# Module 4 - Optimisation

Cours IGE487 – Modélisation de bases de données

Chargé de cours Tudor Antohi

# MISE EN CONTEXTE

# Objectifs de module

- Mise en contexte
- Traitement de SQLs
- Algorithmes d’optimisation SQL et couts logiques associés
- Permutation heuristiques
- Couts physiques
- Statistiques et plans d’exécutions
- Exemples

# Optimisation

- **Systèmes informatiques** (théorie de temps d'attente (*wait analysis)*, planification optimale de capacité, prédictions)
- **Code applicatif** (plus complexe, implique beaucoup de manualité, l'optimisation est encore modeste)

- **Code SQL**
  - Comment le programmeur eut écrire du meilleur code ?
  - **Comment l'optimiseur travaille ?**

# Mot introductif – Trivia

Google Maps, Waze, SQL – optimisations avec des algorithmes trouvés dans le cours connu.

**SQL est déclaratif.**

→ L'utilisateur indique au SGBD la réponse qu'il souhaite, et non pas comment obtenir la réponse.

→ Il peut y avoir une grande différence de performance en fonction de plan utilisé.

# Histoire

Première implantation d’optimiseur – System R 1970 – IBM

→ **Les gens soutenaient que le SGBD ne pourrait jamais choisir un plan de requête meilleur que ce qu'un humain pourrait écrire.**

Nombreux concepts et décisions de conception de l'optimiseur de System R sont encore utilisés aujourd'hui.

**→ Après 2000 Larry Ellison d’Oracle embauche les mathématiciens de MIT pour écrire le SQL Tuning Pack et optimiser Oracle dans une vision basée sur l’apprentissage par renforcement vers une technologie qui rend la base de données autonome en terme de gestion de performance.**

# Histoire

C'est la partie la plus difficile de la construction d'un SGBD.

Si vous êtes bon dans ce domaine, vous serez payé $$$ a l’heure.

Les gens commencent à envisager l'emploi de ML pour améliorer la précision et l'efficacité des optimiseurs.

→ IBM DB2 a essayé cela avec LEO au début des années 2000...

→ **SQL Tuning Pack d’Oracle fait une excellente job.**

→ **Database Advisor de SQL Server**

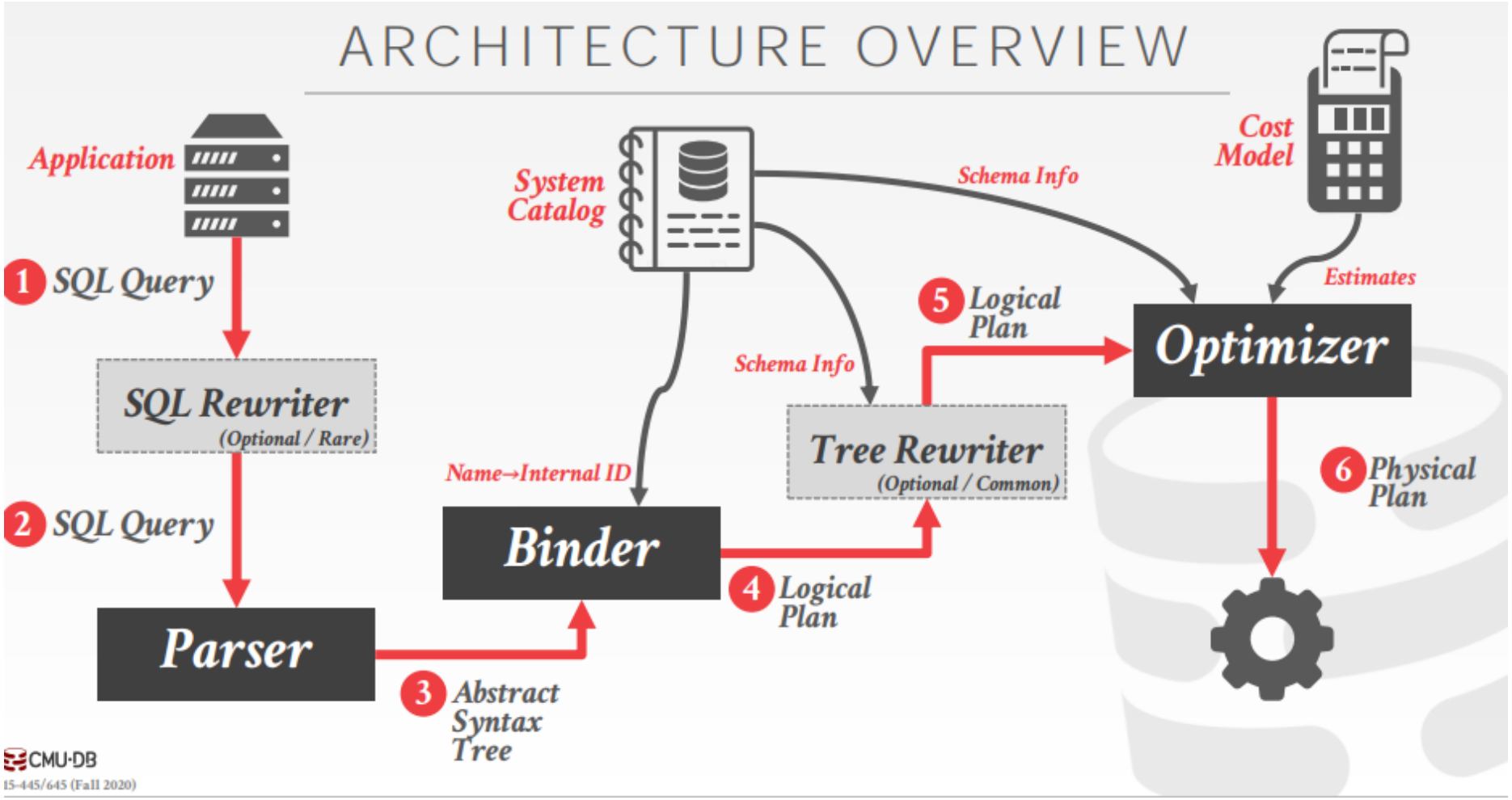
→ **Unravel veut optimizer plusieurs technologies**

# TRAITEMENT DE SQL

# Traitement de SQL

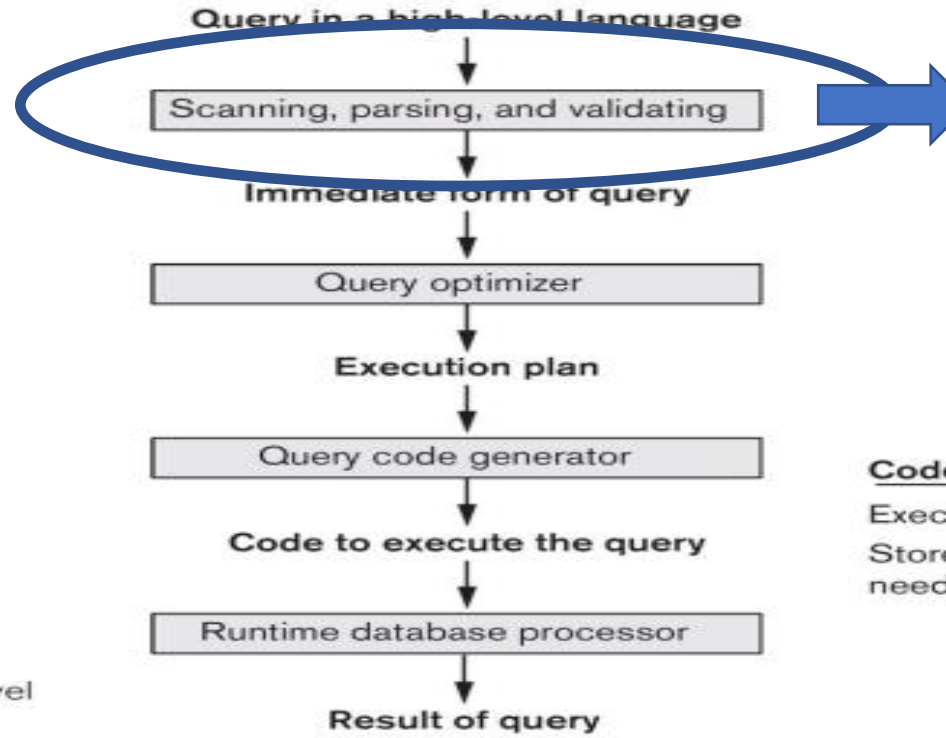
1. Analyse lexicale (LEX) pour identifier les jetons SQL,
2. Analyse syntactique (YACC, BNF) et validation (les noms d'attributs et de relations doivent existent dans le schéma BD)
3. Les variables de liaison (*binder*) sont associées
4. L'arbre de requête correspondant est créé
5. Une stratégie d'exécution optimale est choisie, pour récupérer le résultat de la requête.

# Vue d’architecture



# Traitement

## Introduction to Query Processing (2)




Ajoutez BINDING

**Code can be:**

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

**Figure 19.1**
Typical steps when processing a high-level query.

# Optimisation

Parfois, "optimal" doit être interprété comme "raisonnablement efficace", parce que trouver la stratégie optimale peut prendre trop de temps

Il existe deux techniques principales pour mettre en œuvre l'optimisation des requêtes :

- **optimisation heuristique** des requêtes capable de réorganiser les opérations RA
- **optimisation des requêtes** basée sur les coûts estimer le coût des différents stratégies d'exécution

# Optimisation

**Heuristique / Règles**

→ Réécrire la requête pour supprimer les choses inefficaces.

→ Ces techniques peuvent avoir besoin d'examiner le catalogue, mais elles n'ont pas besoin d'examiner les données.

**Recherche basée sur les coûts**

→ Utiliser un modèle pour estimer le coût d'exécution d'un plan.

→ Évaluer plusieurs plans équivalents pour une requête et choisir celui qui présente le coût le plus faible.

# Plan logique et Plan physique

L'optimiseur génère un mappage d'une **expression d'algèbre logique** à **l'expression algébrique physique** équivalente optimale.

Les opérateurs physiques définissent une stratégie d'exécution spécifique utilisant le chemin d'accès.

→ Ils peuvent dépendre du format physique des données qu'ils elles traitent (c'est-à-dire le tri, la compression).

→ Il n'y a pas toujours de correspondance 1:1 entre le logique et le physique.

# Optimiseur

**Optimisation  de requêtes** = le processus de choix d'une stratégie d'exécution appropriée d'exécution pour traiter une requête.

Deux types de représentations internes :

- Arbre d’expression
- Graph de requêtes

# Arbres et Graphs

**Arbre de requête :** Une structure de données arborescente qui correspond à **une expression d'algèbre relationnelle**. Elle représente les relations d'entrée de la requête comme des nœuds feuilles de l'arbre, et représente les opérations d'algèbre relationnelle comme des nœuds internes.

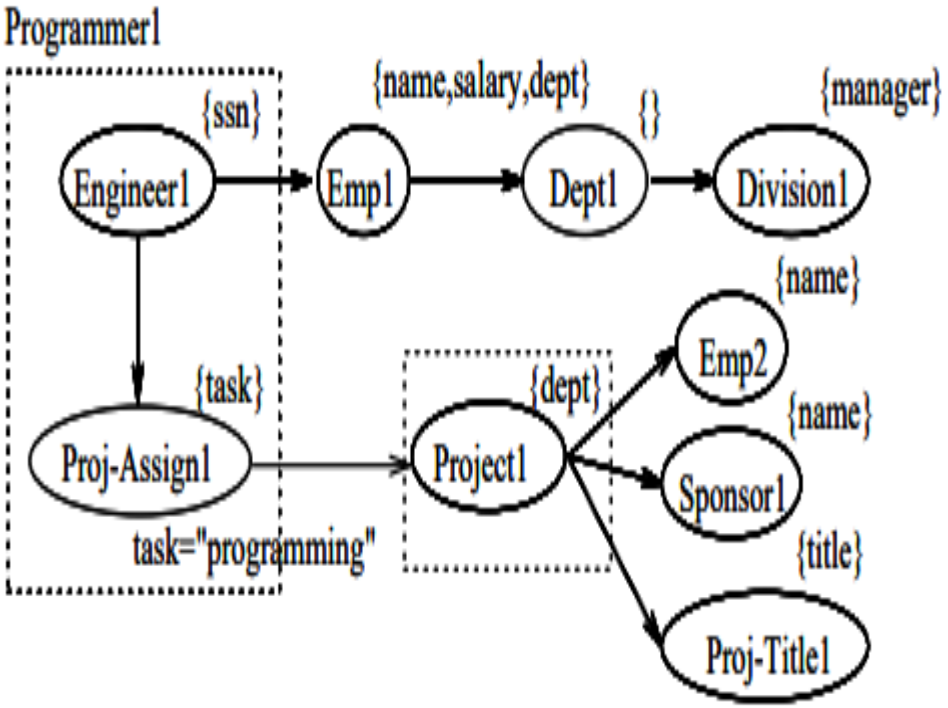
Une exécution de l'arbre de requête consiste

1. à exécuter une opération sur le nœud interne chaque fois que les opérandes sont disponibles,
2. puis à remplacer ce nœud interne par la relation qui résulte de l'exécution de l'opération.

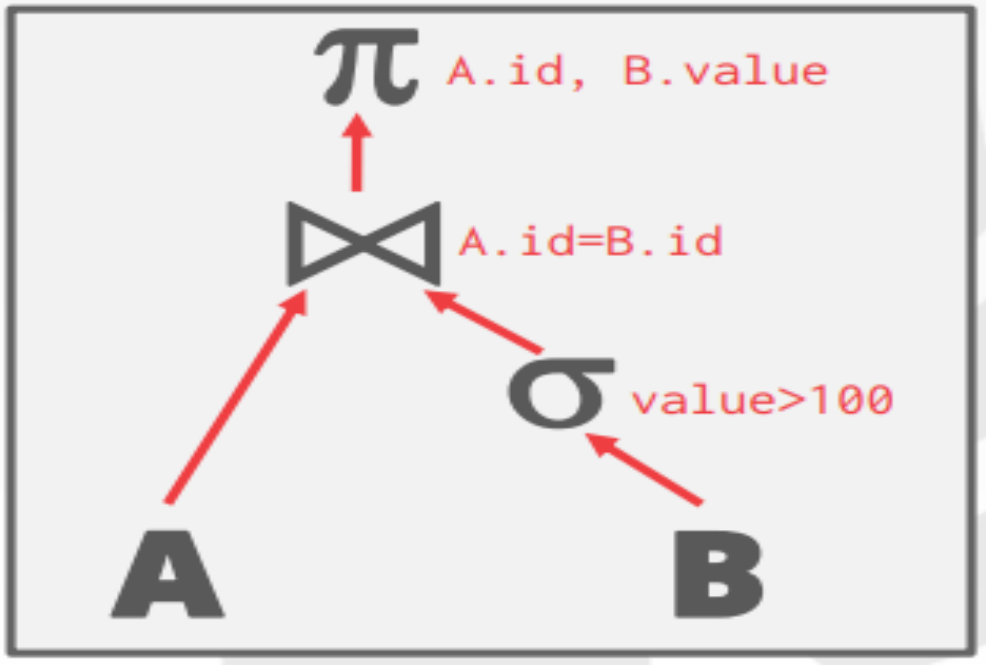
**Graphe de requêtes :**

✓Une structure de données graphique qui correspond à **une expression du calcul relationnel**. Il n'indique pas l'ordre dans lequel les opérations doivent être effectuées en premier. Il n'y a qu'un seul graphe correspondant à chaque requête.

