TP : TRAITEMENT DE REQUETES

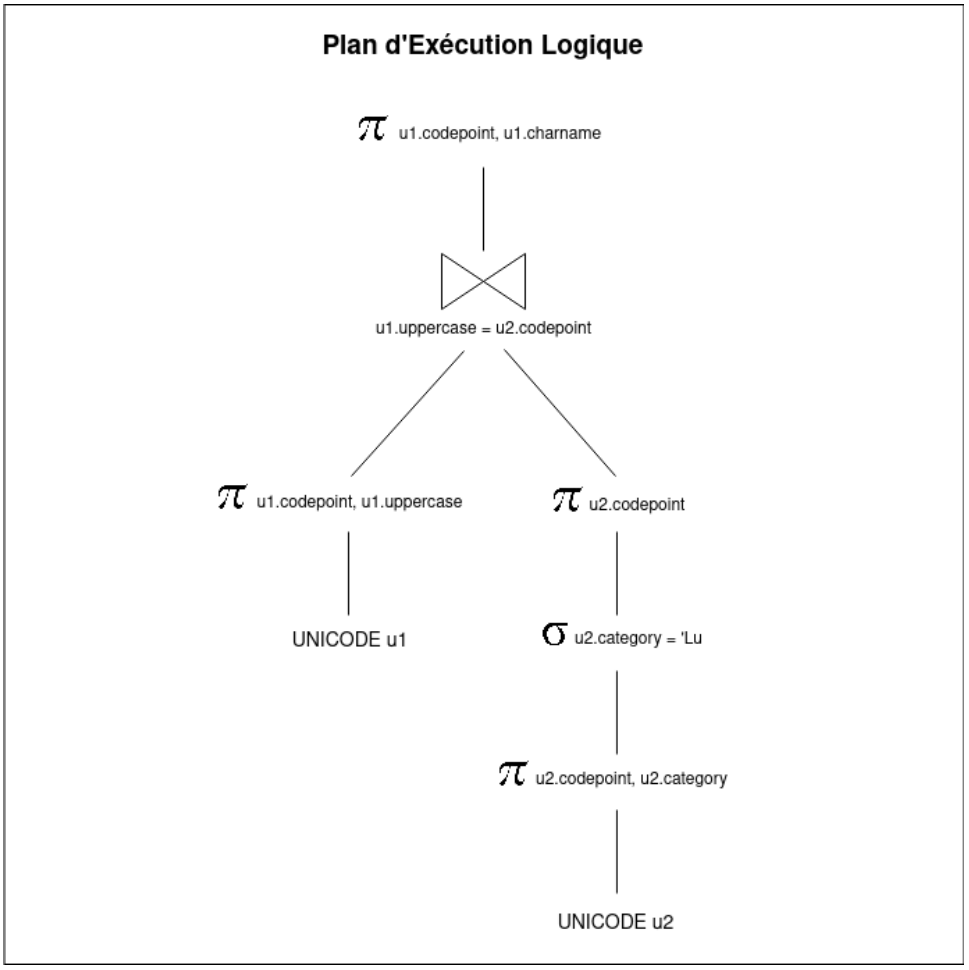
Partie 1 : Prise en main

Découverte du plan d'exécution

Question 1 : Exécuter la requête Q0 et donner la taille du résultat (en n-uplets) et le temps d'exécution.

```
18 ms et 1331 n-uplets
```

Question 2 : Traduire la requête Q0 en une expression algébrique sous forme arborescente, c'est-à-dire,
dessiner son Plan d'Exécution Logique (PEL).



Question 3 : Reporter et analyser les informations délivrées par l'examen du plan d'exécution de la requête Q0. Établir l'ordre de traitement des opérations dans le plan. Déterminer les pro- jections réalisées tout au long de l'arbre d'exécution.

```
PLAN_TABLE_OUTPUT
Plan hash value: 2187253078

-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |  1135 | 90800 |   244   (1)| 00:00:01 |
|*  1 |  HASH JOIN         |         |  1135 | 90800 |   244   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| UNICODE |  1114 | 16710 |   122   (1)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| UNICODE |  1400 | 91000 |   122   (1)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("U2"."CODEPOINT"="U1"."UPPERCASE")
   2 - filter("U2"."CATEGORY_"=U'Lu')
   3 - filter("U1"."UPPERCASE" IS NOT NULL)
```

`SELECT STATEMENT : Pour selectionner codepoint et charname.`

`HASH JOIN : Correspond à l'autojointure u1 u2.`

`TABLE ACCESS FULL : Pour acceder aux tables lors de la jointure`.

