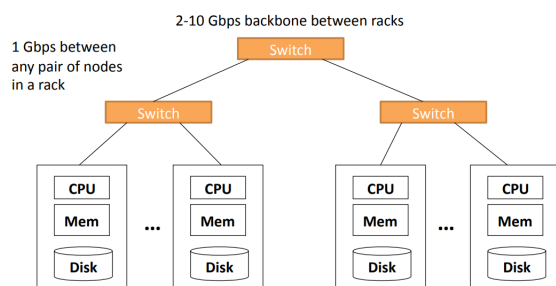
Technologies du BIG-DATA

Hadoop est la solution de parallélisation de Google aux problèmes de base de données.

Sur google, c'est 5.6 milliards de requêtes par jour. L'accès au disque est lent et on ne peut pas attendre. **L'architecture cluster** permet de séparer les données sur plusieurs disques, afin de lancer des requêtes en parallèle.

Plusieurs systèmes d'aujourd'hui utilisent les clusters, sur des racks, Shared-Nothing avec aucune mémoire partagée ...

Cluster Architecture



Des données massives ainsi que : base de données parallèles / Entrepôt de données c'est :

- Cher à configurer et à entretenir
- Cher à préparer et charger les données ETL
- Contraint à l'utilisation d'SQL.

ETL : Extraction, transformation, chargement (ETL), un processus automatisé qui prend les données brutes, extrait l'information nécessaire à l'analyse, la transforme en un format qui peut répondre aux besoins opérationnels et la charge dans un Data Warehouse. L'ETL résume généralement les données afin de réduire leur taille et d'améliorer leur performance pour des types d'analyse spécifiques.

Hadoop : Hadoop est un framework libre et open source écrit en Java destiné à faciliter la création d'applications distribuées (au niveau du stockage des données et de leur traitement) et échelonnables (scalables) permettant aux applications de travailler avec des milliers de nœuds et des pétaoctets de données.

Le **HDFS** est un système de fichier distribué (garde les datas)

Map-reduce est un paradigme de programmation.

Le MAP-REDUCE :

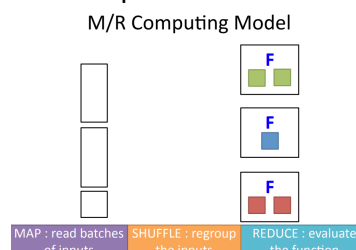
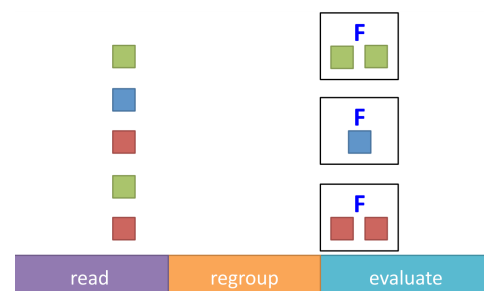
MapReduce est un patron de conception de développement informatique, inventé par Google, dans lequel sont effectués des calculs parallèles, et souvent distribués, de données potentiellement très volumineuses, typiquement supérieures en taille à un téraoctet. Les termes « map » et « reduce », et les concepts sous-jacents, sont empruntés aux langages de programmation fonctionnelle utilisés pour leur construction (map et réduction de la programmation fonctionnelle et des langages de programmation tableau).

Aucun problème n'est parfaitement parallélisable. Beaucoup introduisent des dépendances et des communications.

On va donc utiliser le MAP/REDUCE. Les étapes sont les suivantes :

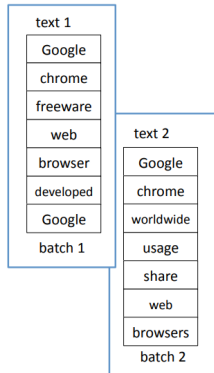
- Lire les inputs et les assigner à des groupes (c'est la phase MAP, elle peut être faite en parallèle)
- Regrouper les inputs selon les critères du MAP (peut aussi être fait en parallèle)
- Évaluer une fonction sur les inputs groupés (toujours possible en parallèle)

Toutes ces étapes peuvent être faites en parallèle.