# Graphes et Arbres



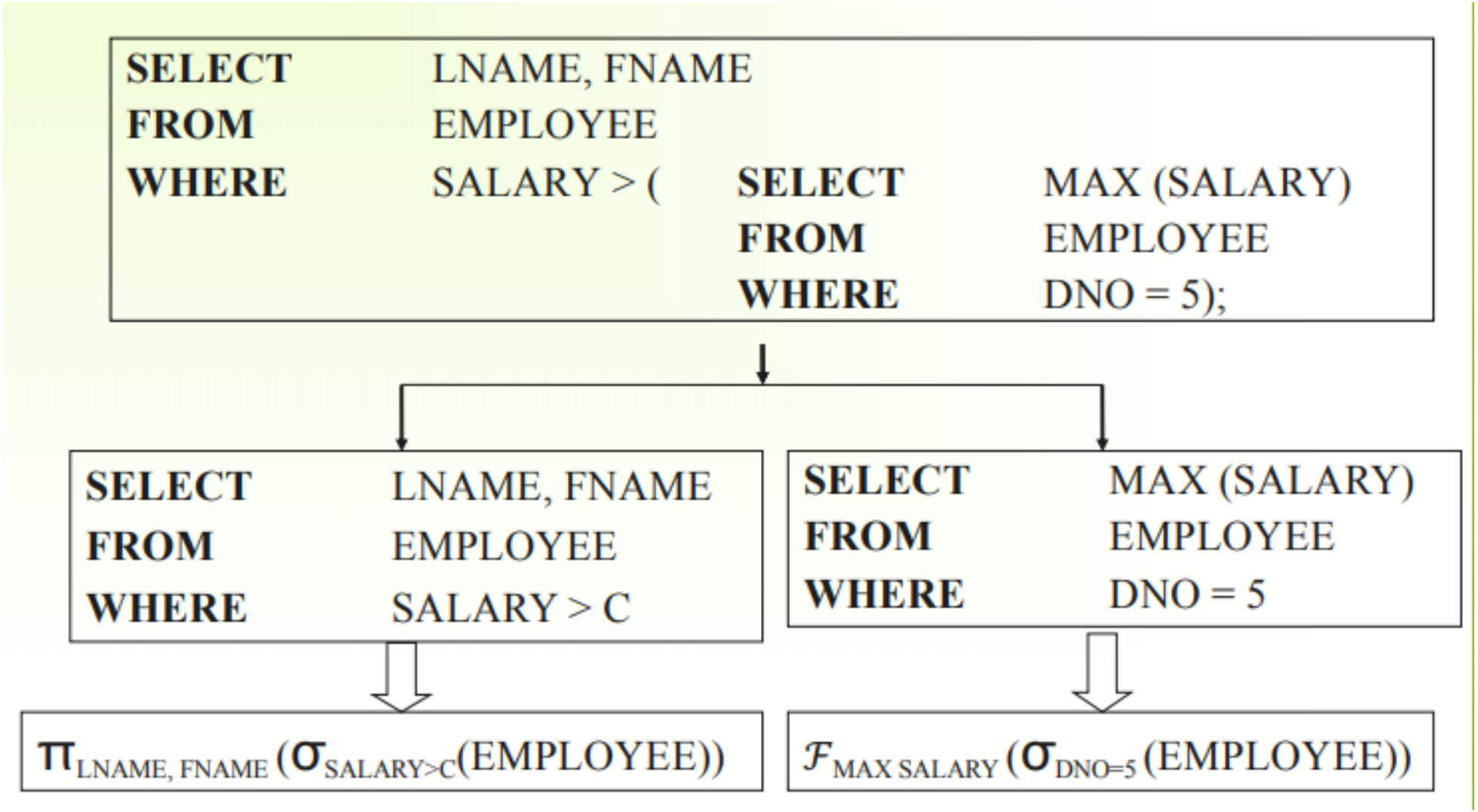

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



# Arbres : Blocs de requête

- **Le block de la requête** – l'unité minimale qui peut être optimisée et traduite dans des opérateurs algébriques.
- Un bloc de requêtes contient une seule expression **SELECT FROM WHERE**, ainsi que les clauses **GROUP BY** et **HAVING** si elles font partie du bloc.
- Dans des requêtes imbriquées dans une requête, ils sont identifiés comme des blocs de requête distincts.
- Dans les opérateurs d'agrégation en SQL, ils doivent être inclus dans l'algèbre étendue.

# Extractions des blocs



# COUTS LOGIQUES

# Couts logiques – nombre d’accès blocs

**Informations de catalogue utilisées dans les fonctions de coût**

- Informations sur la taille d'un fichier (**grandeur**)
- nombre d'enregistrements (tuples) **(r),**
- taille d'enregistrement **(R),**
- nombre de blocs physiques **(b)**
- facteur de blocage **(bfr),** nombre des valeurs dans un bloc physique

**Informations sur les index et les attributs d'indexation d'un fichier**

- Nombre de niveaux **(x)** de chaque index multiniveau
- Nombre de blocs d'index de premier niveau **(bI1)**
- Nombre de valeurs distinctes **(d)** d'un attribut
- Sélectivité **(sel)** d'un attribut
- Cardinalité**(s)** de sélection d'un attribut. **(s = sel * r)**

# Algorithmes simples de recherche π et σ (1)

**S1 – recherche linéaire, brute force sur un fichier sans ordre prédéfinie**

**SELECT * FROM fichier_sans_ordre_sans_indexes WHERE colonne = valeur**

Récupérez chaque enregistrement du fichier et testez si ses valeurs d'attribut satisfont à la condition de sélection.

CS1 = b (nombre de blocs, en mode séquentiel), ou b/2 (condition d'égalité, trouvée)

**S2 – recherche binaire** - si la condition de sélection implique une comparaison d'égalité sur un attribut clé sur lequel le fichier est ordonné

**SELECT * FROM fichier_ordonné_sur_colonne1 WHERE colonne1 = valeur**

CS2 = $\log_2 b$ + int(s/bfr) − 1 ou $\log_2 b$ si on aune condition d'égalité

**S3a – indexe primaire**

Si la condition de sélection implique une comparaison d'égalité sur un attribut clé avec un index primaire (ou une clé de hachage), utilisez l'index primaire (ou la clé de hachage) pour récupérer l'enregistrement.

**Indexe primaire sur (colonneclé1) : SELECT * FROM table WHERE coloneclé1 = a**

CS3a = x + 1 , ou x a nombre des niveaux dans l'indexe B*-Tree

# Algorithmes simples de recherche π et σ (2)

**S3b – indexe hash**

comparaison d'égalité sur un attribut de clé avec une clé de hachage, pour récupérer l'enregistrement. Cette condition récupère un seul enregistrement (au plus)

**Indexe hash (colonneclé1)**

**SELECT * FROM table WHERE coloneclé1 = a ;** CS3b = 1

**S4 – indexe primaire pour trouver plusieurs rangées**

Si la condition de comparaison est >, ≥, < ou ≤ sur un champ clé avec un index primaire, utilisez l'index pour trouver l'enregistrement satisfaisant la condition d'égalité correspondante, puis récupérez tous les enregistrements suivants dans le fichier (ordonné).

**Indexe primaire sur (colonneclé1)**

**SELECT * FROM table WHERE coloneclé1 = a AND colonne2 = b ;** CS4 = x + (b/2).

**S5 – clustering index pour plusieurs rangées**

Si la condition de sélection implique une comparaison d'égalité sur un attribut non clé avec un index cluster,  utilisez l'index de clustering pour récupérer tous les enregistrements satisfaisant la condition de sélection.

CS5 = x + int(s/bfr).

# Algorithmes simples de recherche π et σ (3)

**S6 – Indexe secondaire B-Tree**

À utiliser pour récupérer des enregistrements sur des conditions impliquant >,>=, < ou <=. Peut également être utilisé pour une comparaison d'égalité, pour une recherche d'enregistrement unique si le champ d'indexation a des valeurs uniques (est une clé) ou pour récupérer plusieurs enregistrements si le champ d'indexation n'est pas une clé.

Une seule condition d'égalité : CS6a = x + 1 + s

Plusieurs conditions combinées en inégalité (range): CS6b = x + (bI1/2) + (r/2)

**S7a – Indexe bitmap**

Si la condition de sélection implique un ensemble de valeurs pour un attribut, les bitmaps correspondants pour chaque valeur peuvent être associés par un OR ou AND pour donner l'ensemble des identifiants d'enregistrement qui se qualifient.

**Cas:** Index bitmap COLONNE1 , Index bitmap COLONNE3 , Index bitmap COLONNE3

**SELECT * FROM table WHERE colone1indéxée = a**

**AND (colonne2indexée = b OR colonne3indexée = c)**

# Algorithmes simples de recherche π et σ (4)

**S7b – Indexe de type fonction**

S'il existe un index fonctionnel défini comme :

**CREATE INDEX revenu_ix  ON EMPLOYÉ (Salaire + Salaire*Commission_pct) );**

alors cet index peut être utilisé pour récupérer les dossiers des employés qui se qualifient. Notez que la manière exacte dont la fonction est écrite lors de la création de l'index est sans importance mais il faut respecter la syntaxe exacte de l’appel.

**SELECT * FROM employé WHERE Salaire + Salaire*Commission_pct = 200000**

Le cout est similaire pour S6 conditions d’égalité ou non.

# Algorithmes simples de recherche π et σ (5)

**S8 – Conjonction avec un indexe individuel**

Si un attribut impliqué dans une seule condition simple dans la condition de sélection conjonctive a un chemin d'accès qui permet l'utilisation de l'une des méthodes S2 à S6, utilisez cette condition pour récupérer les enregistrements, puis vérifiez si chaque enregistrement récupéré satisfait les conditions restantes dans la condition de sélection conjonctive

**Indexe sur (colonne1)**

**SELECT * FROM table WHERE colone1indéxée = a AND colonne2 = b**

**S9 – Conjonction avec un indexe composé**

Si deux attributs ou plus sont impliqués dans des conditions d'égalité dans la condition conjonctive et qu'un index composite (ou structure de hachage) existe sur le champ combiné, nous pouvons utiliser l'index directement avec accès bloc.

On utilise S3a, S5 ou S6 depndemment d’indexe.

**Indexe sur (colonne1 , colonne2)**

**SELECT * FROM table WHERE colone1indéxée = a AND colonne2indexée = b**

# Algorithmes simples de recherche π et σ (6)

**S10 – Conjonction sur un indexe résulté d'une intersection avec des pointeurs de blocs**

**SELECT * FROM table WHERE colone1indéxée = a AND colonne2indexée = b**

Cette méthode est possible si des index secondaires sont disponibles sur tous (ou certains) des champs impliqués dans les conditions de comparaison d'égalité dans la condition conjonctive et si les indexes incluent des pointeurs d'enregistrement (plutôt que des pointeurs de bloc).

# Algorithmes simples de recherche π et σ (7)

## S11 – Disjonctions sans indexes bitmaps

$$OP4' : \sigma_{Dno=5 \text{ OR } Salary > 30000 \text{ OR } Sex = 'F'}(EMPLOYEE)$$

les enregistrements satisfaisant à la condition disjonctive sont la réunion des enregistrements satisfaisant aux conditions individuelles.

Si l'une des conditions n'a pas de chemin d'accès, l'approche est une recherche linéaire brute.

# Exemples de calcules logiques de couts

La table EMPLOYEE a $r_E$ = 10 000 enregistrements stockés dans $b_E$ = 2 000 blocs de disque avec un facteur de blocage $bfr_E$ = 5 enregistrements/bloc.

- **index cluster sur Salary**, niveaux B-tree xSalary = 3 et une cardinalité moyenne cardSalary = 20. selSalary = 20/10000 = 0,002
- **index secondaire sur l'attribut clé Ssn**, avec xSsn = 4 (cardSsn = 1, selSsn = 0,0001).
- **index secondaire sur l'attribut non clé Dno**, avec $x_{Dno}$ = 2 et de premier niveau blocs d'index $bl1_{Dno}$ = 4. NDV (Dno, EMPLOYEE) = 125 valeurs distinctes pour Dno, la sélectivité est selDno = (1/ NDV (Dno, EMPLOYEE)) = 0,008, et la cardinalité est cardDno = (rE * selDno) = (rE/NDV (Dno, EMPLOYÉ)) = 80.
- **index secondaire sur Sexe**, avec xSex = 1. Il y en a NDV (Sex, EMPLOYEE) = 2 valeurs pour l'attribut Sex, la cardinalité est selSex = (rE/NDV (Sex, EMPLOYEE)) = 5000.

# Exemples de calcules logiques de couts

**$\sigma_{Dno=5\ AND\ SALARY>30000\ AND\ Sex='F'}$ (EMPLOYEE)**

1. CS6a = x + 1 + s => Dno = 5 donne l'estimation de coût :

**CS6a = 82.**

2. CS4 = x + (b/2) => Salaire > 30000 donne une estimation de coût :

**CS4 = xSalaire + ($b_E$/2) = 3 + (2000/2) = 1003**

3. CS6a = x + 1 + s => L'utilisation de la condition (Sex = 'F') donne d'abord une estimation de coût :

**CS6a = $x_{Sex}$ + $s_{Sex}$ = 1 + 5000 = 5001.**

# Algorithmes de tri

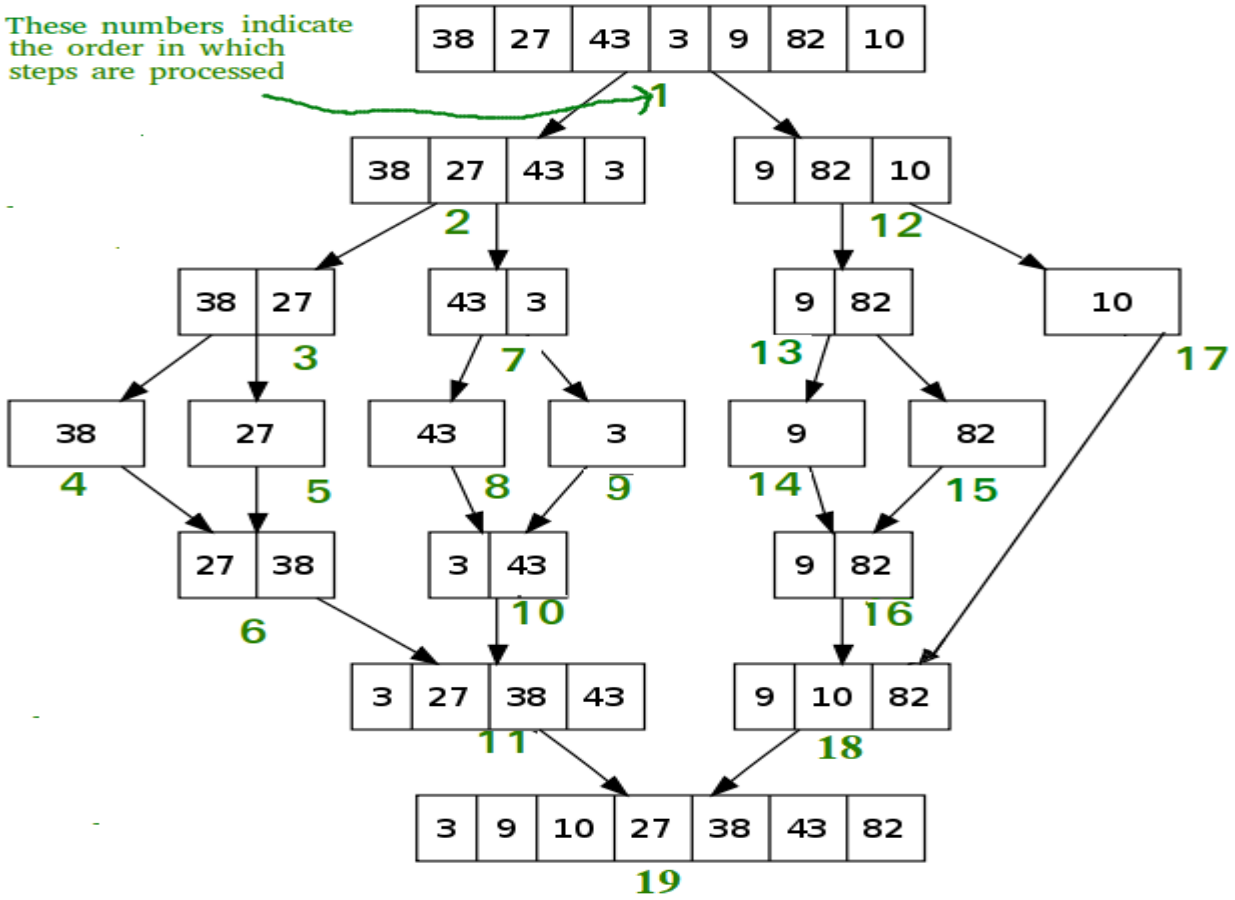
**ORDER BY, JOIN, UNION, INTERSECTION, elimination de duplications utilisant DISTINCT**

