

OPTIMISATION DES BASES DE DONNÉES



Encadré par :

- Dr.CHERRADI MOHAMED

Présenté par :

- ELKAMRI HAJAR
- GHARIBI HANAN

TABLE DE MATIÈRES

01

Introduction

02

Les index:

- index bitmap
- index de hachage
- index b-tree

03

Le partitionnement des tables

04

Le plan d'exécution logique

05

Le plan d'exécution physique

06

Les outils d'optimisation:

- Explain
- Autotrace
- Analyze
- Set Timing On

07

Conclusion



INTRODUCTION

LES INDEX

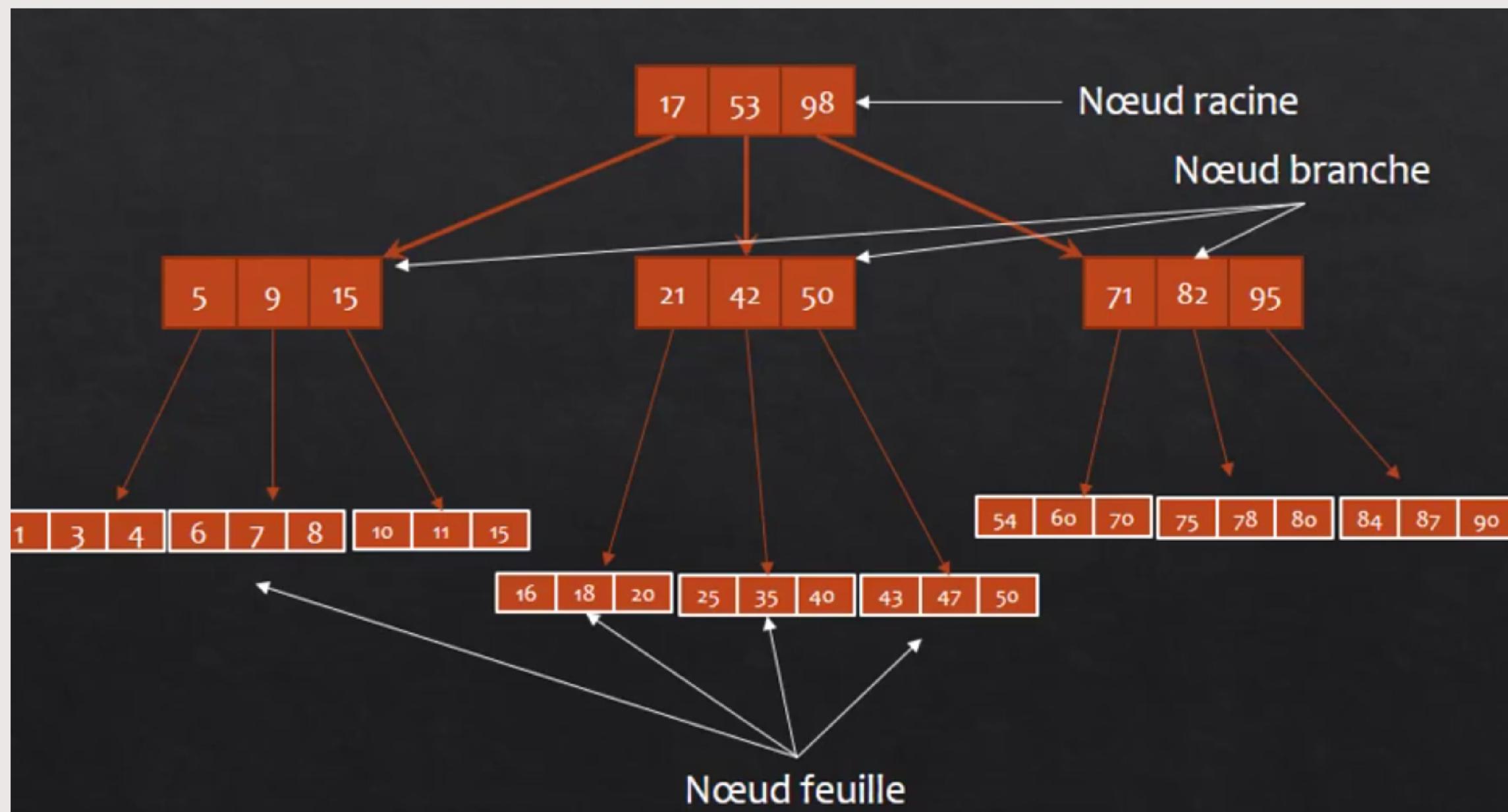
L'indexation dans une base de données est une technique qui permet de structurer les données pour accélérer la recherche et la récupération d'informations. En créant des index, le système de gestion de bases de données (SGBD) peut localiser rapidement les enregistrements sans avoir à parcourir l'ensemble des données.

Les types d'index :

-  **Les index B-Tree**
-  **Les index Bitmap**
-  **Les index de Hachage**

LES INDEX B-TREE

Structure d'un B-tree :



Nœuds : Un B-tree est constitué de nœuds triés par ordre croissant ou décroissant

- **Nœuds internes** : Ils contiennent des clés et des pointeurs vers d'autres nœuds (fils).
- **Nœuds feuilles** : Ils contiennent les clés et les pointeurs directs vers les données dans la base de données.

LES INDEX B-TREE

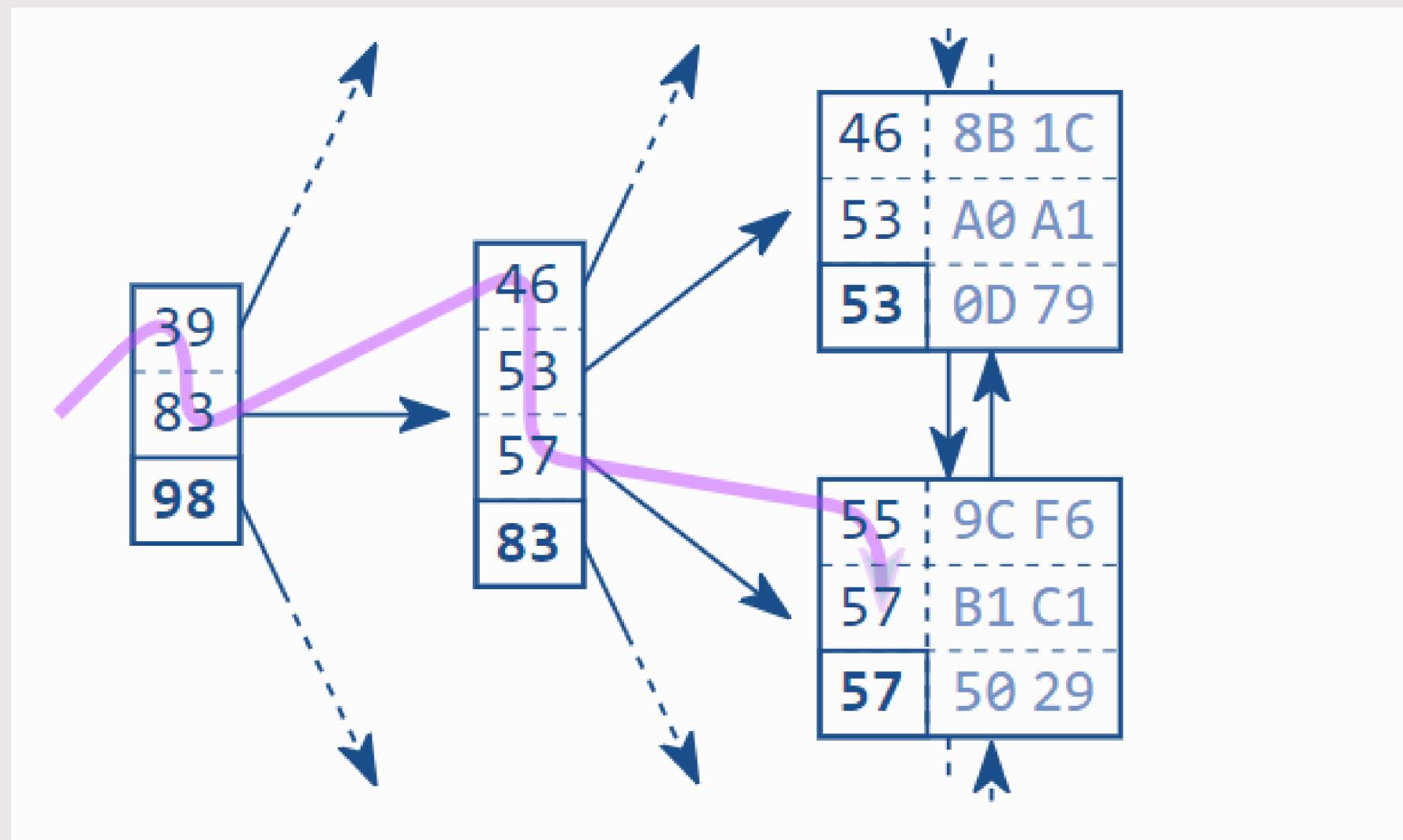
Fonctionnement d'un B-tree :

Recherche : Lorsqu'une requête est exécutée, l'algorithme de recherche par dichotomie est utilisé pour naviguer dans l'arbre .