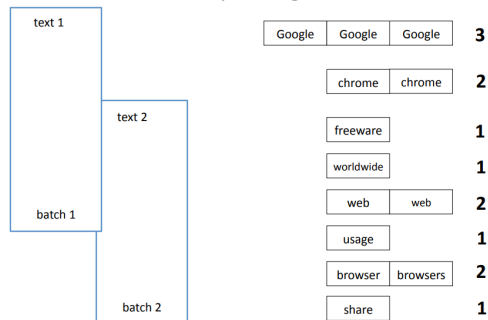


Exemple :

M/R Computing Model



M/R Computing Model

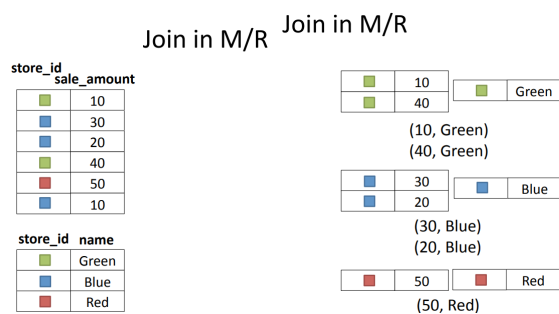


Hadoop n'est pas un système de management de datas, c'est juste un framework. Mais il peut quand même implémenter des requêtes, il peut grouper facilement mais n'est parfois pas plus performant qu'un entrepôt de données classique. Mais il est plus facile à mettre en place et utiliser et est aussi plus flexible.

Par exemple pour la requêtes : `SELECT store_id, sum(sale_amount) FROM sales GROUP BY store_id;`



Pour `SELECT store_name, sale_amount FROM sales, store WHERE sales.store_id = store.store_id;`



Après multiplication de matrices :

Matrix Multiplication

Matrix A (2*3)

L1
L2

$$1+1+1=3$$

Matrix B (3*2)

C1 C2

$$4+4+4=12$$

MAP REDUCE Programming :

Les opération sont exprimées en utilisant des pairs : <key, value>

Reduce(key k_{group} , Set<value> V)
→ Set<key,value>

Read : reduce function takes in input a key and a set of values and outputs a set of key-value pairs

All (k_{group}, v)-pairs for a given k_{group} are evaluated by the same reducer !

Wordcount : k_{group} is a word V number of occurrences
Trends : k_{group} is a word V number of occurrences
Group by : k_{group} is a group-by attribute-value V a set of tuples
Join : k_{group} is a join attribute-value V a set of tuples
Similarity : k_{group} is a user-user pair id V two users' profiles

```
map(key k, value phrase):
for each word in phrase :
emit( word , 1 ) //generates a <key,value> pair
reduce(key word, values occurrences):
emit( key , occurrences.size() )
```

Ici on peut compter le nombre de mots utilisés.

phrase 1		Google,1	Google,1	Google,1	3
Google					
chrome		chrome,1	chrome,1		2
freeware		freeware,1			1
web	phrase 2	worldwide,1			1
browser	Google	web,1	web,1		2
developed	chrome	usage,1			1
Google	worldwide	browser,1	browser,1		2
	usage	share,1			1
	share				
	web				
	browser				

Group By avec MAP/REDUCE :

```
map(key k, value tuple):
emit( tuple.store_id , tuple.sale_amount )
reduce(key store_id, values sales):
total_sales=0;
for each s in sales
total_sales+=s;
emit( store_id , total_sales )
```

Group By in M/R

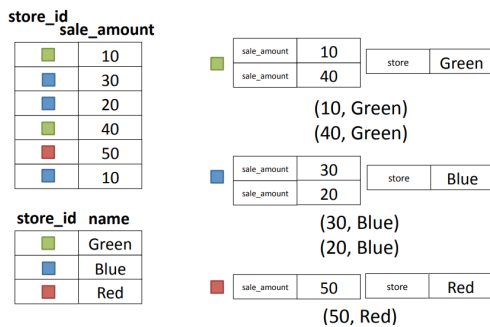
store_id	sale_amount	
1	10	
2	30	
3	20	
4	40	
5	50	
6	10	
7	80	
8	60	

store_id	sale_amount	
1	SUM(sale_amount) = 100	
2	30	
3	20	
4	10	
5	80	
6	50	

Utilisation d'un join :

```
map(key k, value tuple):
if tuple belongs to SALES
  emit( t.store_id , <"profit", t.sale_amount> )
if tuple belongs to STORES
  emit( t.store_id , <"store", t.store_name> )
reduce(key store_id, values mixed-attributes):
for each a in mixed-attributes
  for each b in mixed-attributes
    if a[1]=="sale" and b[1]=="store"
      emit( store_id , <a[2],b[2]>
```

Join in M/R



Map-Reduce Programming

There is not a single line of code dedicated to parallelization !!