**Triage externe** - tri de grands fichiers d'enregistrements qui n’entre pas dans la mémoire principale , les couts physiques sont plus lentes

**Triage interne** – les fichiers rentrent dans la mémoire ou des blocs existent dans la memoire, le tri est plus rapide , les couts sont plus rapides

# Algorithmes SORT-MERGE – le plus populaire

## Il est basé sur le principe de *diviser-et- régner*



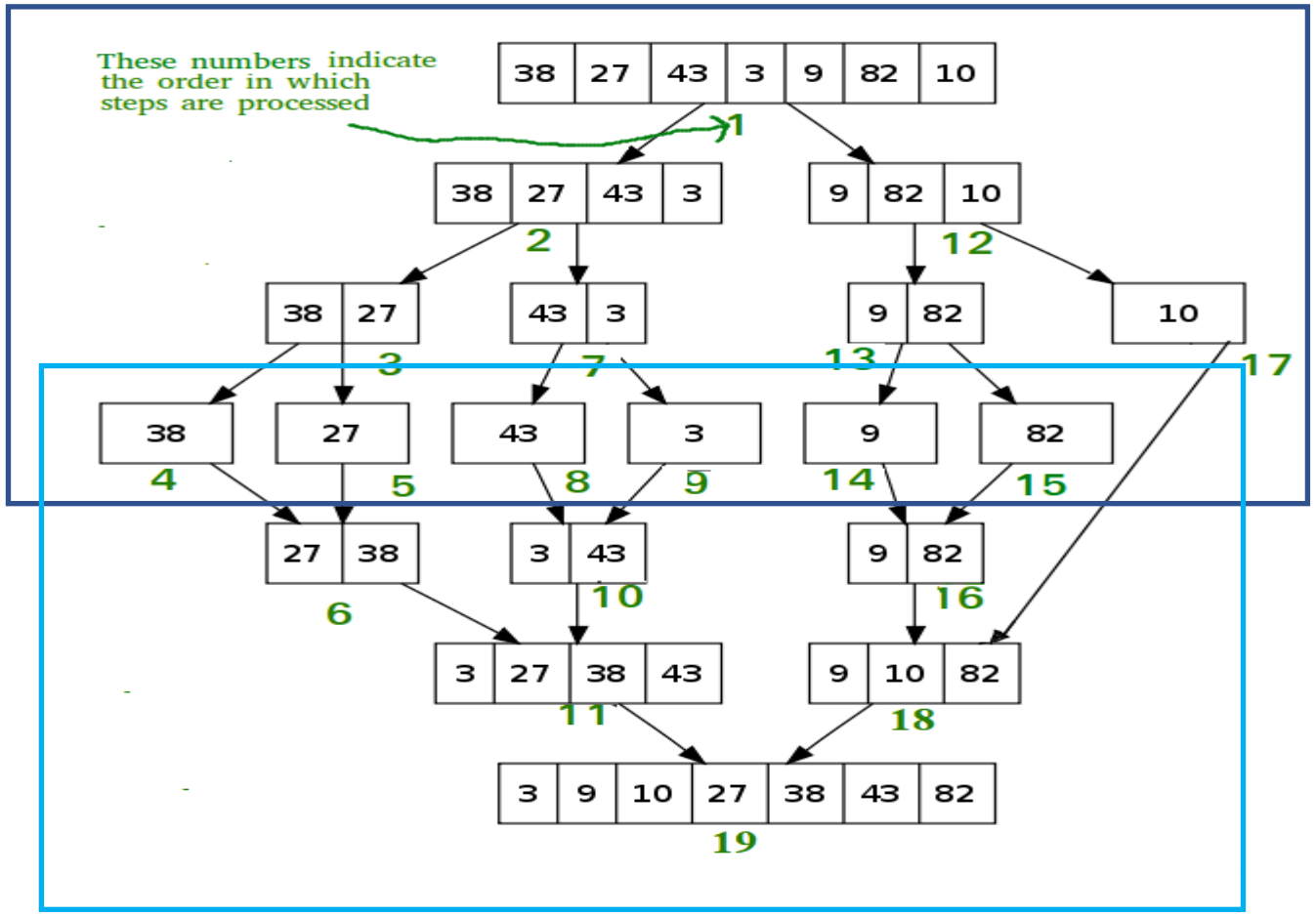

```
MERGE_SORT(arr, beg, end)
if beg < end  set mid = (beg +
end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)  end
of if
END MERGE_SORT
```

**Complexité**

O(N log(N))

# Algorithmes Tri : SORT-MERGE

## Sort-Merge est l'algorithme typique , basé sur le principe de *diviser-et- régner*




**Phase de tri :** les exécutions du fichier pouvant tenir dans la mémoire tampon sont triées et réécrites sur le disque en tant qu'exécutions triées temporaires
nR = [b/nB]+1 (nR is #executions, nB nombre de buffers, b = #blocs)
b = 1024, nB = 5,
**Merge:** les passages triés sont fusionnés au cours d'une ou plusieurs passes.
dM = # pistes pouvant être fusionnées en un seul passage.
dM = min(nB-1 , nR )
#passes = int($\log_{dM}$ (nR)) + 1
Ex.: dM = 4, #passes = $\log_4$ 205 +1 = 4

**#acces blocs = (2*b)+(2*(b*($\log_{dM}$ b)))**

# Couts de jointures – sélection rt cardinalité

Selection de la jointure :

**$js = |(R \bowtie c\ S)| / |(R \times S)| = |(R \bowtie c\ S)| / (|R| * |S|)$**

**$js = 1/ \max (NDV (A, R), NDV (B,S))$**

Cardinalité de la jointure :

**$jc = js * |R| * |S|$**

# Algorithmes de jointure (1)

## J1 – Nested-Loop, force brute, aucun index

- Pour **(1)** chaque enregistrement **t** dans **R** (boucle externe), **(2)** récupérez chaque enregistrement **s** de **S** (boucle interne) et (3) testez si les deux enregistrements satisfont la condition de jointure **t[A] = s[B].**

$$CJ1 = bR + (bR * bS) + ((js * |R| * |S|)/bfrRS)$$

## J2 – Index-Based NL

- Si un index (ou une clé de hachage) existe pour l'un des deux attributs de jointure, par exemple l'attribut B du fichier S, **(1)** récupérez chaque enregistrement dans R (boucle sur le fichier R), **(2)** utilisez la structure d'accès (telle qu'un index ou une clé de hachage) pour **(3)** récupérer les enregistrements correspondants de S qui satisfont s[B] = t[A]. Si on utilise un index primaire :

$$CJ2c = bR + (|R| * (xB + 1))+ ((js * |R| * |S|)/bfrRS$$

# Algorithmes de jointure (2)

**J3 – Sort-Merge**

Si les enregistrements de R et S sont physiquement triés (ordonnés) , les deux fichiers sont (1) et (2) analysés simultanément dans l'ordre des attributs de jointure, en faisant correspondre les enregistrements qui ont les mêmes valeurs pour A et B. Si les fichiers ne sont pas triés, ils peuvent être triés en premier en utilisant le tri externe.

$$CJ3a = bR + bS + ((js * |R| * |S|)/bfrRS)$$

**J4 – Hash-Join - deux phases**

- **Partitionnement -** est utilisé lorsque les projections des tables jointes ne sont pas déjà triées sur les colonnes de jointure. Dans ce cas, l'optimiseur crée une table de partitions de hash en mémoire sur la colonne de jointure de la table interne.
- **Validation -** l'optimiseur analyse la table externe pour rechercher des correspondances avec la table de hachage et joint les données des deux tables en conséquence. Le coût d'exécution d'une jointure par hachage est faible si la totalité de la table de hachage peut entrer dans la mémoire. Le coût augmente considérablement si la table de hachage doit être écrite sur le disque.

$$CJ4 = 3 * (bR + bS) + ((js * |R| * |S|)/bfrRS)$$

# COUTS PHYSIQUES

# Couts de traitement – Composants

**Choix #1 : Coûts physiques**

→ Prédire les cycles CPU, les E/S de disque, les manques de cache, la consommation de la mémoire, communication réseau, pré-chargement, et autres.

→ Dépend fortement du matériel.

**Choix #2 : Estimations logiques**

→ Estimer la taille des résultats par opérateur ou le nombre de blocs accèdes.

→ Moins dépendant de l'algorithme de l'opérateur.

→ Besoin d'estimations pour la taille des résultats des opérateurs.

**Choix #3 : Coûts algorithmiques**

→ Complexité de la mise en œuvre de l'algorithme de l'opérateur.

# Model de cout physique PostgreSQL

Utilise une combinaison de coûts CPU et E/S qui sont  pondérée par des constantes « magiques ».

Les paramètres par défaut sont évidemment pour un résident sur disque

base de données sans beaucoup de mémoire :

→ Le traitement d'un tuple en mémoire est 400x plus rapide que la lecture d’un tuple du disque.

→ Les E/S séquentielles sont 4 fois plus rapides que les E/S aléatoires.

# Model de cout physique IBM DB2

- Caractéristiques des bases de données dans les catalogues système
- Environnement matériel (micro-benchmarks)
- Caractéristiques du dispositif de stockage (micro-benchmarks)
- Bande passante des communications (distribuée uniquement)
- Ressources mémoire (pools de tampons, tas de tri)
- Environnement :
  - → Nombre moyen d'utilisateurs
  - → Isolation et blocage
  - → Nombre de verrous disponibles

# Model de cout Oracle – couts physique

Oracle calcule les statistique de système, parce que chaque système est différent.

**Aucune statistique de charge de travail :**

**CPUSPEEDNW** - Vitesse du processeur

**IOSEEKTIM** - Le temps de recherche d'E/S en millisecondes

**IOTFRSPEED** - Vitesse de transfert d'E/S en millisecondes

**Statistiques liées à la charge de travail :**

**SREADTIM** - Temps de lecture d'un seul bloc en millisecondes

**MREADTIM** - Temps de lecture multi-bloc en ms

**CPUSPEED** - Vitesse du processeur

**MBRC** - Nombre moyen de blocs lus par lecture multi-bloc (voir **db_file_multiblock_read_count**)

**MAXTHR** - Débit maximal d'E/S (uniquement pour OPQ)

**SLAVETHR** - OPQ Factotum (esclave) débit (OPQ uniquement)

# Couts de traitement – Disque

Le nombre d'accès au disque dominera toujours le temps d'exécution d'une requête.

→ Les coûts CPU sont en général négligeables.

→ Doit prendre en compte les E/S séquentielles ou aléatoires.

C’est plus facile à modéliser si le SGBD a un contrôle total sur la gestion des tampons.

→ Nous connaîtrons la stratégie de remplacement, l'épinglage et supposerons un accès exclusif au disque.

# Model de cout physique – impact système

- Le nombre de processeurs disponibles et l'espace mémoire disponible, jouent un rôle dans la détermination de l'accélération globale. Une discussion détaillée de l'effet de ces paramètres est hors de notre portée.
- Un buffer de mémoire plus grand va accélérer les tris ou les accès séquentiels.
- Avoir une vitesse E/S bonne aide.

# Model de cout – impact parallélisme

- Le cout est associé à la complexité. C'est une mesure du temps.
- Si on ajoute du parallélisme , la complexité de l'algorithme est soit la même, soit plus grande (on exécute plus d'instructions et on consomme plus de ressources)
- Le cout évalué par les SGBDs sera  plus petit, parce que le temps d'exécution est plus petit.
- Le parallélisme de un SQL peut consommer les ressources d'autres utilisateurs et créer de problèmes sur le système en ensemble.

Exemples dans le laboratoire.

# Model de cout – intra et inter-noueds

**Intra-requêtes (même nœud)**

- Chaque opération individuelle peut être exécutée en distribuant les données entre plusieurs processeurs et en exécutant l'opération en parallèle sur ces processeurs.
- Pour réaliser une exécution parallèle d'une requête, on utilise un algorithme parallèle pour chaque opération impliquée dans la requête.

**Inter-requêtes (plusieurs nœuds)**

- Dans les architectures sans partage ou à disque partagé, difficile à réaliser.
- Les activités de verrouillage, de journalisation, etc. entre les processeurs doivent être coordonnées, les mises à jour des mêmes données par plusieurs processeurs doivent être évitées. Il doit y avoir une cohérence de cache.

Ex.: Hadoop, Apache Spark, Citus PostgreSQL

# HÉURISTIQUES (Règles)

# Règles de réécriture – *Query Rewrite*

Deux expressions d'algèbre relationnelle sont équivalentes si elles génèrent le même ensemble de tuples. Le SGBD peut identifier de meilleurs plans de requête sans un modèle de coût.

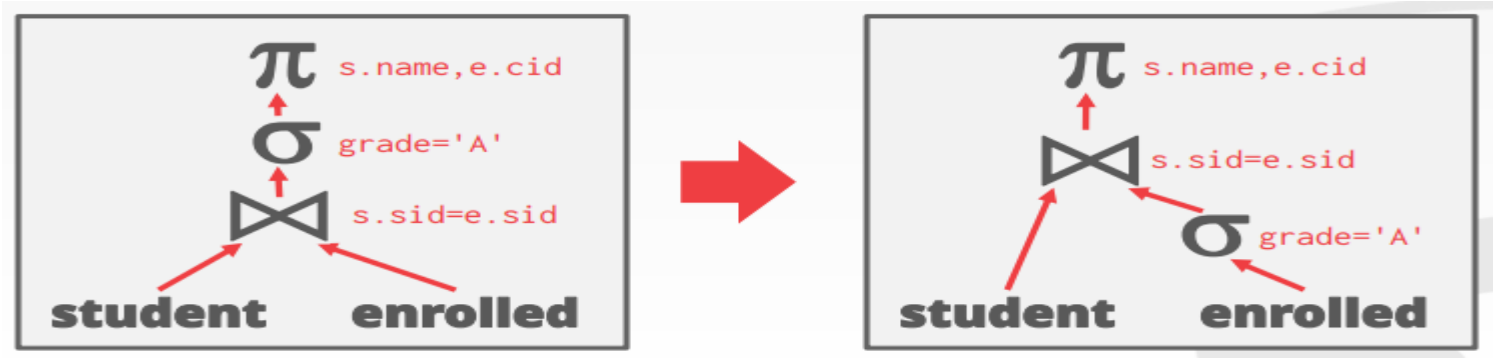
**SELECT s.name, e.cid FROM student**

**AS s, enrolled AS e WHERE s.sid = e.sid AND e.grade = 'A'**

$\pi_{name,\ cid}(\sigma_{grade='A'}(student \bowtie enrolled))$

vers

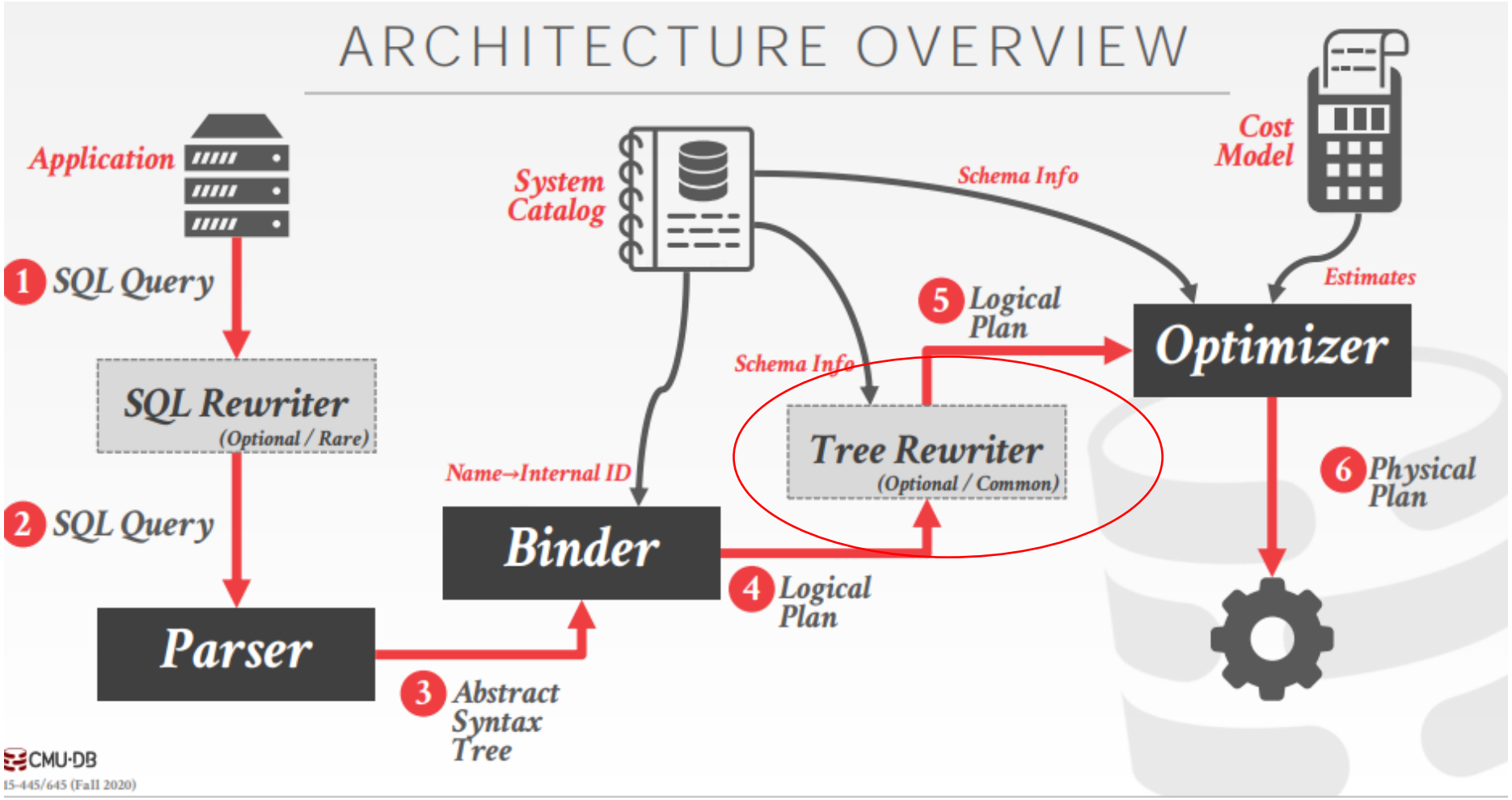
$\pi_{name,\ cid}\ (student \bowtie \sigma_{grade='A'}(enrolled))$