- À chaque nœud visité, la recherche compare la clé recherchée avec les clés présentes dans le nœud.
- En fonction du résultat de la comparaison, elle suit le pointeur correspondant vers le nœud enfant approprié.
- Ce processus se répète à chaque niveau de l'arbre, affinant progressivement la recherche jusqu'à atteindre un nœud feuille où la clé recherchée est trouvée ou déterminée absente.

LES INDEX B-TREE

Exemple de fonctionnement :



Quel est l'enregistrement ayant pour clé la valeur 57 d'une autre façon comment le système va le trouver ?

LES INDEX B-TREE

Syntaxe d'utilisation :

➤ **Création d'un index B-tree :**

```
CREATE INDEX nom_index ON nom_table (colonne);
```

➤ **Forcer l'utilisation d'un index spécifique :**

```
SELECT * FROM nom_table USE INDEX (nom_index) WHERE colonne = valeur;
```

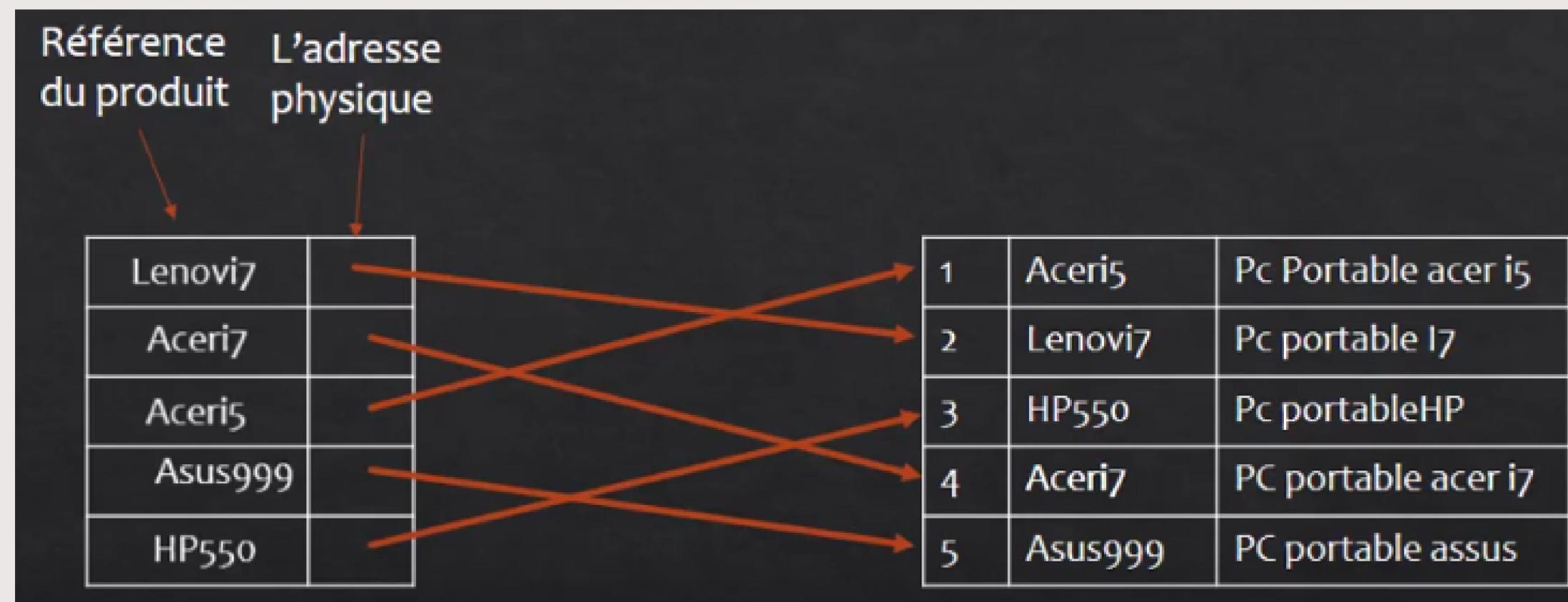
➤ **Suppression d'un index B-tree existant :**

```
DROP INDEX nom_index ON nom_table;
```

LES INDEX DE HACHAGE

La structure :

- Chaque entrée dans l'index de hachage est constituée de deux parties : la clé de recherche et un pointeur vers l'emplacement physique des données correspondantes.
- Une fonction de hachage est utilisée pour convertir la clé de recherche en une valeur de hachage, généralement une valeur numérique.
- Cette valeur de hachage est ensuite utilisée comme indice dans une table de hachage, où chaque case contient un pointeur vers l'emplacement des données correspondant à cette valeur de hachage



LES INDEX DE HACHAGE

Le fonctionnement :

- Lorsqu'une requête de recherche est effectuée, la fonction de hachage est appliquée à la clé de recherche pour obtenir sa valeur de hachage.
- En utilisant cette valeur de hachage comme indice, le système accède directement à la case correspondante dans la table de hachage.
- Si des collisions se produisent , une méthode de résolution de collision est utilisée pour gérer ces cas. Parmi les méthodes courantes, on trouve le hachage ouvert (où une autre fonction de hachage est utilisée pour trouver une nouvelle position pour l'élément en collision).

LES INDEX DE HACHAGE

Caractéristiques des index par Hachage :

- Accès Direct :

*Les index par hachage permettent un accès direct aux enregistrements. Une fois que la valeur de hachage est calculée, le SGBD peut accéder immédiatement à l'emplacement correspondant dans la table de hachage. Ceci est particulièrement efficace pour les recherches sur des valeurs uniques.

- Efficacité pour les Requêtes d'Égalité :

*Les index par hachage sont optimisés pour les requêtes utilisant des opérateurs d'égalité (=). Par exemple, **SELECT * FROM produits WHERE reference = 'ABC123';**

*Ils ne sont pas adaptés aux recherches par intervalle ou aux inégalités (<, >, BETWEEN) .

LES INDEX DE HACHAGE

Syntaxe d'utilisation :

➤ **Création d'un index de hachage :**

CREATE INDEX nom_index ON nom_table (colonne) USING HASH;

➤ **Suppression d'un index de hachage existant :**

DROP INDEX nom_index ON nom_table;

LES INDEX BITMAP

Structure :

- **Tableau de Bits :**
 - Chaque valeur distincte d'une colonne a un tableau de bits dédié.
 - La longueur de chaque tableau de bits est égale au nombre de lignes dans la table.
- **Représentation des Valeurs :**
 - Si une valeur spécifique est présente dans une ligne, le bit correspondant est réglé à 1, sinon à 0.
 - Pour une colonne avec m valeurs distinctes et n lignes, il y aura m tableaux de bits, chacun de longueur n .

LES INDEX BITMAP

Structure :

Collogues		
Name	Street	City
Zack	60 th St	Sana'a
Fawzan	Jamal St	Taiz
Mohammed	Mo'alla St	Aden
Moneer	Boriqa St	Aden
Ahmed	60th St	Sana'a
Ali	Bab Mosa St	Taiz
Faiq	Mo'alla St	Aden
Sarah	Berlin St	Sana'a

City		
Sana'a	Taiz	Aden
1	0	0
0	1	0
0	0	1
0	0	1
1	0	0
0	1	0
0	0	1
1	0	0

Street					
60 th St	Jamal St	Mo'alla St	Boriqa St	Bab Mosa St	Berlin St
1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	0	1

LES INDEX BITMAP

Le fonctionnement :

Recherche Simple :

Pour trouver toutes les lignes contenant une valeur spécifique, accédez directement au tableau de bits correspondant à cette valeur. Les positions des bits à 1 indiquent les lignes où cette valeur est présente.