La requête Q0 commence par accéder à la table unicode en sélectionnant des uppercases non nuls puis accède une deuxième fois à la table en sélectionnant aussi la catégorie LU. Ensuite la requête fait une jointure par hachage et finit par projeter sur coderont et charname.

Question 4 : Comparer et commenter le plan de Q0 avec celui des requêtes suivantes :

1. (a) Q0 réécrite avec une clause

exists ;

```
select u1.charname, u1.codepoint from unicode u1
where exists (select 1 from unicode u2 where u2.codepoint = u
and u2.CATEGORY_= 'LU' );
```

Le PEL est le même que pour la requête Q0 mais on remarque une diminution du nombre de lignes et du nombre de bytes

2. (b) Q0 réécrite avec une clause **in** ;

```
select u1.charname, u1.codepoint from unicode u1
where u1.uppercase
in (select u2.CODEPOINT from unicode u2 where u2.CATEGORY_= '
```

Le PEL est le même.

3. (c) Q0 augmentée d'une tautologie « triviale » : A>0 **or** A<=0 ;

```
select u1.codepoint, u1.charname from unicode u1 join unicode
on u2.CODEPOINT = u1.UPPERCASE
where u2.CATEGORY_ = 'LU' and (u1.COMBINING > 0 or u1.COMBINI
```

Le SGBD peut simplifier la condition et optimiser le plan d'exécution. Mais ce n'est pas le cas ici.

4. (d) Q0 augmentée d'une tautologie non triviale : (¬p ∧ (p ∨ q)) → q, où p et q sont des clauses de type A>x.

```
select u1.codepoint, u1.charname from unicode u1
join unicode u2 on u2.CODEPOINT = u1.UPPERCASE
where u2.CATEGORY_ = 'LU' and (NOT(NOT(u1.COMBINING) >= 0
and (u1.COMBINING >= 0 or u1.DIGIT >= 0)) or u1.DIGIT >= 0);
```

Le plan a changé pour cette dernière requête.

Les Statistiques

Question 6 : En examinant les statistiques de la table unicode, reporter les principaux indicateurs maintenus par le système.

```
BEGIN
    dbms_stats.delete_table_stats('E219118X', 'UNICODE');
    dbms_stats.gather_table_stats('E219118X', 'UNICODE');
END;

select * from user_tab_statistics us where us.table_name = 'U
select * from user_tab_col_statistics uc where uc.table_name
```

USER_TAB_STATISTICS :

- NUM_ROWS: Le nombre total de lignes dans la table.
- BLOCKS: Le nombre de blocs alloués pour stocker les données de la table.
- EMPTY_BLOCKS: Le nombre de blocs vides dans la table.
- AVG_SPACE: L'espace moyen utilisé par une ligne dans la table.
- CHAIN_CNT: Le nombre de chaînes de migration, qui indique le nombre de blocs nécessaires pour stocker une ligne à la suite d'une autre.
- AVG_ROW_LEN: La longueur moyenne d'une ligne en octets.

USER_TAB_COL_STATISTICS :

- NUM_DISTINCT: Le nombre de valeurs distinctes dans la colonne.
- LOW_VALUE et HIGH_VALUE: Les valeurs minimale et maximale de la colonne.
- DENSITY: La densité, qui est le nombre moyen de valeurs distinctes par bloc.
- NUM_NULLS: Le nombre de valeurs nulles dans la colonne.
- NUM_BUCKETS: Le nombre de compartiments utilisés pour l'histogramme.
- SAMPLE_SIZE: La taille de l'échantillon utilisée pour collecter les statistiques.

Question 7 : Rafraîchir les statistiques (*gather table stats*) puis déterminer les colonnes de la table unicode sur lesquelles le système a construit un histogramme. Pour quelle(s) raison(s) ?

Le système a construit un histogramme sur les colonnes category, combining et digit en raison de la distribution qui n'est pas uniforme.