# Optimisation logique - Règles

- Transformez un plan logique en un plan logique équivalent à l'aide de règles de correspondance de modèles.
- L'objectif est d'augmenter la probabilité de trouver le plan optimal dans la recherche.
- Impossible de comparer les plans car il n'y a pas du modèle de coût mais peut coordonner une transformation dans le bon sens.

# Vue d'architecture d'optimiseur



# Règles de réécriture

**Sélections –** filtrer assez vite et utilisez le push-down

**Simplifications de prédicats complexes**

(X=Y AND Y=3) → X=3 AND Y=3

**Jointures –** commutative et associatives

$R \bowtie S = S \bowtie R$ $( R \bowtie S ) \bowtie T = R \bowtie ( S \bowtie T )$

**Projections**

→ Les effectuer tôt pour créer des tuples plus petits et réduire les résultats intermédiaires (si les doublons sont éliminés).

$(\sigma_C (R \times S)) = (R \bowtie_C S)$

→ Éliminer tous les attributs à l'exception de ceux qui sont demandés ou exigés (par exemple, les clés de jonction).

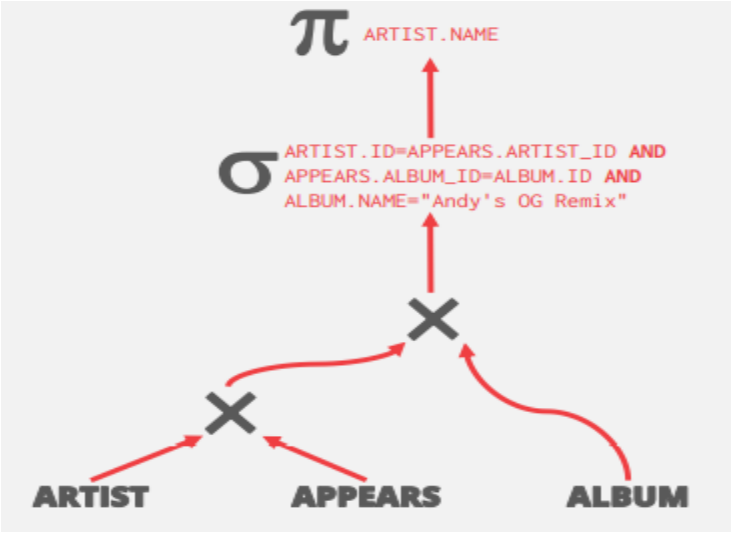
$\pi_L ( R \cup S ) = (\pi_L (R)) \cup (\pi_L (S))$

# Règles

- Dédoubler les prédicats conjonctifs
- Prédicat déroulant (push-down)
- Remplacer les produits cartésiens par des jointures
- Projection déroulante (push-down)

# Règles

**Décomposez les prédicats conjonctifs dans les formes les plus simples :**




**SELECT** ARTIST.NAME

**FROM** ARTIST, APPEARS, ALBUM

**WHERE**

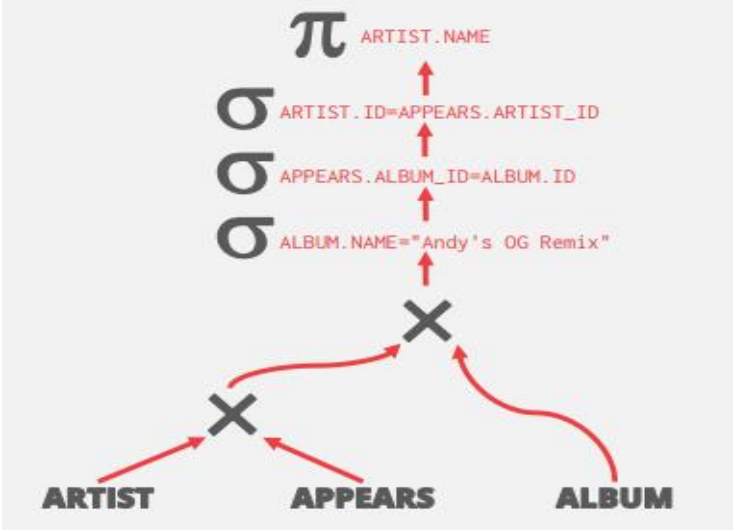
ARTIST.ID=APPEARS.ARTIST_ID

**AND**

APPEARS.ALBUM_ID=ALBUM.ID

**AND**

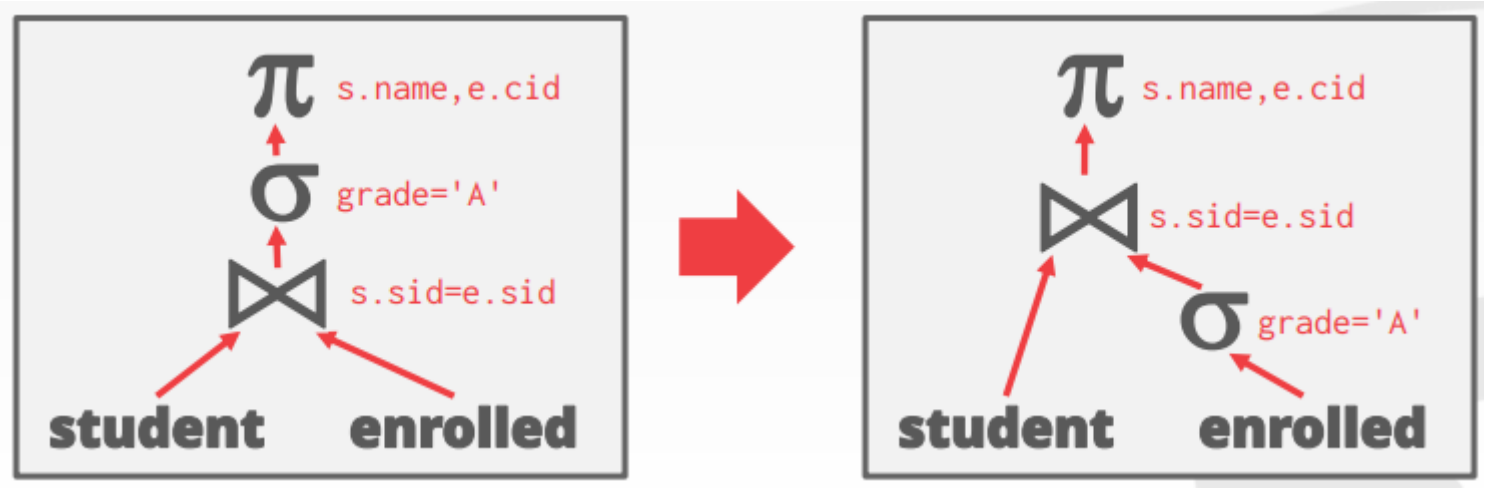
ALBUM.NAME="Andy's OG Remix"



# Heuristiques – Règles – Pushdown

**SELECT** s.name, e.cid **FROM** student AS s, enrolled AS e

**WHERE** s.sid = e.sid AND e.grade = 'A'




$$\pi_{\text{name, cid}}(\text{student} \bowtie (\sigma_{\text{grade='A'}}(\text{enrolled})))$$

# Heuristiques – Produits cartésiens

**SELECT** ARTIST.NAME

**FROM** ARTIST, APPEARS, ALBUM

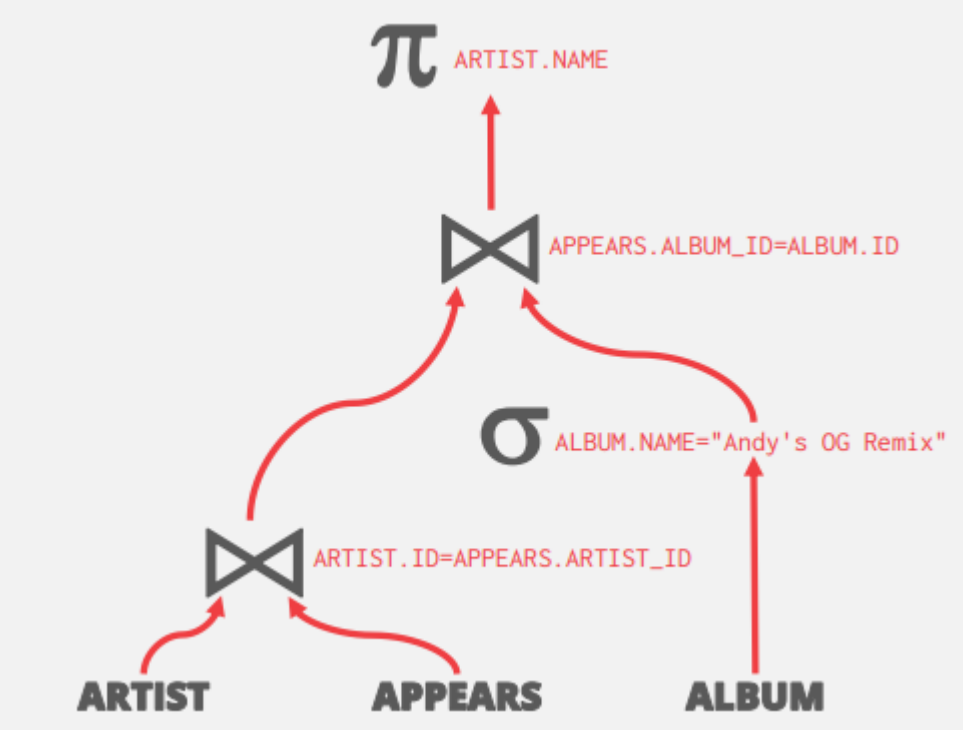
**WHERE**

ARTIST.ID=APPEARS.ARTIST_ID

**AND** APPEARS.ALBUM_ID=ALBUM.ID

**AND** ALBUM.NAME="Andy's OG Remix"

Remplacez tous les produits cartésiens par des jointures internes à l'aide des prédicats de jointure.



# Heuristiques – Push-down de projections

**SELECT** ARTIST.NAME

**FROM** ARTIST, APPEARS, ALBUM

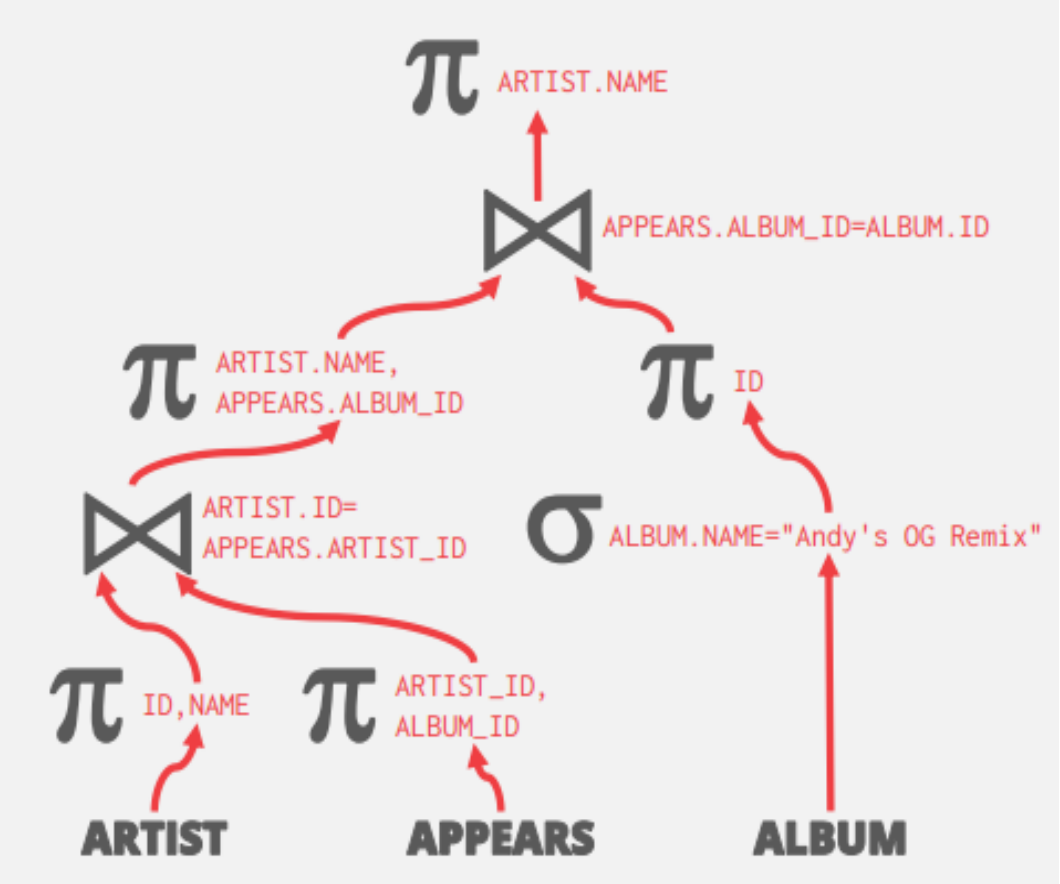
**WHERE**

ARTIST.ID=APPEARS.ARTIST_ID

**AND** APPEARS.ALBUM_ID=ALBUM.ID

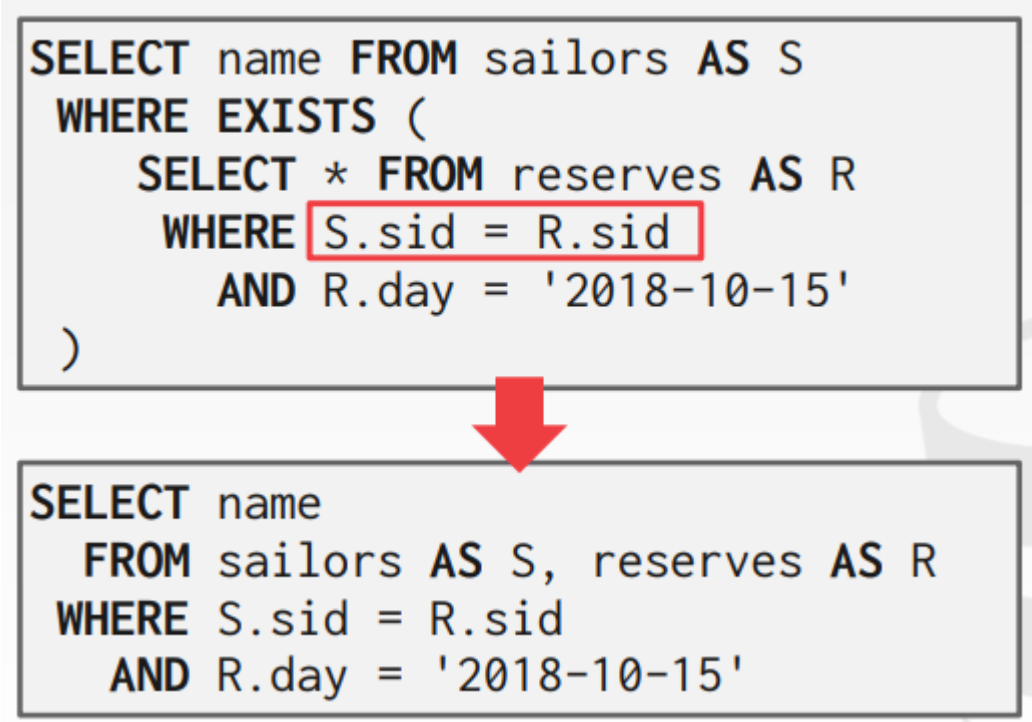
**AND** ALBUM.NAME = "Andy's OG Remix"