Requêtes Complexes :

- AND (ET logique) :

Combinez deux tableaux de bits en utilisant l'opération ET.

Le résultat est un nouveau tableau de bits où les bits à 1 représentent les lignes où les deux conditions sont remplies.

- OR (OU logique) :

Combinez deux tableaux de bits en utilisant l'opération OU.

Le résultat est un nouveau tableau de bits où les bits à 1 représentent les lignes où au moins une des conditions est remplie.

LES INDEX BITMAP

Syntaxe d'utilisation :



Création d'un index :

```
CREATE BITMAP INDEX nom_index ON nom_table (colonne);
```



Suppression d'un index :

```
DROP INDEX nom_de_l_index;
```

LES INDEX

Avantages :

- **Réduction du temps de recherche :** Sans index, une base de données peut devoir effectuer une recherche séquentielle, c'est-à-dire parcourir chaque ligne pour trouver les données correspondantes. Les index permettent de localiser les données rapidement

- • **Amélioration des performances des requêtes:** Les index permettent d'accélérer considérablement les opérations de lecture en réduisant la quantité de données à parcourir pour trouver les enregistrements recherchés.

LES INDEX

Inconvénients :

 **Coût élevé de la maintenance des index :** Les opérations de mise à jour sur des tables munies d'index peuvent entraîner une baisse de performances, notamment lorsqu'elles impliquent un grand nombre de lignes ou des index complexes. La réorganisation des index peut demander d'importantes ressources système, ce qui risque de ralentir les opérations de mise à jour et d'avoir un impact négatif sur les performances globales du système.

LES INDEX



Remarque :

les valeurs nulles peuvent être incluses mais elles peuvent être traitées différemment selon le SGBD. Par exemple, dans certains cas, les lignes avec des valeurs nulles pour une colonne indexée peuvent ne pas être incluses dans l'index B-tree, tandis que dans d'autres cas, elles peuvent être incluses mais avec un coût de performance potentiel.



Solution :

Il peut être bénéfique de créer un index sur une expression qui utilise NVL pour remplacer les valeurs NULL. Cela peut améliorer les performances des requêtes qui utilisent cette expression.

```
CREATE INDEX index_nvl ON ma_table(NVL(colonne_indexée, valeur_par_défaut));
```

LES INDEX

Démonstration de la consommation d'espace :

Supposons que nous avons une table avec 100,000 enregistrements et une colonne avec les caractéristiques suivantes :

B-tree:

Si chaque nœud peut contenir 100 clés, nous aurions environ $100000/100 = 1000$ noeuds

Espace : $1000 \text{ noeuds} \times 4 \text{ KB par noeud} = 4 \text{ MB}$.

Bitmap:

Pour une colonne avec 100 valeurs distinctes : 100 bits par enregistrement.

Espace : $100 \times 100,000 = 10,000,000 \text{ bits} = 1.25 \text{ MB}$

LES INDEX

Hashage:

Supposons une table de hachage de taille $t=100,000$ pour une bonne répartition

Approximativement 4 bytes par entrée pour la table principale + espace pour collisions.

Espace : $100,000 \times 4$ bytes = 400 KB + espace de collisions.

Conclusion

- Bitmap est généralement le plus compact pour les colonnes à faible cardinalité.
- B-tree consomme plus d'espace en raison de la structure d'arbre et des pointeurs.
- Hachage peut être très efficace en espace pour des insertions réparties uniformément, mais moins prévisible avec des collisions.

COMMENT CHOISIR LE TYPE D'INDEX CONVENABLE

- Utilisez un index B-tree si vous avez besoin d'effectuer des recherches par plage ou si vos colonnes ont des valeurs très distinctes. C'est un bon choix par défaut pour la plupart des SGBD transactionnels.
- Utilisez un index bitmap si vous travaillez principalement avec des données de faible cardinalité.
- Utilisez un index basé sur le hachage si vous avez des recherches fréquentes par clé exacte et que vos requêtes ne nécessitent pas de recherches par plage.

LE PARTITIONNEMENT

Principe :

Le partitionnement des tables est une technique d'optimisation de la base de données qui divise une grande table en segments plus petits et plus gérables appelés partitions. Chaque partition est stockée et gérée indépendamment, ce qui peut améliorer les performances des requêtes, faciliter la gestion des données et optimiser l'utilisation des ressources.

Types de Partitionnement :

1. Partitionnement par plage :

Les données sont divisées en partitions basées sur une plage de valeurs d'une colonne spécifique, comme des dates ou des nombres.

2 .Partitionnement par liste:

Les données sont divisées en partitions basées sur des valeurs spécifiques d'une colonne.

LE PARTITIONNEMENT

EXEMPLE SUR LE PARTITIONNEMENT PAR PLAGUE :

```
CREATE TABLE ventes (
    id INT,
    date_vente DATE,
    montant DECIMAL
)
PARTITION BY RANGE (YEAR(date_vente)) (
    PARTITION p2021 VALUES LESS THAN (2022),
    PARTITION p2022 VALUES LESS THAN (2023)
);
```

LE PARTITIONNEMENT

EXEMPLE SUR LE PARTITIONNEMENT PAR LISTE :

```
CREATE TABLE ventes (
    id INT,
    region VARCHAR(50),
    montant DECIMAL
)
PARTITION BY LIST (region) (
    PARTITION p_nord VALUES ('Nord'),
    PARTITION p_sud VALUES ('Sud')
);
```

LE PARTITIONNEMENT

index Locaux :

Lorsque vous créez un index local sur une table partitionnée, chaque partition de la table aura son propre sous-ensemble d'index. Cela signifie que l'index couvre uniquement les données de la partition à laquelle il est associé. Les index locaux sont généralement préférés dans les environnements partitionnés car ils ne concernent qu'une ou quelques partitions spécifiques.

Exemple d'utilisation :

```
CREATE INDEX idx_ventes_local ON ventes(client_id) LOCAL (
    PARTITION p2019,
    PARTITION p2020,
    PARTITION p2021,
    PARTITION p2022
);
```

LE PARTITIONNEMENT

index Globaux :