Map-Reduce environment takes care of:

- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the **group by key/shuffle** step
- Handling machine failures
- Managing required inter-machine communication

Mesure des coûts d'un MAP/REDUCE :

- Coût de communication (total des entrées sorties de chaque process)
- Coût de calcul (temps total d'utilisation du CPU des tous les process)

C'est important car sur un cloud public, on paye pour les calculs et les communications, donc il faut réduire ces consommations.

Il faut choisir la bonne taille de réduction comme pour le problèmes des 3000 drogues ou nous cherchons les interactions, on pourrait faire 100 paquets de 30.

- 3000 utilisateurs → 3000 tasks
- Chacun envoi 2999 copies de chaque enregistrements d'utilisateurs
- Qui font 1KB, ce qui fait 9GB de données sur un réseau de 1Gb, donc 90 secondes de communication.

Les **systèmes de fichiers distribués** stockent les datas distribuées, les fichiers sont séparés dans des chunks, de 64MB. Chaque chunk est répliqué au moins 2 ou 3 fois. On essaie de garder ces répliques dans des racks différents. Ensuite le **Master** stocke les metadata et les localisations de chaque chunks, et permet d'aiguiller.

Un client va se connecter et parler au maître qui va lui retourner l'emplacement du chunk. Ensuite il parlera directement avec le chunk pour accéder aux données.

Question 1 La compagnie souhaite analyser des statistiques d'appel très détaillées.

Une statistique d'appel doit contenir les informations suivantes.

- L'émetteur (numéro ou identifiant de l'appelant)
- Le destinataire (numéro ou identifiant de l'appelé)
- La date et l'heure du début d'appel
- La durée d'appel
- Le type d'appel (Mobile, SMS, Fax, Fixe, ...)
- Le compte facturé pour l'appel (attention : pas toujours l'émetteur)
- Un identifiant unique pour chaque statistique d'appel
- Un code correspondant au dysfonctionnement/erreur éventuellement produit pendant l'appel

Travail demandé :

1. Proposer un schéma logique et physique pour l'entrepôt de données
2. Justifier la (semi,non)-additivité des mesures
3. Proposer 5 requêtes analytiques pertinentes
4. Estimer la taille de l'entrepôt sur 10 ans

1) Schéma logique :

```
statistiques_appel (  
  id INTEGER PRIMARY KEY,  
  emetteur CHAR(255),  
  destinataire CHAR(255),  
  date_heure_debut DATETIME,  
  duree INTEGER,  
  type CHAR(255),  
  compte_facture CHAR(255),  
  code CHAR(255)  
)
```

Schéma physique :

```
CREATE TABLE statistiques_appel (  
  id INTEGER PRIMARY KEY AUTO_INCREMENT,  
  emetteur CHAR(255) NOT NULL,  
  destinataire CHAR(255) NOT NULL,  
  date_heure_debut DATETIME NOT NULL,  
  duree INTEGER NOT NULL,  
  type CHAR(255) NOT NULL,  
  compte_facture CHAR(255) NOT NULL,  
  code CHAR(255)  
);
```

- 1) La plupart des mesures sont non-additives comme le destinataire, le type d'appels, le compte facturé, ou le code de dysfonctionnement.

Nombre total d'appels passés par chaque emetteur sur une période donnée :

```
SELECT emetteur, COUNT(*) as nombre_appels
FROM statistiques_appel
WHERE date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY emetteur
```

Durée totale des appels passés par chaque emetteur sur une période donnée :

```
SELECT emetteur, SUM(duree) as duree_totale
FROM statistiques_appel
WHERE date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY emetteur
```

Nombre total d'appels reçus par chaque destinataire sur une période donnée :

```
SELECT destinataire, COUNT(*) as nombre_appels
FROM statistiques_appel
WHERE date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY destinataire
```

Pourcentage d'appels ayant abouti à un dysfonctionnement/erreur par heure de la journée sur une période donnée :

```
SELECT HOUR(debut) as heure,
       SUM(CASE WHEN code_dysfonctionnement IS NOT NULL THEN 1 ELSE 0
END) / COUNT(*) as pourcentage_dysfonctionnements
FROM statistiques_appel
WHERE date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY HOUR(debut)
```

Nombre total d'appels passés par chaque type d'appel sur une période donnée :

```
SELECT type, COUNT(*) as nombre_appels
FROM statistiques_appel
WHERE date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY type
```

4) En supposant que le nombre d'appels et la durée moyenne des appels restent constants sur cette période, on peut estimer la taille de l'entrepôt de données en multipliant le nombre d'appels par année par la taille moyenne des enregistrements de données, puis en multipliant ce résultat par 10. Par exemple, si la compagnie passe 100 000 appels par année et que chaque enregistrement de données associé à un appel occupe en moyenne 100 octets, alors la taille de l'entrepôt de données sera d'environ $100\,000 * 100 * 10 = 10\,000\,000\,000$ octets, soit environ 9,5 Go. Cette estimation est bien sûr très approximative et devrait être ajustée en fonction des données réelles de l'entreprise.