Éliminez les attributs redondants d'au début



# Heuristiques – Requêtes imbriqués – Nested

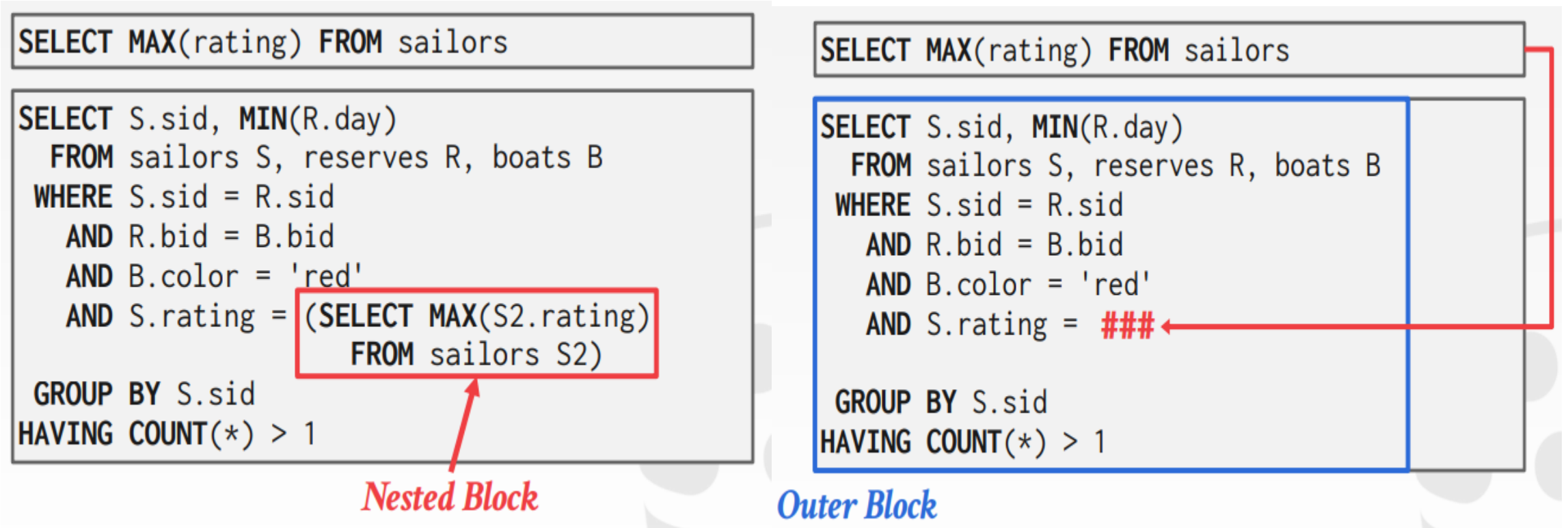
Applatisation



# Heuristiques –Décomposition

Pour les requêtes plus difficiles, l'optimiseur divise les requêtes en blocs, puis se concentre sur un bloc à la fois.

Les sous-requêtes sont écrites dans une table temporaire qui est supprimée une fois la requête terminée.



# Heuristiques – Réécriture des expressions (1)

Un optimiseur transforme les expressions d'une requête (par exemple, les prédicats de la clause WHERE) en l'ensemble optimal/minimal d'expressions.

Implémenté à l'aide de clauses **if/then/else** ou d’un moteur de règles de correspondance de modèles.

→ Rechercher des expressions qui correspondent à un modèle.

→ Lorsqu'une correspondance est trouvée, réécrivez l'expression.

→ Arrêtez s'il n'y a plus de règles qui correspondent.

# Heuristiques – Réécriture des expressions (2)

**Prédicats ne-nécessaires**

SELECT * FROM A WHERE 1=0 – nonsense

**SELECT * FROM A** WHERE 1=1

**Élimination de jointures**

**SELECT A1.* FROM A** AS A1 JOIN A AS A2 ON A1.id=A2.id

**Projections inutiles**

**SELECT * FROM A** AS A1 WHERE EXISTS (SELECT val FROM A AS A2 WHERE A1.id = A2.id);

**Fusion de prédicats**

**SELECT * FROM A WHERE** val **BETWEEN** 1 AND 100 **OR** val **BETWEEN** 50 AND 150;

# Statistiques

# Estimations de couts logiques – reprise

**Informations de catalogue utilisées dans les fonctions de coût**

- Informations sur la taille d'un fichier
- nombre d'enregistrements (tuples) (r),
- taille d'enregistrement (R),
- nombre de blocs (b)
- facteur de blocage (bfr)

**Informations sur les index et les attributs d'indexation d'un fichier**

- Nombre de niveaux (x) de chaque index multiniveaux
- Nombre de blocs d'index de premier niveau (bI1)
- Nombre de valeurs distinctes (d) d'un attribut
- Sélectivité (sel) d'un attribut
- Cardinalité(s) de sélection d'un attribut. (s = sel * r)

# Statistiques

Manual invocations:

→ PostgreSQL/SQLite: **ANALYZE**

→ Oracle/MySQL: **DBMS_STATS.GATHER_STATS**

→ SQL Server: **UPDATE STATISTICS**

→ DB2: **RUNSTATS**

Les SGBD collectent des échantillons à partir de tables pour estimer les sélectivités. Mettez à jour les statistiques lorsque les tables changent de manière significative.

# Statistiques approximatives

Maintenir des statistiques exactes sur la base de données est impossible.

Les algorithmes d'analyse utilisent des structures de données approximatives appelées esquisses pour générer des estimations limitées par les erreurs.

→ Compte distinct

→ Quantiles

→ Valeurs fréquentes de tuples

→ Esquisse de tuple (sketches)

DataSketches | (apache.org)

# Statistiques – sélectivité

Pour chaque relation R, le SGBD maintient la

information suivante:

→ NR : Nombre de tuples dans R.

→ NVD(A,R) : Nombre de valeurs distinctes pour l'attribut A.

La cardinalité de sélection SC(A,R) est le nombre moyen d'enregistrements avec une valeur pour un attribut A donné NR / V(A,R)

Notez que cette formule suppose l'uniformité des données.

La **sélectivité (sel)** d'un prédicat P est la fraction de tuples qui se qualifient.

Les statistiques de sélectivité sur les groups de colonnes sont importantes pour réduire la charge d’optimiseur.

# Sélectivité
# égalité, range, négation, conjonction , disjonction

Supposons que V(âge, people) a cinq valeurs distinctes (0–4) et NR = 5

Prédicat d'égalité : A=constante

**→ sel(A=constante) = card(P) / NR**

SELECT * FROM personnes WHERE âge = 2  - sel = 1/5

**→ sel(A>=a) = (Amax– a+1) / (Amax– Amin+1)**

sel(age>=2) ≈ (4–2+1) / (4–0+1) ≈ 3/5

**→ sel(not P) = 1 – sel(P)**

sel(age != 2) ≈  1 – (1/5) = 4/5

**→ sel(P1 ∧ P2) = sel(P1) · sel(P2)**

**→ sel(P1 ∨ P2) = sel(P1) + sel(P2) – sel(P1∧P2) = sel(P1) + sel(P2) – sel(P1) · sel(P2)**

# Corrélations des attributs

**(marque="Honda" ET modèle="Accord")**

Sous les hypothèses d'indépendance et d'uniformité, la sélectivité est :

→ $1/10 \times 1/100 = 0{,}001$

Mais puisque seule Honda fait de l'Accord le véritable sélectivité est de →
→ **1/100 = 0,01**

# Jointure – calcule de cout

Étant donné une jointure de R et S, la taille estimée d'une jointure sur l'attribut non clé A est d'environ :

**NR** = nombre de rangées de R

**NS** = nombre de rangées de S

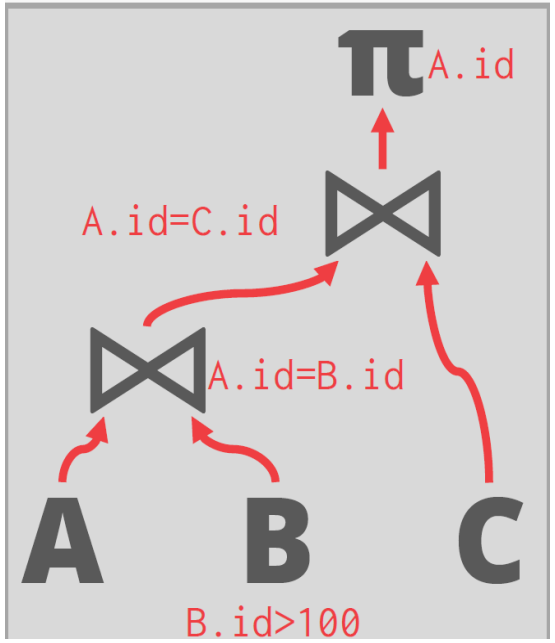
**NVD(A,R)** = nombre de valeurs distinctes de A en R

**NVD(A,S)** = nombre de valeurs distinctes de A en S

$$\textbf{estimation} \approx \textbf{NR} * \textbf{NS}/\max(\textbf{NVD}(\textbf{A}, \textbf{R}), \textbf{NVD}(\textbf{A}, \textbf{S}))$$

# Estimations de calcule de jointures - recap

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```




**1. Calculer la cardinalité de tables**

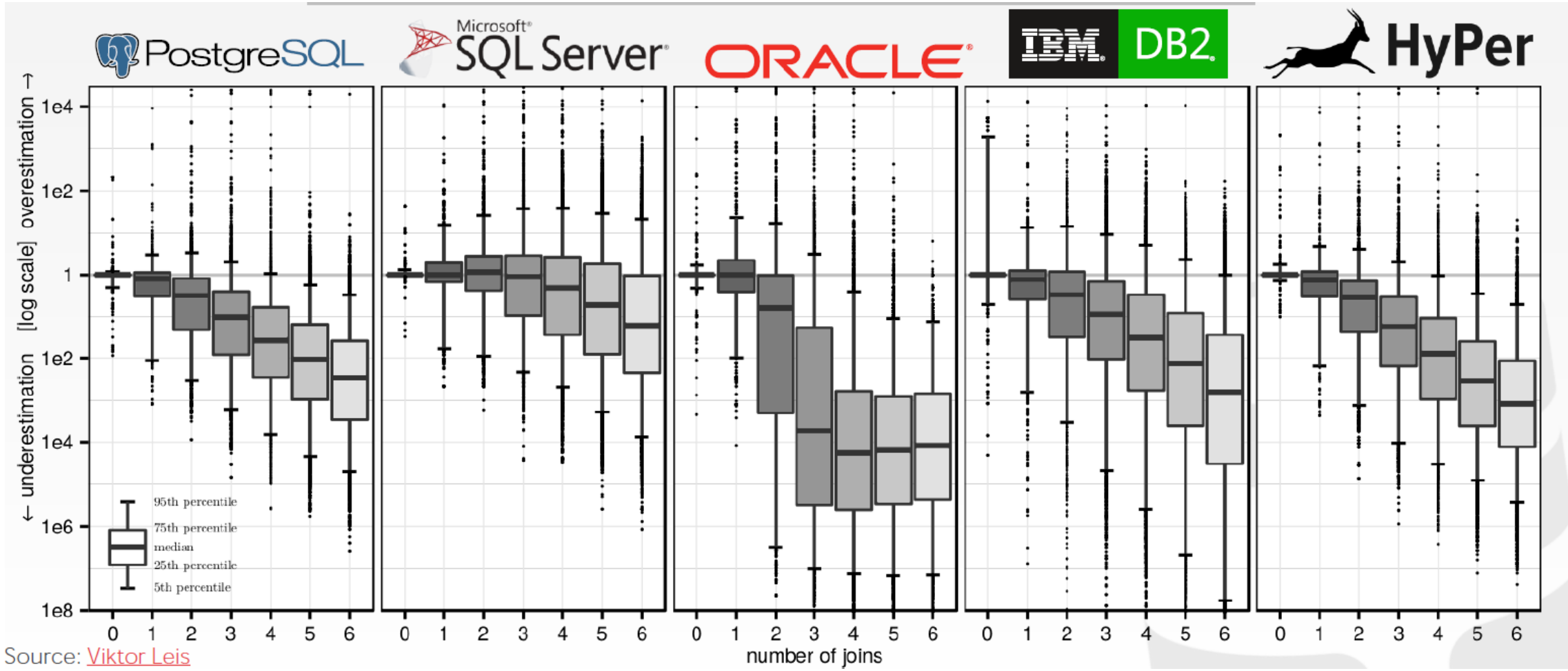
**A**→ |**A**|
**B**.id>100 → |**B**|×***sel***(**B**.id>100)
**C**→ |**C**|

**2. Calculer la cardinalité de jointures**

**A**⋈**B** = (|**A**|×|**B**|)/ ***max***(***sel***(**A**.id=**B**.id), ***sel***(**B**.id>100))

(**A**⋈**B**)⋈**C** = (|**A**|×|**B**|×|**C**|)/***max***(***sel***(**A**.id=**B**.id),***sel***(**B**.id>100),***sel***(**A**.id=**C**.id))

# Correctitude d'estimations par nombre de jointures




Source: Viktor Leis

# Hypothèses

**Hypothèse #1 : Données uniformes**

→ La répartition des valeurs est la même.