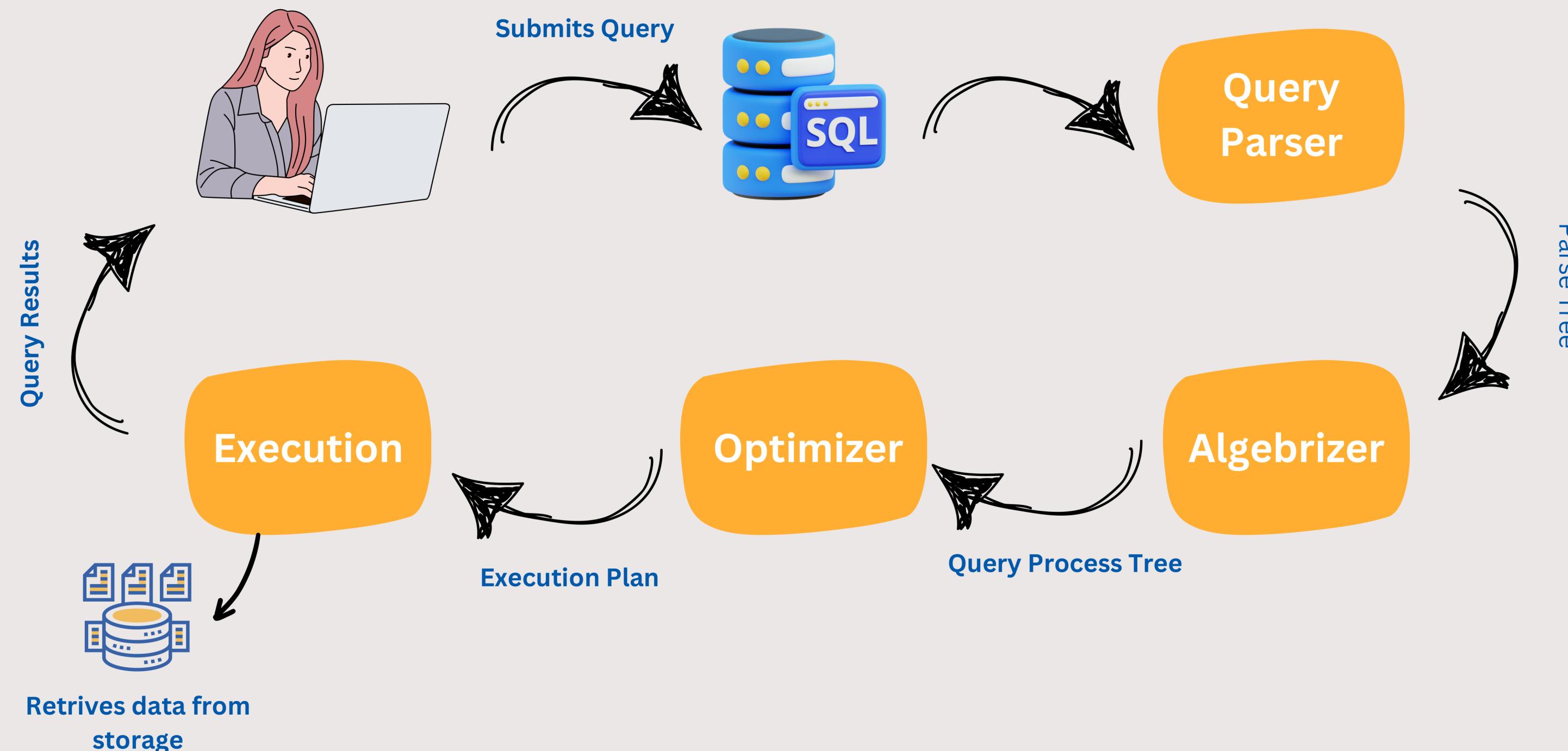
un index global sur une table partitionnée couvre toutes les partitions de la table. Cela signifie que l'index est unique et couvre l'ensemble des données de la table. Les index globaux peuvent être moins efficaces dans un environnement partitionné, car ils peuvent entraîner des opérations coûteuses lors de la maintenance de l'index, en particulier lors de l'ajout ou de la suppression de partitions.

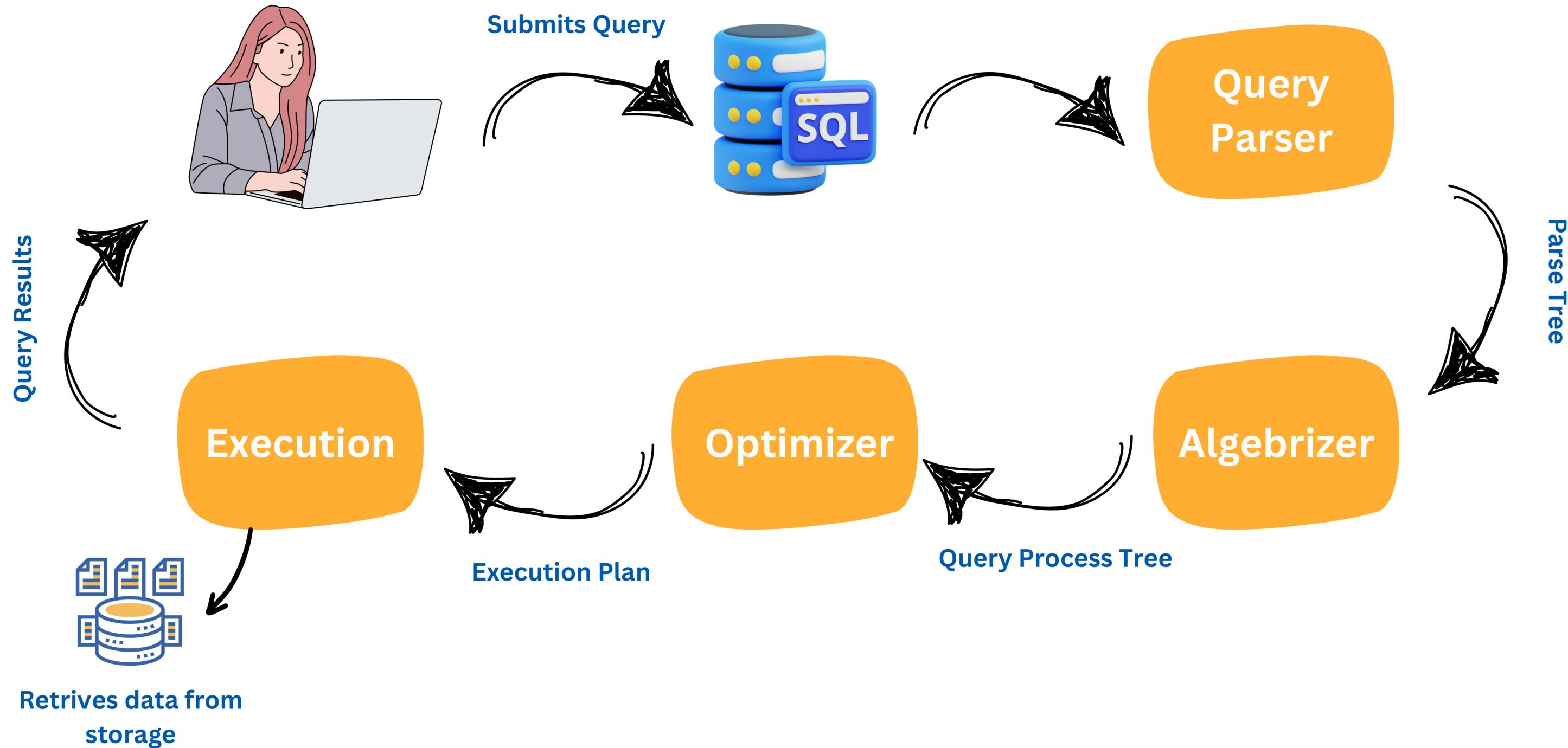
Exemple d'utilisation :

```
CREATE INDEX idx_ventes_global ON ventes(client_id) GLOBAL;
```

PLAN D'EXECUTION LOGIQUE

Traitement des requêtes :





PLAN D'EXECUTION LOGIQUE

Traduction de la requête SQL en algèbre relationnelle :

π

LA PROJECTION

σ

LA SÉLECTION

∞

LA JOINTURE

\cap

L'INTERSECTION



LA DÉFÉRENCE

U

L'UNION

X

LE PRODUIT CARTÉSIEN

PLAN D'EXECUTION LOGIQUE

Exemple : Requête SQL

```
SELECT NOM, BUREAU  
FROM EMPLOYE, DEPARTEMENT  
WHERE EMPLOYE.NUMOD = DEPARTEMENT.NUMOD AND  
EMPLOYE.SALAIRE > 1000
```