Question 2 - La compagnie souhaite mettre en place un système d'analyse des factures client.

Une facture doit contenir les informations suivantes.

- Le compte client (on considère que chaque client dispose d'un seul numéro)
- La période de référence
- Le coût (mensuel) des abonnements souscrits
- Le coût total des communications effectuées (on ne considère que trois types de communications : mobile, livebox, et poste fixe)
 - Incluses dans le forfait
 - Hors forfait
- Un identifiant unique pour chaque facture

Travail demandé :

1. Proposer un schéma logique et physique pour l'entrepôt de données
2. Justifier la (semi,non)-additivité des mesures
3. Proposer 5 requêtes analytiques pertinentes
4. Estimer la taille de l'entrepôt sur 10 ans

1) Voici un exemple de schéma logique pour l'entrepôt de données des factures client :

```
factures (  
  id INTEGER PRIMARY KEY,  
  compte_client CHAR(255),  
  periode_reference DATE,  
  cout_abonnements INTEGER,  
  cout_communications INTEGER  
)
```

```
cout_communications_detail (  
  facture_id INTEGER,  
  type CHAR(255),  
  inclus_forfait BOOLEAN,  
  cout INTEGER,  
  FOREIGN KEY (facture_id) REFERENCES factures (id)  
)
```

Schéma physique :

```
CREATE TABLE factures (  
  id INTEGER PRIMARY KEY AUTO_INCREMENT,  
  compte_client CHAR(255) NOT NULL,  
  periode_reference DATE NOT NULL,  
  cout_abonnements INTEGER NOT NULL,  
  cout_communications INTEGER NOT NULL  
);  
  
CREATE TABLE cout_communications_detail (  
  facture_id INTEGER,  
  type CHAR(255),  
  inclus_forfait BOOLEAN,  
  cout INTEGER,  
  FOREIGN KEY (facture_id) REFERENCES factures (id)  
)
```

```
facture_id INTEGER NOT NULL,
type CHAR(255) NOT NULL,
inclus_forfait BOOLEAN NOT NULL,
cout INTEGER NOT NULL,
FOREIGN KEY (facture_id) REFERENCES factures (id),
PRIMARY KEY (facture_id, type)
);
```

- 2) **Le coût (mensuel)** des abonnements souscrits peut être considéré comme une mesure semi-additive, car il peut être correctement additionné lorsque les abonnements sont souscrits par le même client, mais pas lorsqu'ils sont souscrits par des clients différents.

Le coût total des communications effectuées peut être considéré comme une mesure semi-additive, car il peut être correctement additionné lorsque les communications sont effectuées par le même client, mais pas lorsqu'elles sont effectuées par des clients différents.

L'identifiant unique pour chaque facture ne peut pas être considéré comme une mesure additive, car il s'agit d'un identifiant unique qui ne peut pas être additionné avec d'autres valeurs.

Montant total des abonnements souscrits par chaque compte client sur une période donnée:

```
SELECT compte_client, SUM(cout_abonnements) as total_abonnements
FROM factures
WHERE periode_reference BETWEEN '<debut_pperiode>' AND '<fin_pperiode>'
GROUP BY compte_client;
```

Montant total des communications mobiles hors forfait effectuées par chaque compte client sur une période donnée :

```
SELECT compte_client, SUM(cout_mobile_hors_forfait) as
total_communications_mobile
FROM factures
WHERE periode_reference BETWEEN '<debut_pperiode>' AND '<fin_pperiode>'
GROUP BY compte_client;
```

Pourcentage de factures avec des communications hors forfait par rapport au total des factures sur une période donnée :

```
SELECT COUNT(*) as nombre_factures, SUM(cout_mobile_hors_forfait +
cout_livebox_hors_forfait + cout_fixe_hors_forfait > 0) as
nombre_factures_hors_forfait
FROM factures
WHERE periode_reference BETWEEN '<debut_pperiode>' AND '<fin_pperiode>';
SELECT (nombre_factures_hors_forfait / nombre_factures) * 100 as
pourcentage_factures_hors_forfait;
```