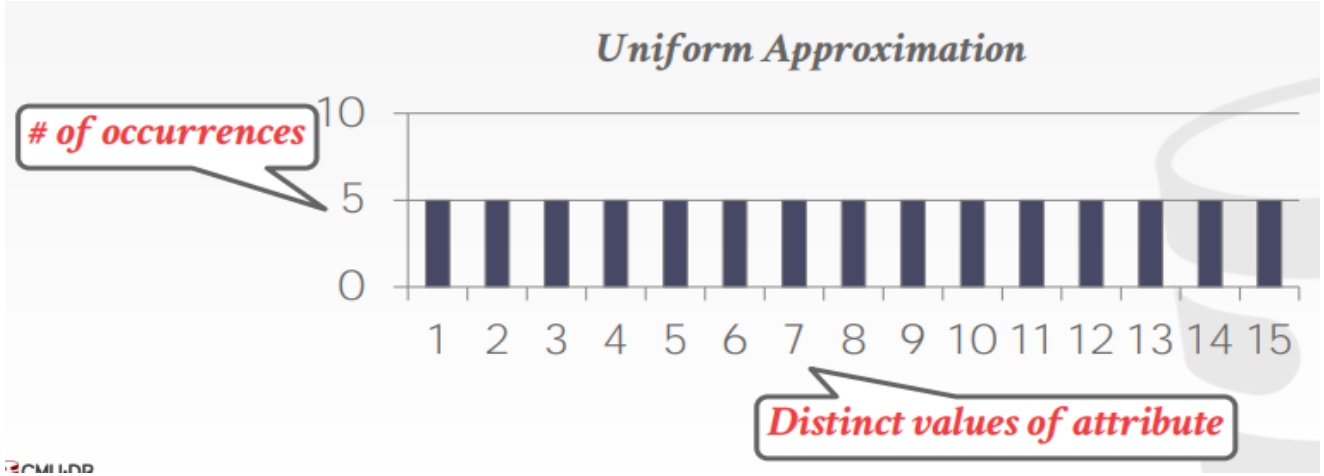
**Hypothèse #2 : Prédicats indépendants**

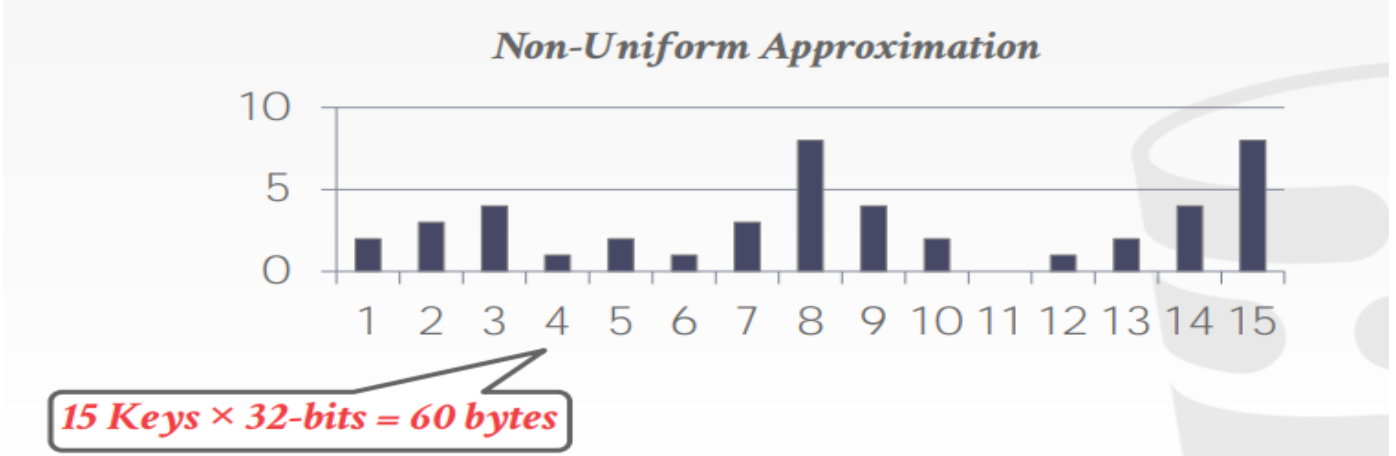
→ Les prédicats sur les attributs sont indépendants

**Hypothèse #3 : Principe d'inclusion**

→ Le domaine des clés de jointure se chevauche de sorte que chaque clé de la relation interne existe également dans la table externe.

# Données uniformes ?



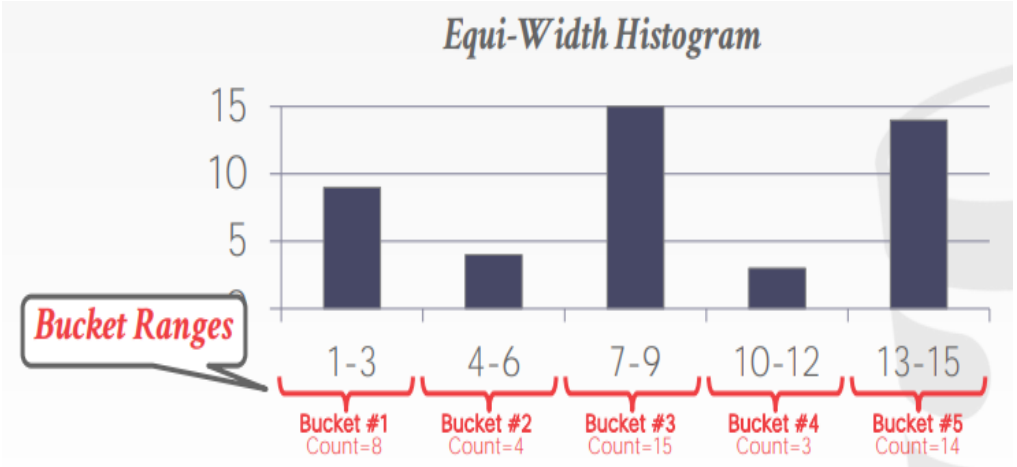
# Histogrammes et échantillons

Dans les bases de données réelles, les valeurs ne sont pas uniformément distribuées, et donc le maintien d'un histogramme est coûteux.

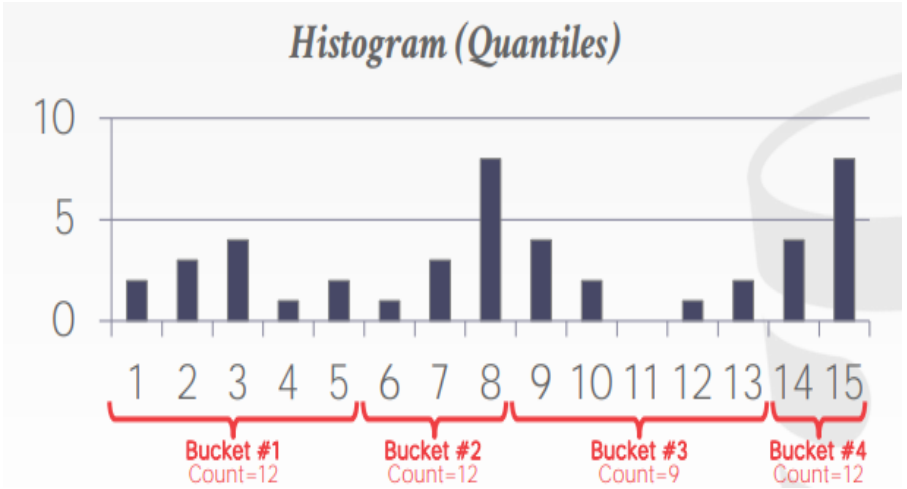
- Nous pouvons mettre des valeurs dans des seaux et limites pour réduire la taille des histogrammes. Cependant, cela peut entraîner des inexactitudes, car les valeurs fréquentes influenceront le nombre de valeurs peu fréquentes.
- Pour contrer cela, nous pouvons dimensionner les buckets de manière à ce que leur propagation soit la même. Ils détiennent chacun une quantité similaire de valeurs.
- Les SGBD modernes utilisent **l'échantillonnage** pour estimer les sélectivités des prédicats. Sélectionnez au hasard et maintenir un sous-ensemble de tuples à partir d'une table et estimer la sélectivité du prédicat en appliquant le calcule sur un échantillon.

# Histogrammes – largeur , profondeur

- Tous les compartiments ont la même largeur (c'est-à-dire le même nombre de valeurs).




Variez la largeur pour que le nombre total d'occurrences pour chaque seau est à peu près la même.



# Modèles probabilistes

Statistiques approximatives sur un ensemble de données.

Le modèle de coût peut remplacer les histogrammes par des croquis pour améliorer la précision de son estimation de sélectivité.

Exemples les plus courants :

→ **Count-Min Sketch** (1988) : Nombre approximatif de fréquences d'éléments dans un ensemble.

→ **HyperLogLog** (2007) : estimer le nombre des éléments distincts d'un ensemble.

# Choix de meilleur plan d’exécution

# Choix de meilleur plan d’exécution

- On a  discuté comment calculer les couts logiques et physiques.
- On a  discuté comment faire la reecriture basée sur les règles
- On a discuté les statistiques

Comment choisir le meilleur plan ?

- Après avoir effectué la réécriture basée sur des règles, le SGBD énumère différents plans pour la requête et estime leurs coûts.
- L’optimiseur  choisit le meilleur plan qu'il a vu pour la requête après avoir épuisé tous les plans ou un certain délai.

Attention, c’est moins d’une seconde pour le travail d’un optimiseur, et il doit choisir de dizaines de milliers de possibilités dépendamment de la complexité SQL.

# Optimisation de requêtes de couts

Les requetes peuvent être :

→ Relation unique

→ Relations multiples 2..N

→ Sous-requêtes imbriquées

→ Multiples choix d'accès  1..M

→ Multiples CPUs, ES 1..M

→ Multiples nœuds 1..M

# Optimisation de relations uniques

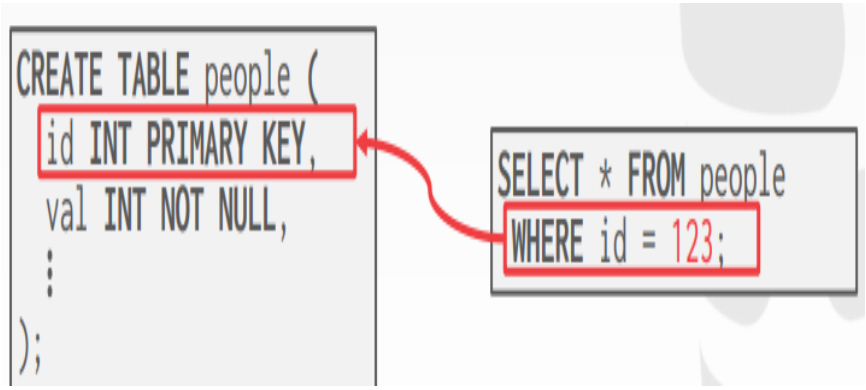
Choisir le meilleur accès, calculez et comparez:

→ Accès séquentiel

→ Recherche binaire (index clustérisés)

→ Scan d'index

→ Ordre d'évaluation des prédicats.




Les heuristiques simples sont souvent suffisantes pour cela.

Les requêtes OLTP sont particulièrement simples… choisir le meilleur index, les jointures sont sur des clés référentielles et les heuristiques marchent bien.

# Choisir le meilleur plan d’exécution

- Choisir un accès séquentiel, ou une recherche binaire, ou un balayage d'index pour chaque table consultée dans la requête **peut prendre du temps.**

- **Pour les requêtes à relations multiples**, le nombre de plans alternatifs augmente rapidement à mesure que le nombre de tables jointes augmente. Pour une jointure à N voies, le nombre de façons différentes d'ordonner les opérations de jointure est $4^n$ !

# Optimisation de relations multiples

Choix pré-défini pour commencer avec  les arbres-gauches dans les jointures.

**Astute :** Au lieu d'écrire les résultats intermédiaires dans des fichiers, utilisez de transfert en mémoire (pipelines, streams).

# Algorithme de recherche de meilleur plan d’exécution

1. Amener la requête sous forme interne canonique en algebre relationnelle
2. Générez des plans alternatifs.
3. Générez des coûts pour chaque plan.
4. Sélectionnez le plan avec le plus petit coût.

# Quelques règles rapides

- Pour les **sélections, commencez avec index scans.**
- Si la condition de sélection est **conjonctive**, utilisez la sélection avec **la plus petite cardinalité en premier.**
- Si les **relations sont triées** sur les attributs mis en correspondance dans une jointure, **préférez la jointure tri-fusion (sort-merge)** aux autres méthodes de jointure.
- Pour l'union et l'intersection de plus de deux relations, **considérez les relations dans l'ordre croissant de leurs cardinalités** estimées.
- Si l'un des arguments **d'une jointure a un index sur l'attribut de jointure, utilisez-le comme relation interne**.
- Si la **relation de gauche est petite et la relation de droite est grande** et qu'elle a un **index sur la colonne de jointure**, essayez une jointure imbriquée (***nested***) basée sur un index.
- Commencez avec **les ordres de jointure qui n’ont pas de produits cartésiens**.

# Requêtes imbriqués – *Nested*

Le SGBD traite les sous-requêtes imbriquées dans la clause WHERE comme des fonctions qui acceptent des paramètres et renvoient une valeur unique ou un ensemble de valeurs.

Deux approches :

1. Réécrire pour décorréler et/ou aplatir les requêtes.

2. Décomposer la requête imbriquée et stocker le résultat dans une sous-table

# Optimisation de relations multiples – planification

Énumérer l’ordre de recherche :

**→ Exemple : Arbre profond à gauche #1, Arbre profond à gauche #2…**

Énumérer les plans pour chaque opérateur de jointure etc.

**→ Exemple : Hash, Sort-Merge, Nested Loop…**

Énumérer les chemins d'accès pour chaque table, l’ordre d’analyse

**→ Exemple : Index #1, Index #2, Seq Scan…**

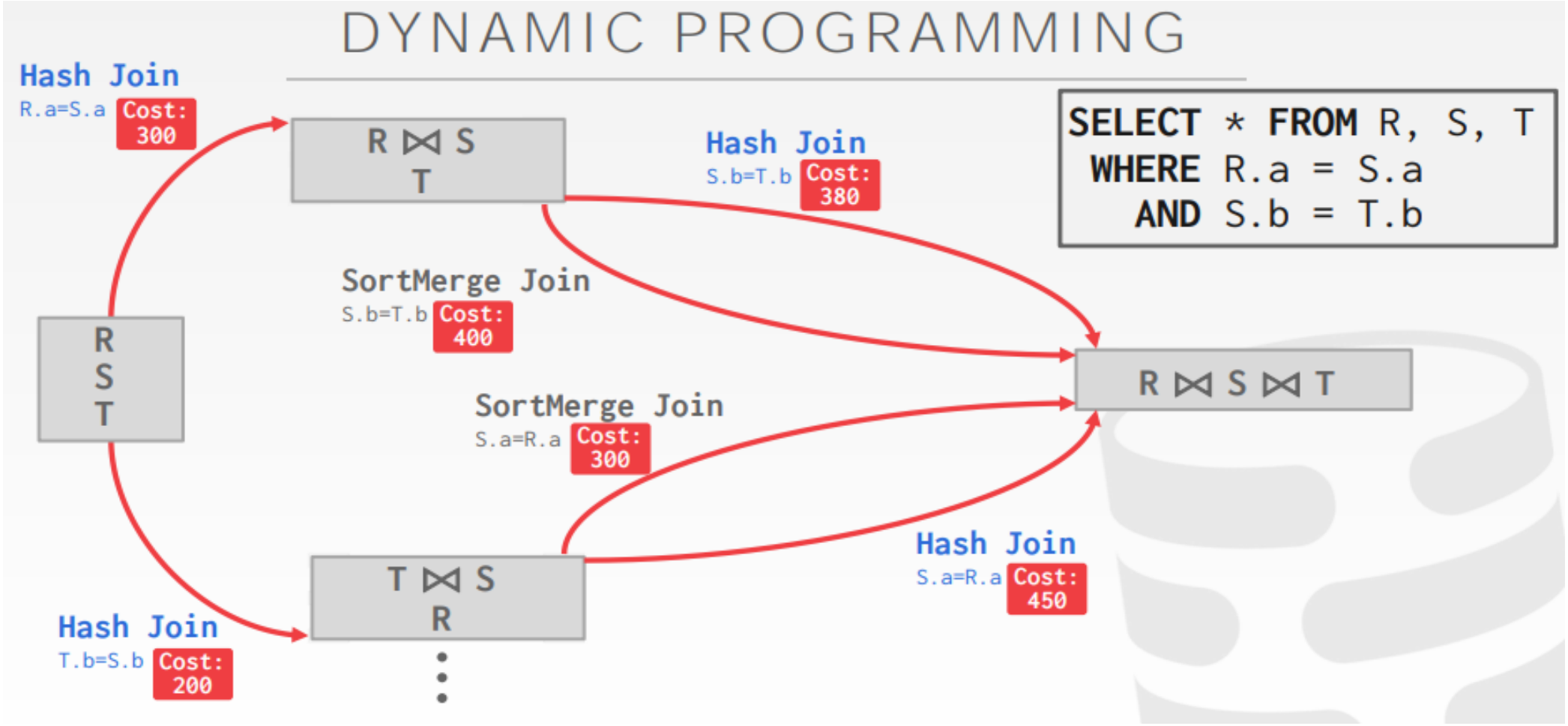
Utiliser la programmation dynamique pour réduire le nombre d'estimations de coûts.