Traduction

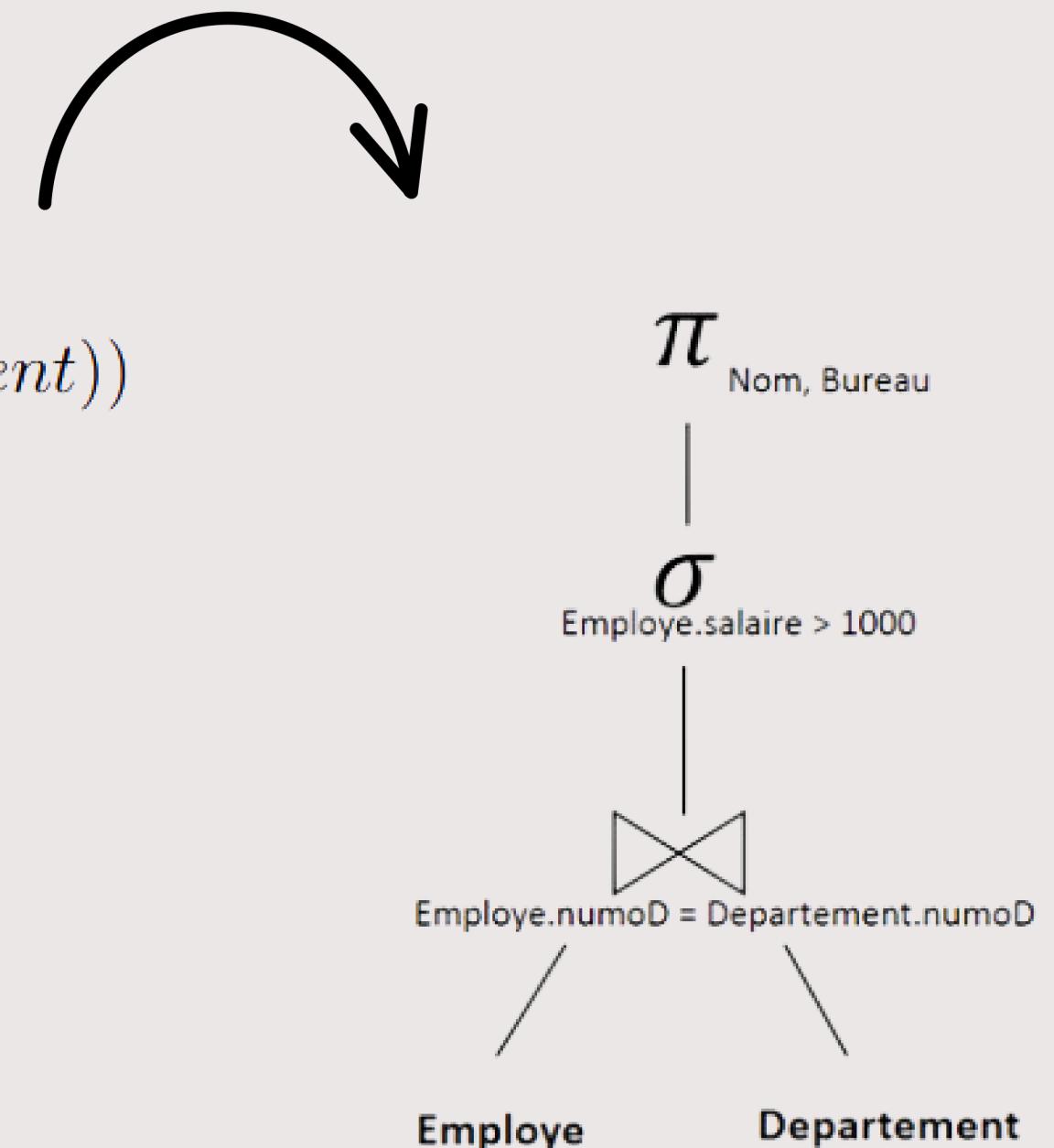
$$\pi_{Nom,Bureau}(\sigma_{E.salaire>1000}(Employe \bowtie_{E.numD=D.numD} Departement))$$

PLAN D'EXECUTION LOGIQUE

Tracer l'arbre algébrique:

$\pi_{Nom, Bureau}(\sigma_{E.salaire > 1000}(Employe \bowtie_{E.numD=D.numD} Departement))$

- **Feuilles :** tables utilisées dans la requête
- **Nœuds intermédiaires :** opérations algébriques
- **Nœud racine :** dernière opération algébrique avant le retour du résultat



PLAN D'EXECUTION LOGIQUE

Optimisation de l'arbre algébrique :

Le principe général de l'optimisation repose sur le constat suivant :

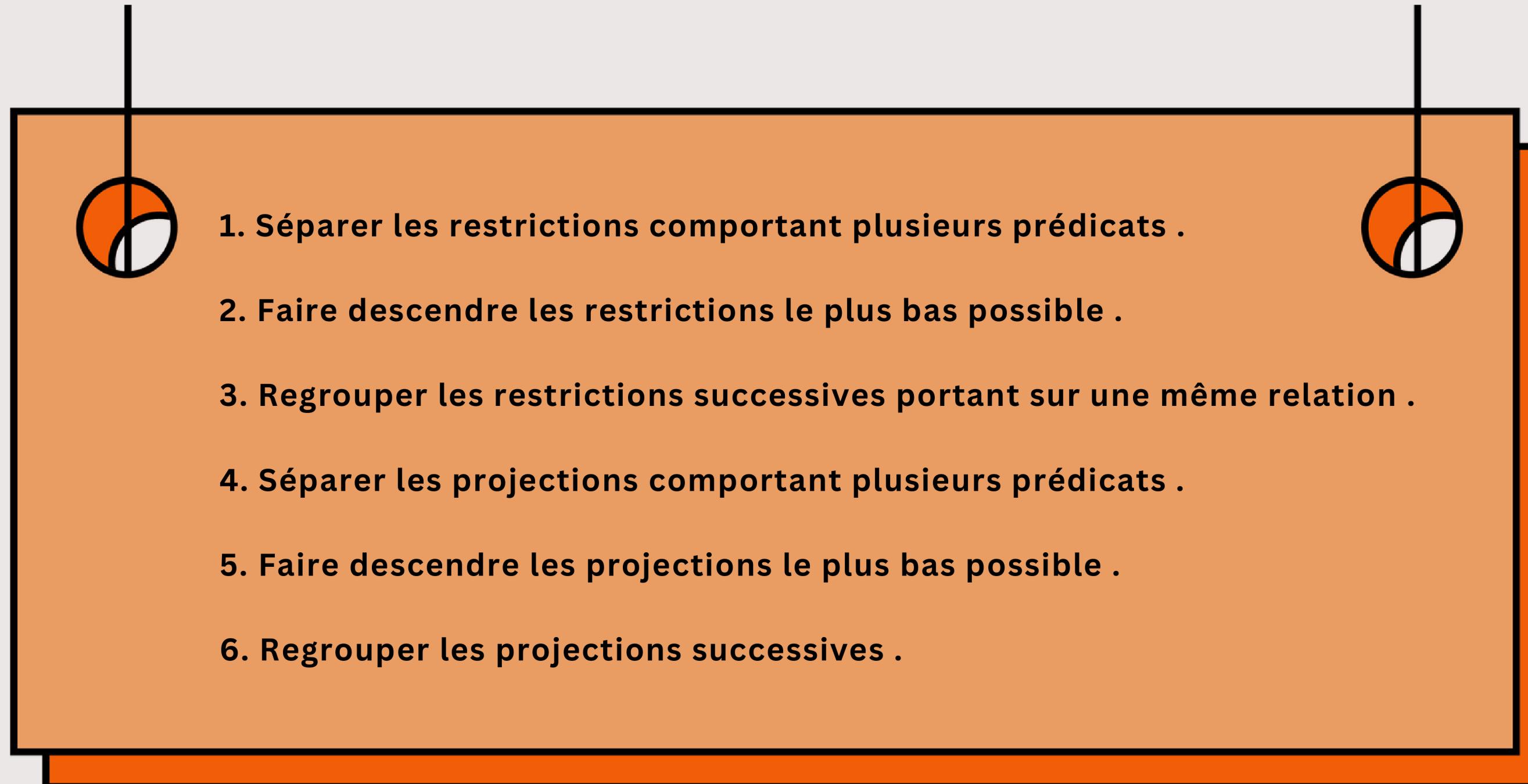
- Les opérations unaires produisent des tables plus petites que la table d'origine.
- Les opérations binaires produisent des tables plus grandes que la table d'origine.



Supprimer un maximum de lignes et de colonnes avant de faire les jointures et faire **les jointures** avant **les produits cartésiens**.