Montant moyen des factures par compte client sur une période donnée :

```
SELECT compte_client, AVG(cout_total) as cout_moyen
FROM factures
WHERE periode_reference BETWEEN '<debut_periode>' AND '<fin_periode>'
GROUP BY compte_client;
```

Nombre de factures par periode de reference sur une période donnée :

```
SELECT periode_reference, COUNT(*) as nombre_factures
FROM factures
WHERE periode_reference BETWEEN '<debut_periode>' AND '<fin_periode>'
GROUP BY periode_reference;
```

Voici quelques éléments à considérer pour déterminer la taille de l'entrepôt :

Nombre de factures : combien de factures pensez-vous générer chaque année ? Cela dépendra du nombre de clients que vous avez et de la fréquence à laquelle vous envoyez des factures (par exemple, mensuellement, trimestriellement, etc.).

Taille de chaque facture : combien de données chaque facture contient-elle ? Cela dépendra de la quantité de communications effectuées et des abonnements souscrits par chaque client.

Durée de stockage : pendant combien de temps souhaitez-vous stocker les factures ? Si vous avez l'intention de les conserver pendant 10 ans, vous devrez prévoir un espace de stockage suffisant pour accueillir toutes ces données.

En utilisant ces informations, vous devriez pouvoir estimer la taille de l'entrepôt de données sur 10 ans en multipliant le nombre de factures prévues par an par la taille de chaque facture et en multipliant le résultat par la durée de stockage

Hierarchies

Un leader mondial dans la construction d'appareils photo souhaite faire évoluer sa propre offre de produits sur la base de données de la plateforme Flickr. Précisément, l'objectif est de concevoir un entrepôt de données permettant d'analyser l'utilisation des appareils par le biais des photos publiées sur Flickr. L'entrepôt doit permettre d'étudier les lieux ainsi que les périodes de l'année et les horaires de la journée où les appareils sont utilisés, mais aussi le lignage des appareils photos (par exemple, le modèle Nikon D3200 dérive du modèle Nikon D3100 qui dérive du D3000), et le créateur de chaque modèle (par exemple, le modèle Nikon D3200 a été conçu par Eiji Fumio).

1) Proposez un schéma d'entrepôt de données permettant les analyses suivantes.

1. compter le nombre de photos réalisées pour chaque appareil photo ;
2. compter le nombre de photos prises par un appareil conçu par Eiji Fumio ;
3. pour tous les modèles dérivés du Nikon D3000, compter le nombre de photos réalisées par jour
4. indiquer les modèles historiquement les plus influents (on considère ici les appareils au sommets des hiérarchies de lignage).

Voir TP

```
SELECT marque, modele, COUNT(*)
FROM Appareils_Dimension, Captures_Fait
WHERE Appareils_Dimension.id = Captures_Fait.idAppareil
GROUP BY modele, marque;
```

```
SELECT createur, COUNT(*)
FROM Appareils_Dimension, Captures_Fait
WHERE Appareils_Dimension.id = Captures_Fait.idAppareil
AND createur = 'Eiji FUMIO'
GROUP BY createur;
```

```

SELECT marque, modele
FROM Appareils_Dimension, Captures_Fait, Appareils_Bridge
WHERE Appareils_Dimension.id = Captures_Fait.idAppareil
AND Appareils_Dimension.id = Appareils_Bridge.idAppareil
AND topFlag = 1
GROUP BY modele, marque
HAVING COUNT(*) >= ALL(
SELECT COUNT(*)
FROM Captures_Fait
GROUP BY idAppareil
);

```

Questions ouvertes

(réponse 10 lignes max)

Quelle est la différence entre une base de données relationnelles et un entrepôt de données ?

Pourquoi les bases de données relationnelles ne sont pas adaptées à la gestion des données massives ?

Pourquoi a-t-on introduit les plateformes de Big-Data ? Quels sont les avantages et les inconvénients par rapport aux entrepôts de données ?