# Programmation dynamique pour optimiser les accès

1. La structure d'une solution optimale est développée.
2. La valeur de la solution optimale est définie récursivement.
3. La solution optimale est calculée et sa valeur est calculée d’une manière récursive et utilise la mémoïsation

Mais time-out a un momment donné, impossible d’attendre trop.

# Exemple – planification – CMSU Andy Pavlo



# Optimiseur PostgreSQL

Examine tous les types d'arbres de jointure

→ Profond vers la gauche, profond vers la droite, partout

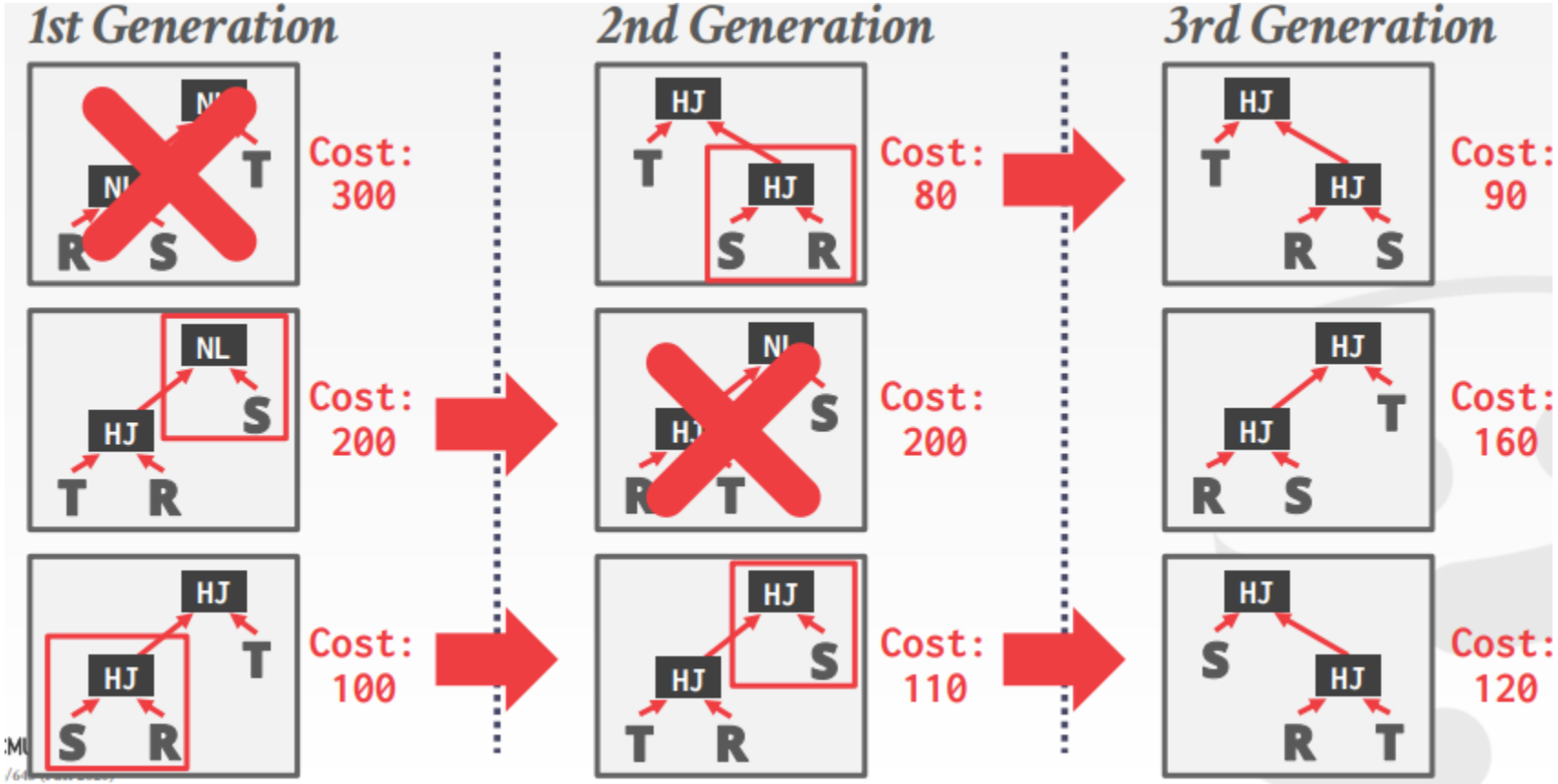
Deux implémentations d'optimiseur :

→ Approche de programmation dynamique traditionnelle

→ Optimiseur de requête génétique (**GEQO**)

PostgreSQL utilise l'algorithme traditionnel lorsque le nombre de tables dans la requête est inférieur à 12 et passe à GEQO lorsqu'il y en a 12 ou plus.

# Optimiseur PostgreSQL – algorithme génétique



# Conclusions (1)

- **Dimensioner bien le typage des attributs**

Les comparaisons INT (integer) sont plus rapides que les comparaisons VARCHAR. INT occupent beaucoup moins d'espace que les VARCHAR.

C’est vrai aussi pour les accès non indexés que pour les accès indexés. La solution la plus rapide est une colonne INT INDEXÉE.

Normaliser bien , utilisant des ids numériques pour les clés, au lieu de varchars

# Conclusions (2) – placer les bons indexes

- Indexes uniques – les plus performantes
- Indexes composés quand les requêtes sont sur des ensembles des attributs indépendants
- Indexes bitmaps pour les petites cardinalités sur des systèmes non-transactionnels
- Indexes hash pour des requetes d’égalité

# Conclusions (3) – vue matérialisées

Une vue matérialisée est un ensemble de données pré-calculées dérivées d'une spécification de requête (le SELECT dans la définition de la vue) et stockées pour une utilisation ultérieure.

Comme les données sont précalculées, l'interrogation d'une vue matérialisée est plus rapide que l'exécution d'une requête sur la table de base de la vue.

Plus que ca , dans une requete, l'optimiseur se rend compte que c'est une vue matérialisée a l’intérieur d’une requête complexe et va prendre les données pre-calculées.

# Conclusions – écrire bien la requête

- Filtrez le plus tôt possible, projetez seulement ce qui compte
- Commencez avec les étapes les plus selectives
- Clarté , simplification de jetons SQL

```
SELECT
 miner_name,
  MAX(fan_percentage)
FROM miner_data
WHERE miner_name IN
   (SELECT DISTINCT "name"
    FROM miners)
GROUP BY 1
ORDER BY 1;
```

```
Time: 9173.750 ms (00:09.174)
```

```
SELECT
  DISTINCT ON (miner_name) miner_name,
  MAX(fan_percentage)
FROM miner_data
GROUP BY miner_name;
```

```
Time: 2794.690 ms (00:02.795)
```

**Hypothèse :** miner_data et miners ont le même miner_name

# Conclusions – estimations de sélectivité

Tenir compte de :

→ Uniformité de données

→ Indépendance des attributs

→ Histogrammes

→ Sélectivité

→ Paramètres des optimiseur

# Conclusions – leçons

L’optimisation peut être plus importante qu’un ordinateur rapide.

Les estimations correctes de couts et un modèle parfait de couts sont difficiles a obtenir.

Accès séquentiels et hash joins et beaucoup d’indexes – en général sont bons choix

# Exemples

| OWNER | TABLE_NAME | TABLESPACE | NUM_ROWS | BLOCKS | CHAIN_CNT | AVG_ROW_ | DEGREE | INSTANCES | CACHE |
|---|---|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | LP_INTERVALS | NULL | 194 476 110 031 | 1 751 685 224 | 0 | 53 | 1 | 1 | N |

| AF | AG | AH | AI | CF |
|---|---|---|---|---|
| SAMPLE_SIZE | LAST_ANALYZED | PARTITIONED | IOT_TYPE | LOGICAL_REPLICATION |
| 194 476 110 031 | 06-25-2022 01:08:36 | YES | NULL | ENABLED |

| OWNER | INDEX_NAME | INDEX_TYPE | TABLE_OWNER | TABLE_NAME | TABLE_TYPE | UNIQUENESS | BLEVEL | LEAF_BLOCKS | DISTINCT_KEYS | AVG_LEAF_BLOCKS_PER_KEY |
|---|---|---|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | PK_LP_INTERVALS | NORMAL | XEDORA1A_MUDR | LP_INTERVALS | TABLE | UNIQUE | 3 | 941 918 406 | 194 476 110 031 | 1 |

| AB | AC | AD | AE | AF | AG |
|---|---|---|---|---|---|
| AVG_DATA_BLOCKS_PER_KEY | CLUSTERING_FACTOR | STATUS | NUM_ROWS | SAMPLE_SIZE | LAST_ANALYZED |
| 1 | 4 470 230 418 | N/A | 194 476 110 031 | 194 476 110 031 | 06-25-2022 01:52:59 |

# Exemple Oracle DBA_TABLES

| OWNER | TABLE_NAME | TABLESPACE | NUM_ROWS | BLOCKS | CHAIN_CNT | AVG_ROW_ | DEGREE | INSTANCES | CACHE |
|---|---|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | LP_INTERVALS | NULL | 194 476 110 031 | 1 751 685 224 | 0 | 53 | 1 | 1 | N |

| SAMPLE_SIZE | LAST_ANALYZED | PARTITIONED | IOT_TYPE | LOGICAL_REPLICATION |
|---|---|---|---|---|
| 194 476 110 031 | 06-25-2022 01:08:36 | YES | NULL | ENABLED |

# DBA_INDEXES

| OWNER | INDEX_NAME | INDEX_TYPE | TABLE_OWNER | TABLE_NAME | TABLE_TYPE | UNIQUENESS | BLEVEL | LEAF_BLOCKS | DISTINCT_KEYS | AVG_LEAF_BLOCKS_PER_KEY |
|---|---|---|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | PK_LP_INTERVALS | NORMAL | XEDORA1A_MUDR | LP_INTERVALS | TABLE | UNIQUE | 3 | 941 918 406 | 194 476 110 031 | 1 |

| AVG_DATA_BLOCKS_PER_KEY | CLUSTERING_FACTOR | STATUS | NUM_ROWS | SAMPLE_SIZE | LAST_ANALYZED | DE |
|---|---|---|---|---|---|---|
| 1 | 4 470 230 418 | N/A | 194 476 110 031 | 194 476 110 031 | 06-25-2022 01:52:59 | 1 |

# Exemple Oracle DBA_TAB_COLUMNS (1)

| OWNER | TABLE_NAME | COLUMN_NAME | DATA_TYPE | DATA_LENGTH | DATA_PRECISION | DATA_SCALE | NULLABLE | COLUMN_ID | NUM_DISTINCT | LOW_VALUE | HIGH_VALUE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | NUMBER | 22 | NULL | NULL | N | 1 | 13 968 930 | C42D323106 | C45A170509 |
| XEDORA1A_MUDR | LP_INTERVALS | INTERVAL_END_TIME | TIMESTAMP(0) | 7 | NULL | 0 | N | 2 | 35 609 | 7877061E180601 | 78780C01010101 |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | NUMBER | 22 | 21 | 6 | N | 3 | 5 384 346 | 3C3F2436611566 | C31B1629 |
| XEDORA1A_MUDR | LP_INTERVALS | ORIG_READ_VALUE | NUMBER | 22 | 21 | 6 | Y | 4 | 3 613 252 | 3D5036354166 | C31B1629 |
| XEDORA1A_MUDR | LP_INTERVALS | VALIDATION_STATUS | VARCHAR2 | 3 | NULL | NULL | Y | 5 | 1 | 4E56 | 4E56 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANGE_METHOD | VARCHAR2 | 3 | NULL | NULL | Y | 6 | 0 | NULL | NULL |
| XEDORA1A_MUDR | LP_INTERVALS | FAIL_CODE | NUMBER | 22 | NULL | NULL | Y | 7 | 0 | NULL | NULL |
| XEDORA1A_MUDR | LP_INTERVALS | LOCKED | CHAR | 1 | NULL | NULL | Y | 8 | 0 | NULL | NULL |
| XEDORA1A_MUDR | LP_INTERVALS | ESTIMATED | CHAR | 1 | NULL | NULL | Y | 9 | 0 | NULL | NULL |
| XEDORA1A_MUDR | LP_INTERVALS | EXT_VERSION_TIME | TIMESTAMP(0) | 7 | NULL | 0 | Y | 10 | 0 | NULL | NULL |
| XEDORA1A_MUDR | LP_INTERVALS | MEAS_TYPE_ID | NUMBER | 22 | NULL | NULL | N | 11 | 25 | C20202 | C20433 |
| XEDORA1A_MUDR | LP_INTERVALS | FLAGS | NUMBER | 22 | NULL | NULL | Y | 12 | 17 | C102 | CA053E115702551C275435 |
| XEDORA1A_MUDR | LP_INTERVALS | INTERVAL_LEN | NUMBER | 22 | NULL | NULL | N | 13 | 2 | C204 | C20A |
| XEDORA1A_MUDR | LP_INTERVALS | DEVICE_ID | NUMBER | 22 | NULL | NULL | Y | 14 | 0 | NULL | NULL |
| XEDORA1A_MUDR | LP_INTERVALS | DEVICE_DATA_SRC_ID | NUMBER | 22 | NULL | NULL | Y | 15 | 1 | C102 | C102 |
| XEDORA1A_MUDR | LP_INTERVALS | AMI_RECORD_NUM | NUMBER | 22 | NULL | NULL | Y | 16 | 0 | NULL | NULL |
| XEDORA1A_MUDR | LP_INTERVALS | ORG_ID | NUMBER | 22 | NULL | NULL | N | 17 | 1 | C135 | C135 |
| XEDORA1A_MUDR | LP_INTERVALS | LAST_UPD_BY | NUMBER | 22 | NULL | NULL | N | 18 | 1 | C137 | C137 |
| XEDORA1A_MUDR | LP_INTERVALS | LAST_UPD_TIME | TIMESTAMP(0) | 7 | NULL | 0 | N | 19 | 337 426 | 787A060F0E0E0A | 787A06180E3304 |
| XEDORA1A_MUDR | LP_INTERVALS | REC_VERSION_NUM | NUMBER | 22 | NULL | NULL | N | 20 | 2 | C102 | C103 |

# Exemple Oracle DBA_TAB_COLUMNS (2)

| OWNER | TABLE_NAME | COLUMN_NAME | DATA_TYPE | DATA_LENGTH | DATA_PRECISION | DATA_SCALE | DENSITY | NUM_NULLS | NUM_BUCKETS | LAST_ANALYZED | S |
|---|---|---|---|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | NUMBER | 22 | NULL | NULL | 0 | 0 | 254 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | INTERVAL_END_TIME | TIMESTAMP(0) | 7 | NULL | 0 | 0 | 0 | 254 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | NUMBER | 22 | 21 | 6 | 0 | 0 | 254 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | ORIG_READ_VALUE | NUMBER | 22 | 21 | 6 | 0 | 0 | 254 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | VALIDATION_STATUS | VARCHAR2 | 3 | NULL | NULL | 1 | 0 | 1 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANGE_METHOD | VARCHAR2 | 3 | NULL | NULL | 0 | 194 476 110 031 | 0 | 06-25-2022 01:08:36 | N |
| XEDORA1A_MUDR | LP_INTERVALS | FAIL_CODE | NUMBER | 22 | NULL | NULL | 0 | 194 476 110 031 | 0 | 06-25-2022 01:08:36 | N |
| XEDORA1A_MUDR | LP_INTERVALS | LOCKED | CHAR | 1 | NULL | NULL | 0 | 194 476 110 031 | 0 | 06-25-2022 01:08:36 | N |
| XEDORA1A_MUDR | LP_INTERVALS | ESTIMATED | CHAR | 1 | NULL | NULL | 0 | 194 476 110 031 | 0 | 06-25-2022 01:08:36 | N |
| XEDORA1A_MUDR | LP_INTERVALS | EXT_VERSION_TIME | TIMESTAMP(0) | 7 | NULL | 0 | 0 | 194 476 110 031 | 0 | 06-25-2022 01:08:36 | N |
| XEDORA1A_MUDR | LP_INTERVALS | MEAS_TYPE_ID | NUMBER | 22 | NULL | NULL | 0,000000000002571009878387 11 | 0 | 26 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | FLAGS | NUMBER | 22 | NULL | NULL | 0,0588235294117647 | 194 183 681 183 | 1 | 06-25-2022 01:08:36 | |
| XEDORA1A_MUDR | LP_INTERVALS | INTERVAL_LEN | NUMBER | 22 | NULL | NULL | 1 | 0 | 1 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | DEVICE_ID | NUMBER | 22 | NULL | NULL | 0 | 194 476 110 031 | 0 | 06-25-2022 01:08:36 | N |
| XEDORA1A_MUDR | LP_INTERVALS | DEVICE_DATA_SRC_ID | NUMBER | 22 | NULL | NULL | 0,000000000002571009878387 11 | 0 | 1 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | AMI_RECORD_NUM | NUMBER | 22 | NULL | NULL | 0 | 194 476 110 031 | 0 | 06-25-2022 01:08:36 | N |
| XEDORA1A_MUDR | LP_INTERVALS | ORG_ID | NUMBER | 22 | NULL | NULL | 1 | 0 | 1 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | LAST_UPD_BY | NUMBER | 22 | NULL | NULL | 1 | 0 | 1 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | LAST_UPD_TIME | TIMESTAMP(0) | 7 | NULL | 0 | 0 | 0 | 254 | 06-25-2022 01:08:36 | 1 |
| XEDORA1A_MUDR | LP_INTERVALS | REC_VERSION_NUM | NUMBER | 22 | NULL | NULL | 1 | 0 | 1 | 06-25-2022 01:08:36 | 1 |