Question 8 : Effacer les statistiques de la table unicode. Puis, recalculer le plan de la requête Q0. Reporter et commenter les différences avec la version précédente du plan d'exécution.

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |  1163 |   149K|   244   (1)| 00:00:01 |
|*  1 |  HASH JOIN         |         |  1163 |   149K|   244   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| UNICODE |  1944 | 27216 |   122   (1)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| UNICODE | 29747 |  3427K|   122   (1)| 00:00:01 |
-------------------------------------------------------------------------------
```

La principale différence avec la version précédente est l'augmentation de de l'utilisation de la mémoire et du nombre de lignes.

Les index

Question 10 : Donner les propriétés principales de l'index arbre B+ sur la clé primaire de la table unicode, dont sa taille, en consultant les tables du dictionnaire user_indexes et user_segments. Reporter les requêtes SQL requises et leurs résultats.

```
-- Obtenir les propriétés de l'index à partir de user_indexes
SELECT *
FROM user_indexes
WHERE table_name = 'UNICODE';
```

```
-- Obtenir les propriétés de la taille à partir de user_segme
SELECT *
FROM user_segments where segment_name = 'UNICODE';
```

INI_TRANS	MAX_TRANS	INITIAL_EXTENT	NEXT_EXTENT	MIN_EXTENTS	MAX_EXTENTS	PCT_INCREASE
2	255	65536	1048576	1	2147483645	<null>
2	255	65536	1048576	1	2147483645	<null>

	BYTES	BLOCKS	EXTENTS	INITIAL_EXTENT	NEXT_EXTENT	MIN_EXTENTS	MAX_EXTENTS	MAX_SIZE
1	4194304	512	19	65536	1048576	1	2147483645	2147483645

Question 11 : Déclarer une contrainte d'unicité sur la colonne oldname, puis observer et reporter la création automatique d'un « index unique » par le système. Commenter.

L'index sur old_name est plus performant car son clustering_factor est moins élevé que l'index sur la clé primaire. Sachant que le clustering_factor correspond au nombre i/o pour lire entièrement une table via un single range scan.

Question 12 : Construire un index couvrant composite pour la requête Q0. Donner sa taille. Vérifier sa pertinence en réévaluant et en analysant son plan d'exécution.

```
PLAN_TABLE_OUTPUT
3  ------------------------------------------------------------------------------------------
4  | Id  | Operation            | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
5  ------------------------------------------------------------------------------------------
6  |   0 | SELECT STATEMENT     |                     |  1163 |   149K|   207   (1)| 00:00:01 |
7  |*  1 |  HASH JOIN           |                     |  1163 |   149K|   207   (1)| 00:00:01 |
8  |*  2 |   INDEX FAST FULL SCAN| IDX_UNICODE_COUVRANT |  1944 | 27216 |   104   (1)| 00:00:01 |
9  |   3 |   INDEX FAST FULL SCAN| IDX_UNICODE_COUVRANT | 29747 |  3427K|   103   (0)| 00:00:01 |
10 ------------------------------------------------------------------------------------------
```

On voit qu'il y a une augmentation de l'utilisation mémoire et du nombre de lignes. On voit la taille qui est de 3427K bytes.

Question 13 : Reproduire l'index couvrant composite pour Q0, mais cette fois-ci à l'aide d'un index plaçant (*cluster*), par création d'une réplique unicode2 de la table unicode.

```
CREATE table unicode2 (
    codepoint NVARCHAR2(6) PRIMARY KEY,
    charname NVARCHAR2(100),
    uppercase NVARCHAR2(6),
    category_ NCHAR(2),
    FOREIGN KEY (codepoint) REFERENCES UNICODE(CODEPOINT)
)

ORGANIZATION INDEX
INCLUDING category_ overflow;

insert into unicode2 (codepoint, charname, uppercase, categor
SELECT codepoint, charname, uppercase, category_ from unicode

BEGIN
    dbms_stats.gather_table_stats('E219118X', 'UNICODE');
end;