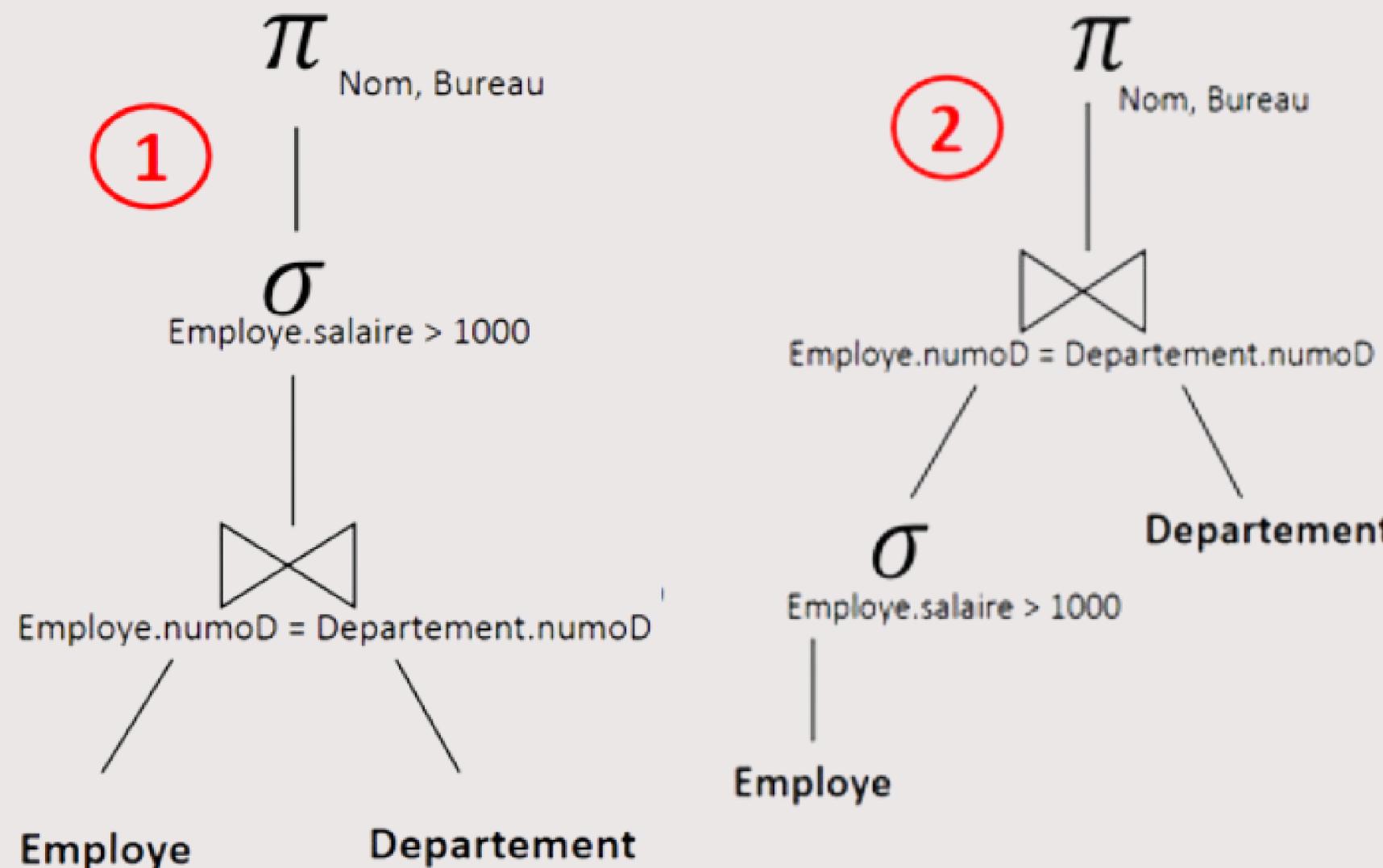
PLAN D'EXECUTION LOGIQUE

Règles de transformation :



PLAN D'EXECUTION LOGIQUE

Exemple : Requête SQL

$$\pi_{Nom, Bureau}(\sigma_{E.salaire > 1000}(Employe \bowtie_{E.numD=D.numD} Departement))$$


```

SELECT NOM, BUREAU
FROM (
    SELECT *
    FROM EMPLOYE
    WHERE SALAIRE > 1000
) EMPLOYEFILTRE
JOIN DEPARTEMENT ON EMPLOYEFILTRE.NUMOD
DEPARTEMENT.NUMOD;
    
```

PLAN D'EXECUTION LOGIQUE

Calculer les coûts des plans :

On s'intéresse au coût des opérateurs \bowtie et σ en terme d'Entrées/Sortie :

Algorithme :

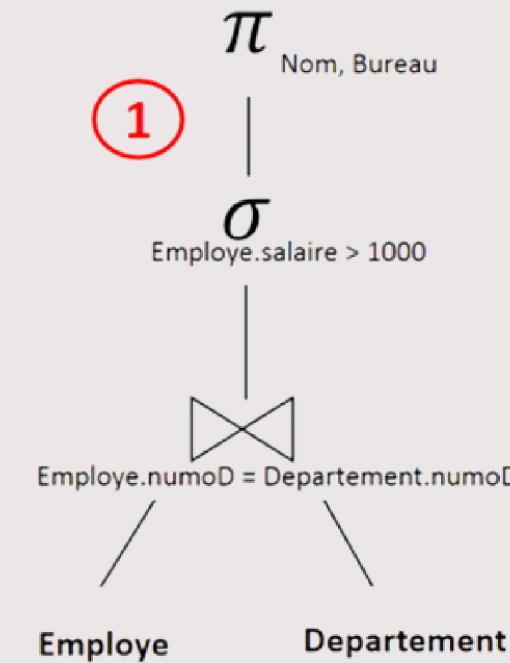
- Sélection :
 - Coût total (E/S) = $T(R)$ + $s \times T(R)$
 - Où $T(R)$ est le nombre de tuples dans **R** et s est la sélectivité de la condition.
- Jointure :
 - Coût total (E/S) = $T(R1)$ + $T(R1) \times T(R2)$ + $j \times (T(R1) \times T(R2))$
 - Où $T(R1)$ et $T(R2)$ sont les nombres de tuples dans **R1** et **R2** respectivement, et j est la sélectivité de la condition de jointure.

PLAN D'EXECUTION LOGIQUE

Calculer les coûts des plans :

Hypothèses :

1. Il y a 300 lignes dans "Employe"
2. Il y a 12 lignes dans "Departement"
3. On suppose qu'il n'y a que 20% d'employés ayant un salaire > 1000



Coût Total = 4560 E/S

Coût plan initial

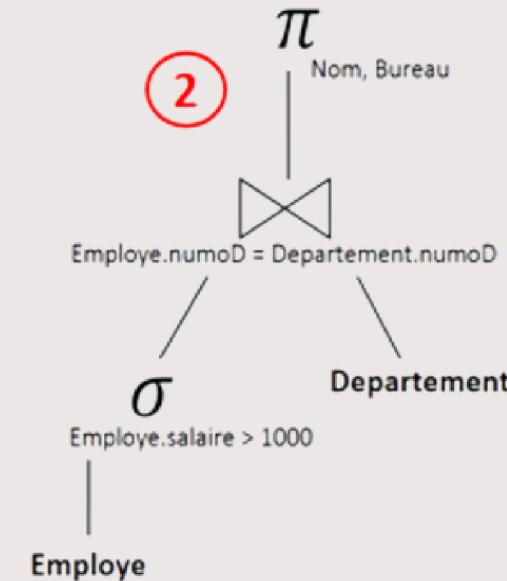
- Coût de la jointure :
 - coût (Entrée) = $300 + (300 * 12) = 3900$ E
 - coût (Sortie) = 300 S (pour chaque employé, un seul Département)
 - Total = $3900 + 300 = 4200$ E/S
- Coût de la sélection :
 - coût (Entrée) = 300 E
 - coût (Sortie) = $300 * 0.2 = 60$ S
 - Total = $300 + 60 = 360$ E/S

PLAN D'EXECUTION LOGIQUE

Calculer les coûts des plans :

Hypothèses :

1. Il y a 300 lignes dans "Employe"
2. Il y a 12 lignes dans "Departement"
3. On suppose qu'il n'y a que 20% d'employés ayant un salaire > 1000