Pourquoi est-il nécessaire d'optimiser l'évaluation des requêtes dans les bases de données relationnelles ? Illustrez l'intérêt de l'optimisation avec un exemple

- 1) Une base de données est souvent utilisée pour stocker des données structurées et peut être interrogée à l'aide de langages de requête tels que SQL. Un entrepôt de données, en revanche, est un système conçu pour stocker de grandes quantités de données. Les entrepôts de données sont souvent utilisés pour l'analyse de données en grande quantité. Ils sont généralement conçus pour être facilement interrogés à l'aide de requêtes SQL ou de langages de requête spécialisés. En résumé, les bases de données relationnelles sont principalement utilisées pour stocker des données structurées, tandis que les entrepôts de données sont principalement utilisés pour l'analyse de données en grande quantité.
- 2) **Performances** : Les bases de données relationnelles ont tendance à devenir de plus en plus lentes à mesure qu'elles grossissent, ce qui peut poser problème lorsqu'il s'agit de traiter de grandes quantités de données en temps réel.

Structure de données rigide : Les bases de données relationnelles sont basées sur une structure de tables rigide, ce qui peut rendre difficile l'ajout de nouvelles données à mesure qu'elles arrivent.

Schéma fixe : Dans une base de données relationnelle, le schéma de la base de données (c'est-à-dire la structure des tables et des relations entre elles) doit être défini à l'avance, ce qui peut s'avérer difficile lorsque les données sont très hétérogènes ou que leur structure change fréquemment.

Pour ces raisons, les entrepôts utilisent des architectures distribuées et des technologies de stockage spécialisées pour permettre un traitement rapide de grandes quantités de données, même lorsqu'elles sont hétérogènes et en constante évolution.

- 3) Les plateformes de Big Data ont été introduites pour permettre de traiter de grandes quantités de données de manière plus efficace et à moindre coût. Elles sont généralement basées sur des architectures distribuées et utilisent des technologies de stockage spécialisées pour permettre un traitement rapide de données volumineuses. Les avantages :

Capacité de traiter de très grandes quantités de données : Les plateformes de Big Data peuvent traiter de très grandes quantités de données, même lorsqu'elles sont hétérogènes et en constante évolution.

Coût réduit : Les plateformes de Big Data permettent de traiter de grandes quantités de données à un coût inférieur à celui des entrepôts de données traditionnels, grâce à l'utilisation de matériel moins coûteux et à une meilleure utilisation des ressources.

Agilité : Les plateformes de Big Data sont généralement plus souples et plus faciles à mettre en œuvre que les entrepôts de données traditionnels, ce qui en fait une option intéressante pour les entreprises qui ont besoin de mettre en place rapidement une solution de gestion de données.

Cependant, il y a aussi des inconvénients à utiliser une plateforme de Big Data :

Complexité : Les plateformes de Big Data peuvent être complexes à mettre en œuvre et à gérer, en particulier pour les entreprises qui n'ont pas l'expertise en interne.

Coût de la formation : Il peut être coûteux de former les employés à utiliser une plateforme de Big Data, en particulier si l'entreprise n'a pas l'expertise en interne.

En résumé, les entrepôts de données sont principalement utilisés pour l'analyse de données en grande quantité, tandis que les plateformes de Big Data sont conçues pour traiter de très grandes quantités de données de manière efficace.

- 4) Il est important d'optimiser l'évaluation des requêtes dans les bases de données relationnelles pour plusieurs raisons :

Performances : Les requêtes peuvent être très complexes et prendre un temps considérable à s'exécuter si elles ne sont pas optimisées. L'optimisation des requêtes permet de les exécuter de manière plus rapide et de réduire les temps de réponse.

Utilisation des ressources : Les bases de données relationnelles peuvent être utilisées par de nombreux utilisateurs en même temps, ce qui peut entraîner une forte utilisation des ressources si les requêtes ne sont pas optimisées. L'optimisation des requêtes permet de réduire l'utilisation des ressources et d'améliorer les performances globales de la base de données.

Coût : Les bases de données relationnelles peuvent être coûteuses à maintenir et à faire évoluer, en particulier si elles sont fortement utilisées. L'optimisation des requêtes permet de réduire le coût global de la base de données en améliorant ses performances et en réduisant l'utilisation des ressources.

Un exemple d'intérêt de l'optimisation des requêtes :

Imaginons une base de données relationnelle contenant des millions d'enregistrements et que vous souhaitez exécuter une requête pour récupérer les enregistrements correspondant à un certain critère (par exemple, les enregistrements d'une table "Clients" ayant un code postal spécifique). Si la requête n'est pas optimisée, le moteur de base de données devra parcourir chaque enregistrement de la table pour trouver ceux qui correspondent au critère de recherche. Cela peut prendre un temps considérable et entraîner une forte utilisation des ressources.