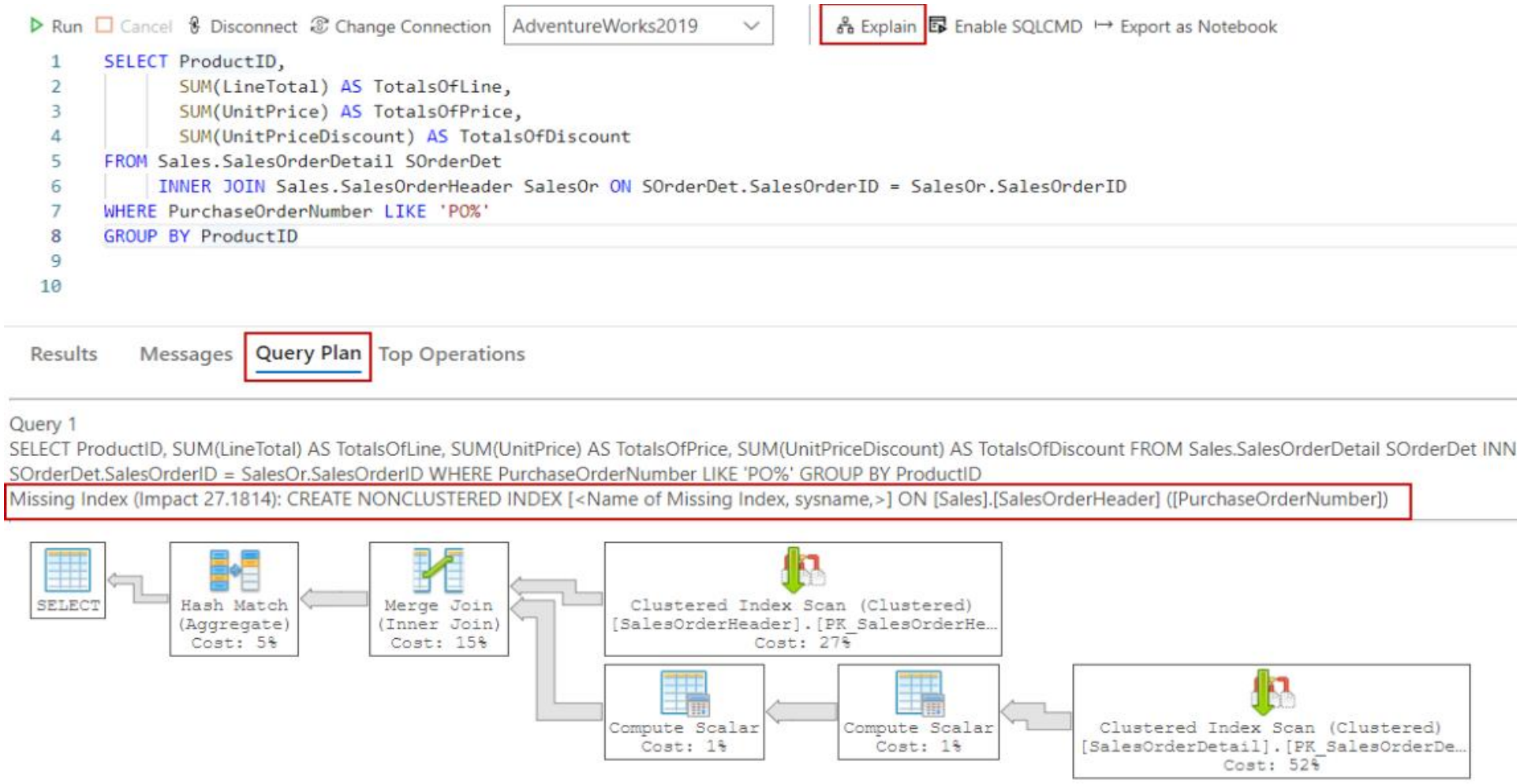
# Exemple Oracle DBA_HISTOGRAMS (1)

| OWNER | TABLE_NAME | COLUMN_NAME | ENDPOINT_NUMBER | ENDPOINT_VALUE | ENDPOINT_ACTUAL_VALUE | ENDPOINT_ACTUAL_VALUE_RAW | ENDPOINT_REPEAT_COUNT |
|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 764 477 154 | 82 000 980 | 82000980 | C453010A51 | 220448,053581346 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 781 638 644 | 82 168 358 | 82168358 | C45311543B | 512830,697612464 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 798 904 976 | 82 319 899 | 82319899 | C453206364 | 474416,151030793 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 816 275 832 | 82 501 421 | 82501421 | C453330F16 | 573706,752615719 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 833 511 447 | 82 581 215 | 82581215 | C4533B0D10 | 522635,197169594 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 850 547 925 | 82 656 471 | 82656471 | C453424148 | 678273,788299284 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 867 788 801 | 82 728 429 | 82728429 | C45349551E | 742560,301087183 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 884 806 774 | 82 823 418 | 82823418 | C453532313 | 890971,337942518 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 901 982 953 | 82 901 315 | 82901315 | C4535B0E10 | 566928,217864214 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 919 487 564 | 82 969 393 | 82969393 | C453615E5E | 598073,848872899 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 936 761 050 | 83 026 943 | 83026943 | C45403462C | 545429,334915102 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 953 741 928 | 83 086 104 | 83086104 | C454093E05 | 647 877 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 970 893 837 | 83 168 304 | 83168304 | C454115405 | 824952,800038834 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 2 988 168 929 | 83 232 282 | 83232282 | C454181753 | 734946,996194752 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 3 005 503 691 | 83 304 029 | 83304029 | C4541F291E | 477014,603298463 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 3 022 673 779 | 83 391 234 | 83391234 | C454280D23 | 711749,217180571 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 3 040 104 591 | 83 475 868 | 83475868 | C454303B45 | 653388,419427988 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 3 057 664 862 | 83 556 473 | 83556473 | C45438414A | 728623,555350608 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 3 075 080 009 | 83 602 579 | 83602579 | C4543D1A50 | 729315,275374738 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 3 092 515 758 | 83 664 374 | 83664374 | C454432C4B | 776938,092804182 |
| XEDORA1A_MUDR | LP_INTERVALS | CHANNEL_ID | 3 109 703 261 | 83 726 923 | 83726923 | C454494618 | 592 869 |

# Exemple Oracle DBA_HISTOGRAMS (2)

| OWNER | TABLE_NAME | COLUMN_NAME | ENDPOINT_NUMBER | ENDPOINT_VALUE | ENDPOINT_ACTUAL_VALUE | ENDPOINT_ACTUAL_VALUE_RAW | ENDPOINT_REPEAT_COUNT | SCOPE |
|---|---|---|---|---|---|---|---|---|
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 915 023 762 | 247 | 247,0742 | C20330082B | 628866,350045863 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 926 788 966 | 247 | 247,3 | C203301F | 178814,773267952 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 939 099 349 | 248 | 247,5378 | C20330364F | 309863,740281185 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 950 772 507 | 248 | 247,8164 | C203305241 | 257034,319409295 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 962 615 085 | 248 | 248,1132 | C203310C21 | 347 303 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 974 696 943 | 249 | 248,6328 | C20331401D | 128073,172751603 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 986 520 475 | 249 | 249,3156 | C203322039 | 68790,5057458123 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 3 998 161 234 | 251 | 250,7109 | C20333480A | 32868,6147260484 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 009 858 692 | 334 | 333,8247 | C204225330 | 6113,09739059892 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 021 502 328 | 341 | 340,9908 | C204296409 | 8200,95227516608 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 033 168 915 | 343 | 343,158 | C2042C1051 | 5828,40330144808 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 044 800 460 | 345 | 344,892 | C2042D5A15 | 5865,80675469485 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 056 492 139 | 346 | 346,1271 | C2042F0D48 | 6215,37552171247 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 068 179 394 | 347 | 347,0544 | C20430062D | 68531,6570241503 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 079 848 830 | 348 | 347,9715 | C204306210 | 5414,81378970188 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 091 652 890 | 349 | 348,75 | C204314C | 26880,0917219646 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 103 346 797 | 349 | 349,488 | C204323151 | 17293,3623816683 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 115 077 185 | 350 | 350,028 | C204330351 | 14279,2001376141 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 126 771 995 | 351 | 350,5686 | C204333957 | 5735,36625134393 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 138 401 013 | 351 | 351,1005 | C204340B06 | 6100,92333577483 | SHARED |
| XEDORA1A_MUDR | LP_INTERVALS | LP_VALUE | 4 150 120 615 | 352 | 351,7008 | C204344709 | 24577,0225668692 | SHARED |

# Plan d’exécution SQL Server



# Plan et statistiques adaptives

```
-------------------------------------------------------------------------------------------------------
|Id| Operation                | Name           |Start|E-Rows|A-Rows|A-Time|Buff|OMem|1Mem|O/1/M|
-------------------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT         |                |   1|      | 269|00:00:00.09|1338|    |    |     |
| 1|  NESTED LOOPS            |                |   1|  1   | 269|00:00:00.09|1338|    |    |     |
| 2|   MERGE JOIN CARTESIAN|                |   1|  4   |9135|00:00:00.03|  33|    |    |     |
|*3|    TABLE ACCESS FULL  |PRODUCT_INFORMAT|   1|  1   |  87|00:00:00.01|  32|    |    |     |
| 4|    BUFFER SORT           |                |  87|105   |9135|00:00:00.01|   1|4096|4096|1/0/0|
| 5|     INDEX FULL SCAN   | ORDER_PK       |   1|105   | 105|00:00:00.01|   1|    |    |     |
|*6|   INDEX UNIQUE SCAN   | ORDER_ITEMS_UK |9135|  1   | 269|00:00:00.03|1305|    |    |     |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(("MIN_PRICE"<40 AND "LIST_PRICE"<50))
   6 - access("O"."ORDER_ID"="ORDER_ID" AND "P"."PRODUCT_ID"="O"."PRODUCT_ID")
```

# Plan d’exécution parallélisme

```
PLAN_TABLE_OUTPUT
------------------------------------
SQL_ID    1q2m84vq594pj, child number 0
------------------------------------
delete /*+ PARALLEL(a 42) */ from a where object_id in (select object_id from b)
Plan hash value: 1475376193
```

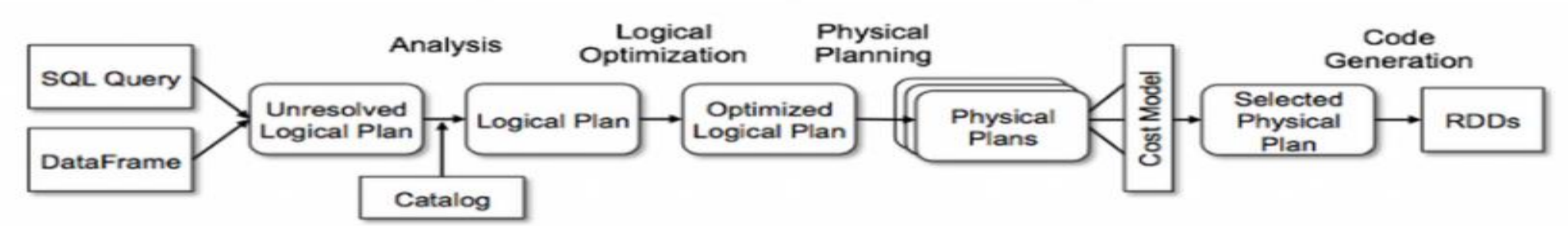
| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time | TQ | IN-OUT | PQ Distrib |
|---|---|---|---|---|---|---|---|---|---|
| 0 | DELETE STATEMENT | | | | 855 (100) | | | | |
| 1 | DELETE | A | | | | | | | |
| 2 | PX COORDINATOR | | | | | | | | |
| 3 | PX SEND QC (RANDOM) | :TQ10002 | 91277 | 891K | 855 (1) | 00:00:01 | Q1,02 | P->S | QC(RAND) |
| * 4 | HASH JOIN SEMI BUFFERED | | 91277 | 891K | 855 (1) | 00:00:01 | Q1,02 | PCWP | |
| 5 | PX RECEIVE | | 91277 | 445K | 427 (1) | 00:00:01 | Q1,02 | PCWP | |
| 6 | PX SEND HASH | :TQ10000 | 91277 | 445K | 427 (1) | 00:00:01 | Q1,00 | P->P | HASH |
| 7 | PX BLOCK ITERATOR | | 91277 | 445K | 427 (1) | 00:00:01 | Q1,00 | PCWC | |
| * 8 | TABLE ACCESS FULL | A | 91277 | 445K | 427 (1) | 00:00:01 | Q1,00 | PCWP | |
| 9 | PX RECEIVE | | 91278 | 445K | 427 (1) | 00:00:01 | Q1,02 | PCWP | |
| 10 | PX SEND HASH | :TQ10001 | 91278 | 445K | 427 (1) | 00:00:01 | Q1,01 | S->P | HASH |
| 11 | PX SELECTOR | | | | | | Q1,01 | SCWC | |
| 12 | TABLE ACCESS FULL | B | 91278 | 445K | 427 (1) | 00:00:01 | Q1,01 | SCWP | |

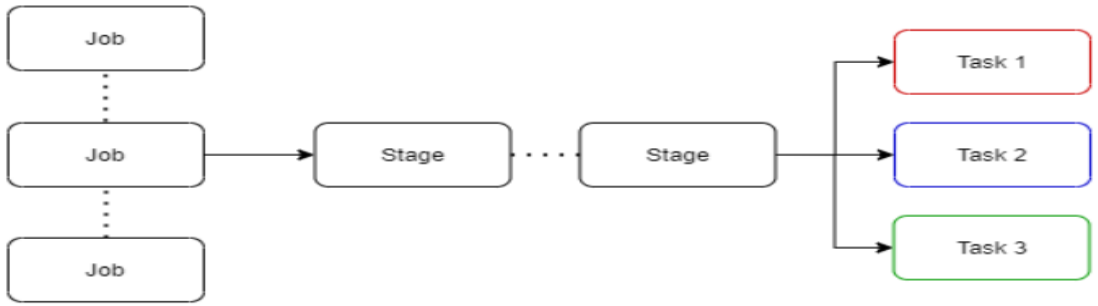
# Plan d’exécution inter-nœuds - Spark

**Task** – opération unitaire sur une seule partition

**Stage** – séquence qui peut être exécutée en parallèle

**Job** – séquence de stages et taches



- The Optimizer in Oracle Database 19c
- Indexation automatique
- Algorithmes adaptives de couts

# AI ML dans la planification

- The Optimizer in Oracle Database 19c
- Indexation automatique
- Algorithmes adaptives de couts