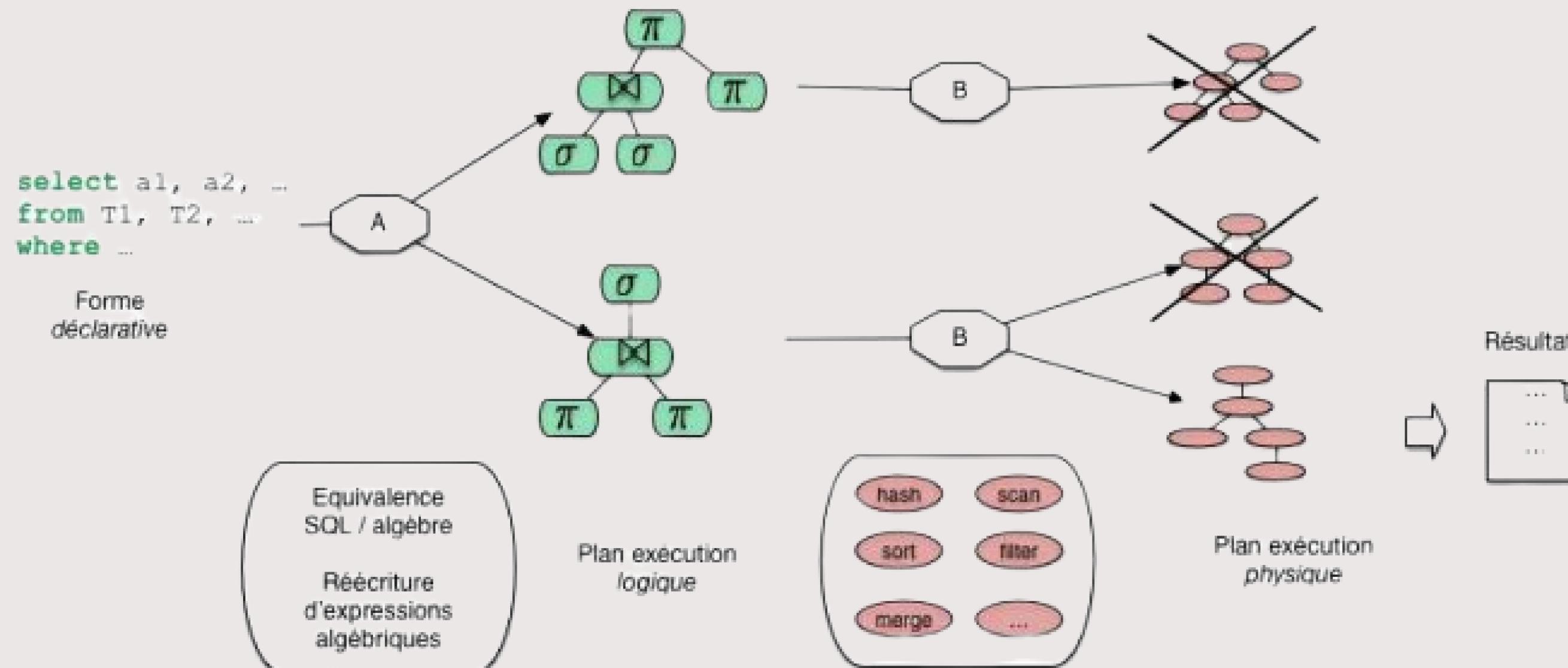


Coût Total = 1200 E/S

Coût plan optimisé

- Coût de la jointure :
 - coût (Entrée) = $60 + (60 * 12) = 780$ E
 - coût (Sortie) = 60 S (pour chaque employé, un seul Département)
 - Total = $780 + 60 = 840$ E/S
- Coût de la sélection :
 - coût (Entrée) = 300 E
 - coût (Sortie) = $300 * 0.2 = 60$ S
 - Total = $300 + 60 = 360$ E/S

PLAN D'EXECUTION PHYSIQUE



PLAN D'EXECUTION PHYSIQUE

- ▶ Les plans d'exécution montrent les étapes détaillées nécessaires pour exécuter une instruction SQL.
- ▶ Ces étapes sont exprimées comme un ensemble d'opérateurs de base de données qui consomment ou produisent des lignes ou les deux.
- ▶ L'ordre des opérateurs et leur mise en œuvre sont décidés par l'optimiseur.

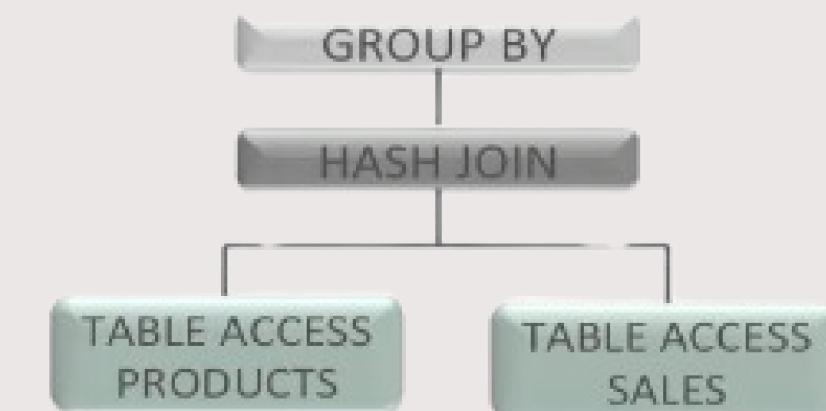
Query:

```
SELECT prod_category, avg(amount_sold)
FROM   sales s, products p
WHERE p.prod_id = s.prod_id
GROUP BY prod_category;
```

Tabular representation of plan

Id	Operation	Name
0	SELECT STATEMENT	
1	HASH GROUP BY	
*	HASH JOIN	
2	TABLE ACCESS FULL	PRODUCTS
3	TABLE ACCESS FULL	SALES
4		

Tree-shaped representation of plan



PLAN D'EXECUTION PHYSIQUE

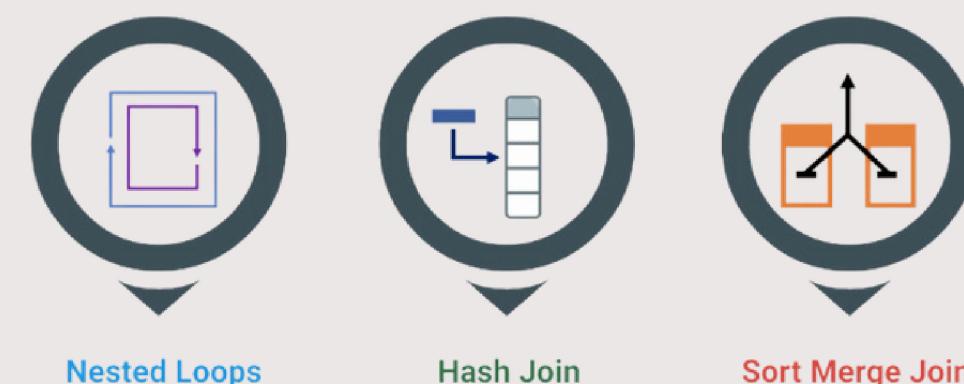
Opérateurs dans un Plan d'Exécution Physique:

- **Définition:** Les opérateurs sont les composants de base d'un plan d'exécution physique. Ils représentent les actions prises pour accéder et manipuler les données.
- **Rôle:** Chaque opérateur effectue une tâche spécifique, comme la lecture de données depuis une table, le filtrage des lignes, le tri des résultats, etc.

Accès aux Données



Jointures de tables



Traitement des données

SORT	GROUP BY
FILTER	AGGREGATE
UNION	UNION ALL
DISTINCT	LIMIT

PLAN D'EXECUTION PHYSIQUE

Opérateurs dans un Plan d'Exécution Physique:

Accès aux Données

Les opérateurs d'accès dans un plan d'exécution physique peuvent être classés en deux catégories principales : accès par table et accès par index.

- **Accès par Table :**

- **FULL TABLE SCAN :** Lit toutes les lignes de la table et filtre celles qui ne répondent pas aux critères de sélection.

- **Accès par Index :**

- **TABLE ACCESS BY ROWID :** Utilise le ROWID pour accéder directement aux lignes spécifiques.
- **INDEX UNIQUE SCAN :** Accède à une seule ligne en utilisant une contrainte UNIQUE ou de clé primaire.

PLAN D'EXECUTION PHYSIQUE

Opérateurs dans un Plan d'Exécution Physique:

Accès aux Données

- **INDEX RANGE SCAN** : Parcourt les entrées de l'index qui correspondent à la condition de plage spécifiée (comme WHERE column > value), en extrayant les lignes de la table correspondantes.
- **INDEX FULL SCAN** : Lit l'intégralité de l'index, récupère ensuite les ROWIDs associés à chaque entrée d'index.
- **INDEX FAST FULL SCAN** : Parcourt tous les blocs de l'index pour remplacer un scan de table complet.
- **INDEX SKIP SCAN** : Saute la colonne principale de l'index si elle n'est pas parmi les colonnes de sélection autrement dit si elle n'est pas sélective.
- **BITMAP INDEX** : Parcourt le bitmap en récupérant les enregistrements associés aux bits ayant 1.