En optimisant la requête, par exemple en créant un index sur la colonne "code postal", le moteur de base de données peut trouver rapidement les enregistrements correspondants sans avoir à parcourir l'ensemble de la table. Cela permet de réduire les temps de réponse et d'améliorer les performances de la base de données.

Vues Matérialisées

1) Sur la base du schéma d'entrepôt de données que vous avez proposé pour Amazon (TD DW1), traduire les interrogations suivantes en requêtes SQL.

Le nombre de produits par pays achetés après 22h

Le nombre de produits achetés pour chaque heure de la journée

Le nombre de produits achetés après 22h par des clients ayant un compte Amazon premium

Le nombre de produits achetés après 21h par des clients français

- Proposer deux (2) ensembles de vues matérialisées permettant d'optimiser ce (petit) workload. Pour cela, il faudra d'abord construire le treillis d'aggrégation.

2) Considérons le workload {Q1,Q2,Q3,Q4}.

Calculer le treillis d'agrégation pour le workload.

Proposer un ensemble de vues matérialisées permettant de répondre aux requêtes.

Donner la réécriture des requêtes {Q1,Q2,Q3,Q4} sur l'ensemble des vues.

Amazon Voir TP.

1)

```
SELECT SUM(id_Produits), id_Pays FROM Ventes, Clients, Produits WHERE
Ventes.id_Produit = Produit.id_Produit AND Ventes.id_Clients = Clients.id_Clients
GROUP BY Clients.Pays;
```

2)

```
SELECT SUM(id_Produits), Date.Heure FROM Ventes, Date, Produits WHERE
Ventes.id_Produit = Produit.id_Produit AND Ventes.id_Date = Date.id_Date GROUP
BY Date.Heure;
```

3)

```
SELECT SUM(id_Produits), Date.Heure, Clients.id_Client FROM Ventes, Date,
Produits, Client WHERE Ventes.id_Produit = Produit.id_Produit AND Ventes.id_Date
```

```
= Date.id_Date AND Ventes.id_Client = Client.id_Client AND Client.Premium = 1  
AND Date.Heure > 22 GROUP BY Date.Heure, Clients.id_Client;
```

4)

```
SELECT SUM(id_Produits), Date.Heure, Client.Pays FROM Ventes, Date, Produits,  
Client WHERE Ventes.id_Produit = Produit.id_Produit AND Ventes.id_Date =  
Date.id_Date AND Ventes.id_Client = Client.id_Client AND Date.Heure > 21 AND  
Client.Country = 'FRANCE' GROUP BY Date.Heure, Client.Pays;
```

Q1 :

```
SELECT Magasin, Ville, avg(montant_vente)
```

```
FROM Ventes
```

```
GROUP BY Magasin, Ville
```

Q2 :

```
SELECT Ville, Produit, Date, sum(montant_vente)
```

```
FROM Ventes
```

```
GROUP BY Ville, Produit, Date
```

Q3 :

```
SELECT Ville, sum(montant_vente)
```

```
WHERE Date BETWEEN '01/01/2017' AND '01/01/2018'
```

```
FROM Ventes
```

```
GROUP BY Ville
```

Q4 :

```
SELECT Date, max(montant_vente)
```

```
FROM Ventes
```

```
GROUP BY Date
```

Une vue matérialisée qui calcule la moyenne des montants de vente par magasin et par ville:

```
CREATE MATERIALIZED VIEW vue_agregate_magasin_ville AS  
SELECT Magasin, Ville, avg(montant_vente) as avg_montant_vente  
FROM Ventes  
GROUP BY Magasin, Ville;
```

Une vue matérialisée qui calcule la somme des montants de vente par ville, par produit et par date :

```
CREATE MATERIALIZED VIEW vue_agregate_ville_produit_date AS  
SELECT Ville, Produit, Date, sum(montant_vente) as sum_montant_vente  
FROM Ventes  
GROUP BY Ville, Produit, Date;
```

Une vue matérialisée qui calcule la somme des montants de vente par ville et par date :

```
CREATE MATERIALIZED VIEW vue_agregate_ville_date AS  
SELECT Ville, Date, sum(montant_vente) as sum_montant_vente  
FROM Ventes  
GROUP BY Ville, Date;
```

Une vue matérialisée qui calcule le montant de vente maximum par date :

```
CREATE MATERIALIZED VIEW vue_agregate_date AS
```

```
SELECT Date, max(montant_vente) as max_montant_vente
FROM Ventes
GROUP BY Date;
```