-- VERIF DE LA CREATION --
select * from user_indexes where table_name='UNICODE2';
```

Question 14 : Reporter et commenter le plan d'exécution de la requête Q0 adaptée pour fonctionner sur la table unicode2.

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |    24 |  3168 |  6125   (1)| 00:00:01 |
|*  1 |  HASH JOIN            |                     |    24 |  3168 |  6125   (1)| 00:00:01 |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_130367  |     4 |    56 |  3063   (1)| 00:00:01 |
|   3 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_130367  | 31890 |  3674K|  3062   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Le Pel est similaire on voit qu'il y a la le même nombre de lignes, le même temps d'exécution.

Question 15 : Comparer les deux approches : index couvrant et index cluster.

`L'index cluster est plus efficace que l'index couvrant composite malgré que les 2 soient plus performant que  l'index classique, l'index cluster va analyser moins de butes ce qui va lui perlettre de  gagner du temps.`

Partie 2 : Analyse du plan d'exécution

Question 1 : **Recherche** : les noms d'usage triés des caractères commençant par un préfixe donné (charname like ? || '%'), et appartenant à une catégorie donnée (category_ = ?).

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |    24 |  3168 |  6125   (1)| 00:00:01 |
|*  1 |  HASH JOIN            |                     |    24 |  3168 |  6125   (1)| 00:00:01 |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_130367  |     4 |    56 |  3063   (1)| 00:00:01 |
|   3 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_130367  | 31890 |  3674K|  3062   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Le système fait d'abord un Fullscan en utilisant un filtre sur charname et category_ et ne récupère que l'attribut charname. Ensuite, il attend tous les tuples pour les trier selon charname.

Question 2 : **Quantification** : le nombre de caractères d'un type bidirectionnel donné (bidi = ?).

```
-----------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |    24 |  3168 |  6125   (1)| 00:00:01 |
|*  1 |  HASH JOIN            |                    |    24 |  3168 |  6125   (1)| 00:00:01 |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_130367 |     4 |    56 |  3063   (1)| 00:00:01 |
|   3 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_130367 | 31890 |  3674K|  3062   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Le système fait un Fullscan avec un filtre sur bidi. Ensuite il fait un count sur le nombre de tuples retourner.

Question 3 : **Information détaillée** : toute l'information à propos d'un caractère choisi selon son code (codepoint = ?). Les variations de casse en minuscule (lowercase), en majuscule (uppercase) et de titre (titlecase) sont présentées par leur nom d'usage (charname).

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |     1 |   280 |     5   (0)| 00:00:01 |
|   1 |  NESTED LOOPS OUTER           |                     |     1 |   280 |     5   (0)| 00:00:01 |
|   2 |   NESTED LOOPS OUTER          |                     |     1 |   217 |     4   (0)| 00:00:01 |
|   3 |    NESTED LOOPS OUTER         |                     |     1 |   154 |     3   (0)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| UNICODE             |     1 |    91 |     2   (0)| 00:00:01 |
|*  5 |      INDEX UNIQUE SCAN        | PK_UNICODE          |     1 |       |     1   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN          | IDX_UNICODE_COUVRANT |     1 |    63 |     1   (0)| 00:00:01 |
|*  7 |    INDEX RANGE SCAN           | IDX_UNICODE_COUVRANT |     1 |    63 |     1   (0)| 00:00:01 |
|*  8 |   INDEX RANGE SCAN            | IDX_UNICODE_COUVRANT |     1 |    63 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Question 4 : **Statistique** : la longueur moyenne des noms d'usage (charname) des caractères dont l'ancien nom de la casse majuscule(uppercase) se conforme à un motif quelconque(oldname like ?).

```
-----------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |    70 |   221   (1)| 00:00:01 |
|   1 |  SORT AGGREGATE       |                     |     1 |    70 |            |          |
|*  2 |   HASH JOIN           |                     |   101 |  7070 |   221   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL  | UNICODE             |    99 |  1485 |   122   (1)| 00:00:01 |
|*  4 |    INDEX FAST FULL SCAN| IDX_UNICODE_COUVRANT |  1400 | 77000 |    99   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Partie 3 : Les opérateurs

Question 1.a : INDEX RANGE SCAN : recherche d'une plage de *rowid's* dans l'index.