PLAN D'EXECUTION PHYSIQUE

Opérateurs dans un Plan d'Exécution Physique:

Jointures de tables

- **NESTED LOOP JOIN** : Combinaison de deux tables en utilisant une boucle imbriquée.
- **HASH JOIN** : Utilisation de tables de hachage pour combiner rapidement de grandes tables.
- **SORT MERGE JOIN** : Trie les tables avant de les fusionner pour effectuer la jointure.

Join	Complexity	Work	Total
Nested loops	$O(\#T1 * \#T2)$	Comparisons $52 * 52$	2,704
Merge join	$O(\#T1 \log \#T1 + \#T2 \log \#T2)$	Sorting+ Comparisons $52 \log 52 + 52 \log 52 \sim 180$ ~ 150	~ 300
Hash join	$O(\#T1 + \#T2)$	Build hash + Probe hash $52 + 52$	104 WINNER

PLAN D'EXECUTION PHYSIQUE

Opérateurs dans un Plan d'Exécution Physique:

Traitement des données

- **FILTER** : Applique des conditions pour filtrer les lignes en fonction des critères de sélection.
- **SORT** : Trie les lignes en fonction des colonnes spécifiées.
- **AGGREGATE** : Effectue des opérations d'agrégation telles que SUM, COUNT, AVG.
- **UNION** : Combine les résultats de plusieurs requêtes en supprimant les doublons.
- **UNION ALL** : Combine les résultats de plusieurs requêtes sans supprimer les doublons.
- **GROUP BY** : Regroupe les lignes en ensembles basés sur les valeurs d'une ou plusieurs colonnes.

OUTILS D'OPTIMISATION D'ORACLE

L'outil EXPLAIN:

UTILITÉ:

Permet de générer le plan d'exécution physique d'une requête sans l'exécuter. Le plan est inséré dans la table PLAN_TABLE.

SYNTAXE:

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'identifier'
  INTO plan_table_name
FOR sql_statement;
```

Id Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0 SELECT STATEMENT				6 (100)	
* 1 HASH JOIN		3	192	6 (0)	00:00:01
* 2 HASH JOIN		3	147	4 (0)	00:00:01
* 3 TABLE ACCESS FULL DEPARTMENTS		2	42	2 (0)	00:00:01
* 4 TABLE ACCESS FULL EMPLOYEES		10	280	2 (0)	00:00:01
5 TABLE ACCESS FULL ROLES		20	300	2 (0)	00:00:01

OUTILS D'OPTIMISATION D'ORACLE

L'outil EXPLAIN:

DBMS_XPLAN:

DBMS_XPLAN est un package **PL/SQL** fourni par Oracle. Ce package contient des fonctions qui permettent de formater et d'afficher les plans d'exécution des requêtes SQL.

LES FONCTIONS DE DBMS_XPLAN:

- **DISPLAY** : Affiche le plan d'exécution stocké dans une table de plan.
- **DISPLAY_CURSOR** : Affiche le plan d'exécution pour une requête en cours d'exécution, basée sur le curseur SQL.

```
DBMS_XPLAN.DISPLAY(  
    table_name => 'PLAN_TABLE',  
    statement_id => NULL,  
    format => 'TYPICAL');
```

```
DBMS_XPLAN.DISPLAY_CURSOR(  
    cursor => cursor_variable,  
    format => 'TYPICAL');
```

OUTILS D'OPTIMISATION D'ORACLE

L'outil EXPLAIN:

EXEMPLE:

```
EXPLAIN PLAN FOR  
SELECT ename, dname  
FROM emp  
NATURAL JOIN dept;
```

```
SELECT *  
FROM TABLE(DBMS_XPLAN.DISPLAY);
```

```
Plan hash value: 3625962692  
  
-----  
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |  
-----  
| 0  | SELECT STATEMENT   |        |       |       |    3  (0)  | 00:00:01 |  
| 1  | NESTED LOOPS       |        |       |       |    3  (0)  | 00:00:01 |  
| 2  | NESTED LOOPS       |        |       |       |    3  (0)  | 00:00:01 |  
| 3  | TABLE ACCESS FULL  | EMP    |       |       |    3  (0)  | 00:00:01 |  
| 4  | TABLE ACCESS BY INDEX ROWID | DEPT |       |       |    1  (0)  | 00:00:01 |  
| 5  | INDEX UNIQUE SCAN  | PK_DEPT|       |       |    0  (0)  | 00:00:01 |  
  
-----  
Predicate Information (identified by operation id):  
-----  
 4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

OUTILS D'OPTIMISATION D'ORACLE

L'outil AUTOTRACE:

UTILITÉ:

La commande **AUTOTRACE** est utilisée pour afficher automatiquement le plan d'exécution d'une instruction SQL et éventuellement, les statistiques d'exécution.

Contrairement à **EXPLAIN PLAN**, AUTOTRACE exécute réellement la requête.

SYNTAXE:

```
SET AUTOTRACE ON
```

```
SET AUTOTRACE TRACEONLY EXPLAIN
```

```
SET AUTOTRACE TRACEONLY STATISTICS
```

```
SET AUTOTRACE TRACEONLY
```

```
SET AUTOTRACE OFF
```

OUTILS D'OPTIMISATION D'ORACLE

L'outil AUTOTRACE:

EXEMPLE:

```
SET AUTOTRACE TRACEONLY  
SELECT ename, dname FROM emp NATURAL JOIN dept;
```

```
Statistics  
-----  
      1 recursive calls  
      0 db block gets  
     10 consistent gets  
      4 physical reads  
      0 redo size  
  947 bytes sent via SQL*Net to client  
 380 bytes received via SQL*Net from client  
      2 SQL*Net roundtrips to/from client  
      0 sorts (memory)  
      0 sorts (disk)  
   15 rows processed
```

OUTILS D'OPTIMISATION D'ORACLE

L'outil ANALYZE:

UTILITÉ:

Collecte de Statistiques : ANALYZE permet de collecter des statistiques qui aident l'optimiseur à choisir les chemins d'accès les plus efficaces pour exécuter des requêtes.

→ Il est recommandé d'utiliser l'outil **ANALYZE** lorsque des modifications significatifs sur les tables, index ou colonnes sont faites.

SYNTAXE:

```
ANALYZE TABLE nom_table COMPUTE STATISTICS;
```

L'output de la commande **ANALYZE** n'est pas directement affiché à l'utilisateur, mais les statistiques collectées sont stockées dans les tables du dictionnaire de données.

- DBA/ALL/USER_TABLES
- DBA/ALL/USER_index
- DBA/ALL/USER_TAB_COLUMNS

OUTILS D'OPTIMISATION D'ORACLE

L'outil ANALYZE:

EXEMPLE :

```
ANALYZE TABLE employees COMPUTE STATISTICS;
```

```
SELECT table_name, num_rows, blocks, avg_row_len  
FROM dba_tables  
WHERE table_name = 'EMPLOYEES';
```

TABLE_NAME	NUM_ROWS	BLOCKS	AVG_ROW_LEN
EMPLOYEES	1000	50	120

- **NUM_ROWS** : Le nombre de lignes dans la table ou l'index.
- **BLOCKS** : Le nombre de blocs de base de données utilisés par la table.
- **AVG_ROW_LEN** : La longueur moyenne des lignes dans la table.
- **DISTINCT_KEYS** : Le nombre de valeurs distinctes dans l'index.
- **LEAF_BLOCKS** : Le nombre de blocs feuille dans l'index.

OUTILS D'OPTIMISATION D'ORACLE

L'outil SET TIMING On:

UTILITÉ:

Lorsqu'elle est activée, **SQLPlus** affiche le temps écoulé pour une commande SQL exécutée, .

SYNTAXE:

```
SET TIMING ON;  
SELECT * FROM ma_table;
```

EXEMPLE:

```
Elapsed: 00:00:00.05
```

```
SET TIMING OFF;
```



CONCLUSION