Les requêtes Q1, Q2, Q3 et Q4 peuvent être réécrites sur l'ensemble de ces vues matérialisées de la manière suivante :

```
Q1 : SELECT Magasin, Ville, avg_montant_vente FROM vue_agregate_magasin_ville;
Q2 : SELECT Ville, Product, Date, sum_montant_vente FROM
vue_agregate_ville_produit_date;
Q3 : SELECT Ville, sum_montant_vente FROM vue_agregate_ville_date WHERE Date BETWEEN
'01/01/2017' AND '01/01/2018';
Q4 : SELECT Date, max_montant_vente FROM vue_agregate_date;
```

Optimisation

- Expliquer le plan d'exécution suivant fourni par le SGBD Oracle.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3	261	69(0)	00:00:01
1	NESTED LOOPS		3	261	69 (0)	00:00:01
2	TABLE ACCESS BY INDEX ROWID	DEPARTEMENT	1	31	1 (0)	00:00:01
3	INDEX UNIQUE SCAN	SYS_C00251319	1	1	1 (0)	00:00:01
4	TABLE ACCESS FULL	VILLE	3	168	68 (0)	00:00:01

Predicate Information (identified by operation id):

```
3 - access("DEP"."ID"='91')
4 - filter("V"."DEP"='91')
```

- Dessiner l'arbre ou donner l'expression algébrique du plan d'exécution logiques fourni par Oracle ci-dessus et proposer un autre plan d'exécution logique que celui d'Oracle. Indiquer celui qui est optimal en argumentant.

Le plan d'exécution fourni par le SGBD Oracle est une représentation de la manière dont la requête sera exécutée par le moteur de base de données. Il est généralement utilisé pour comprendre comment la requête est traitée et pour identifier les éventuels points de blocage ou de performance.

Voici une explication détaillée de chaque étape du plan d'exécution :

- L'étape 0 (SELECT STATEMENT) représente la requête SELECT en elle-même.
- L'étape 1 (NESTED LOOPS) représente une jointure en boucle imbriquée, qui consiste à parcourir une table et à chercher chaque ligne correspondante dans une autre table.
- L'étape 2 (TABLE ACCESS BY INDEX ROWID) représente l'accès à la table DEPARTEMENT en utilisant l'index SYS_C00251319. Cet index est unique, ce qui signifie qu'il ne peut y avoir qu'une seule ligne correspondant à chaque valeur dans l'index.
- L'étape 3 (INDEX UNIQUE SCAN) représente la recherche de la ligne correspondante dans l'index unique SYS_C00251319. La partie "Predicate Information" indique que l'accès à cette ligne est filtré en fonction de la colonne "ID" de la table DEPARTEMENT, et que la valeur de cette colonne doit être égale à "51".
- L'étape 4 (TABLE ACCESS FULL) représente l'accès complet à la table VILLE, c'est-à-dire que toutes les lignes de la table seront parcourues pour trouver les lignes correspondantes. La partie "Predicate Information" indique que les lignes retournées sont filtrées en fonction de la colonne "DEP" de la table VILLE, et que la valeur de cette colonne doit être égale à "91".

En résumé, le plan d'exécution indique que la requête effectue une jointure en boucle imbriquée entre la table DEPARTEMENT et la table VILLE, en utilisant l'index unique SYS_C00251319 de la table DEPARTEMENT pour accélérer la recherche. La table VILLE est parcourue entièrement pour trouver les lignes correspondantes. Les lignes de la table DEPARTEMENT sont filtrées en fonction de la colonne "ID", tandis que les lignes de la table VILLE sont filtrées en fonction de la colonne "DEP".

Voici l'expression algébrique du plan d'exécution logique correspondant à votre plan d'exécution Oracle :

```
SELECT *  
FROM DEPARTEMENT d  
JOIN VILLE v ON d.1D = v.DEP  
WHERE d.1D = 51 AND v.DEP = 91;
```

Cette expression indique que la requête effectue une jointure entre les tables DEPARTEMENT et VILLE en utilisant la colonne "1D" de la table DEPARTEMENT et la colonne "DEP" de la table VILLE comme clé de jointure. Les lignes sont filtrées en fonction des valeurs "51" et "91" de ces colonnes respectivement.

La jointure en boucle imbriquée (NESTED LOOPS) est une opération qui consiste à parcourir chaque ligne de la table DEPARTEMENT et à chercher chaque ligne correspondante dans la table VILLE.