```
explain plan for select * from unicode where codepoint betwee
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |            |    10 |   910 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| UNICODE    |    10 |   910 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | PK_UNICODE |    10 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Question 1.b : INDEX SKIP SCAN : recherche de *rowid's* par des clés secondaires d'un index composite.

```
create index idx_skip on unicode(category_, charname, combini

explain plan for
select u1.CATEGORY_, u1.CHARNAME, u1.COMBINING from unicode u
where 'INVERTED QUESTION MARK' = u1.CHARNAME;

select * from table (DBMS_XPLAN.DISPLAY('PLAN_TABLE'));
```

```
-------------------------------------------------------------------------------
| Id  | Operation        | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |          |     1 |    61 |    31   (0)| 00:00:01 |
|*  1 |  INDEX SKIP SCAN | IDX_SKIP |     1 |    61 |    31   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Question 2.a : NESTED LOOPS : jointure par double boucle imbriquée.

```
explain plan for
select count(codepoint)
from unicode
where bidi = 'ON';
```

```
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |    60 | 10920 |   182   (1)| 00:00:01 |
|   1 |  NESTED LOOPS                |            |    60 | 10920 |   182   (1)| 00:00:01 |
|   2 |   NESTED LOOPS               |            |    60 | 10920 |   182   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | UNICODE    |    60 |  5460 |   122   (1)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_UNICODE |     1 |       |     0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| UNICODE    |     1 |    91 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Question 2.b : MERGE JOIN : jointure par tri-fusion.

```
explain plan for
select /*+ use_merge(u1,u2) */ u1.UPPERCASE, u2.LOWERCASE
                  from unicode u1 join unicode u2 on u1.UPPERC
```

```
-----------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |  1409 |  5636 |   223   (2)| 00:00:01 |
|   1 |  MERGE JOIN           |                     |  1409 |  5636 |   223   (2)| 00:00:01 |
|   2 |   SORT JOIN           |                     |  1383 |  2766 |   123   (2)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL  | UNICODE             |  1383 |  2766 |   122   (1)| 00:00:01 |
|*  4 |   SORT JOIN           |                     |  1400 |  2800 |   100   (1)| 00:00:01 |
|*  5 |    INDEX FAST FULL SCAN| IDX_UNICODE_COUVRANT |  1400 |  2800 |    99   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Question 3 :

Proposer une requête montrant dans son plan d'exécution une « triple jointure » (R I S I
T I U) avec, pour chaque jointure, un algorithme différent4.

```
SELECT /*+ use_merge(u1,u2) */ *
FROM unicode u1
JOIN unicode u2 ON u1.codepoint = u2.uppercase
JOIN unicode u3 ON u1.lowercase = u3.codepoint
JOIN unicode u4 ON u2.codepoint = u4.lowercase;
```

```
-------------------------------------------------------------------------------
| Id  | Operation             | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |         |  1383 |   491K|   490   (2)| 00:00:01 |
|*  1 |  HASH JOIN            |         |  1383 |   491K|   490   (2)| 00:00:01 |
|   2 |   MERGE JOIN          |         |  1383 |   368K|   368   (2)| 00:00:01 |
|   3 |    SORT JOIN          |         |  1383 |   245K|   245   (2)| 00:00:01 |
|*  4 |     HASH JOIN         |         |  1383 |   245K|   244   (1)| 00:00:01 |
|*  5 |      TABLE ACCESS FULL| UNICODE |  1400 |   124K|   122   (1)| 00:00:01 |
|*  6 |      TABLE ACCESS FULL| UNICODE |  1383 |   122K|   122   (1)| 00:00:01 |
|*  7 |    SORT JOIN          |         |  1383 |   122K|   123   (2)| 00:00:01 |
|*  8 |     TABLE ACCESS FULL | UNICODE |  1383 |   122K|   122   (1)| 00:00:01 |
|   9 |   TABLE ACCESS FULL   | UNICODE | 32292 |  2869K|   122   (1)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("U1"."LOWERCASE"="U3"."CODEPOINT")
   4 - access("U2"."CODEPOINT"="U4"."LOWERCASE")
   5 - filter("U2"."UPPERCASE" IS NOT NULL)
   6 - filter("U4"."LOWERCASE" IS NOT NULL)
   7 - access("U1"."CODEPOINT"="U2"."UPPERCASE")
       filter("U1"."CODEPOINT"="U2"."UPPERCASE")
   8 - filter("U1"."LOWERCASE" IS NOT NULL)

Hint Report (identified by operation id / Query Block Name / Object Alias):
Total hints for statement: 1 (U - Unused (1))
```

Question 4.a : SORT UNIQUE : élimination des doublons.

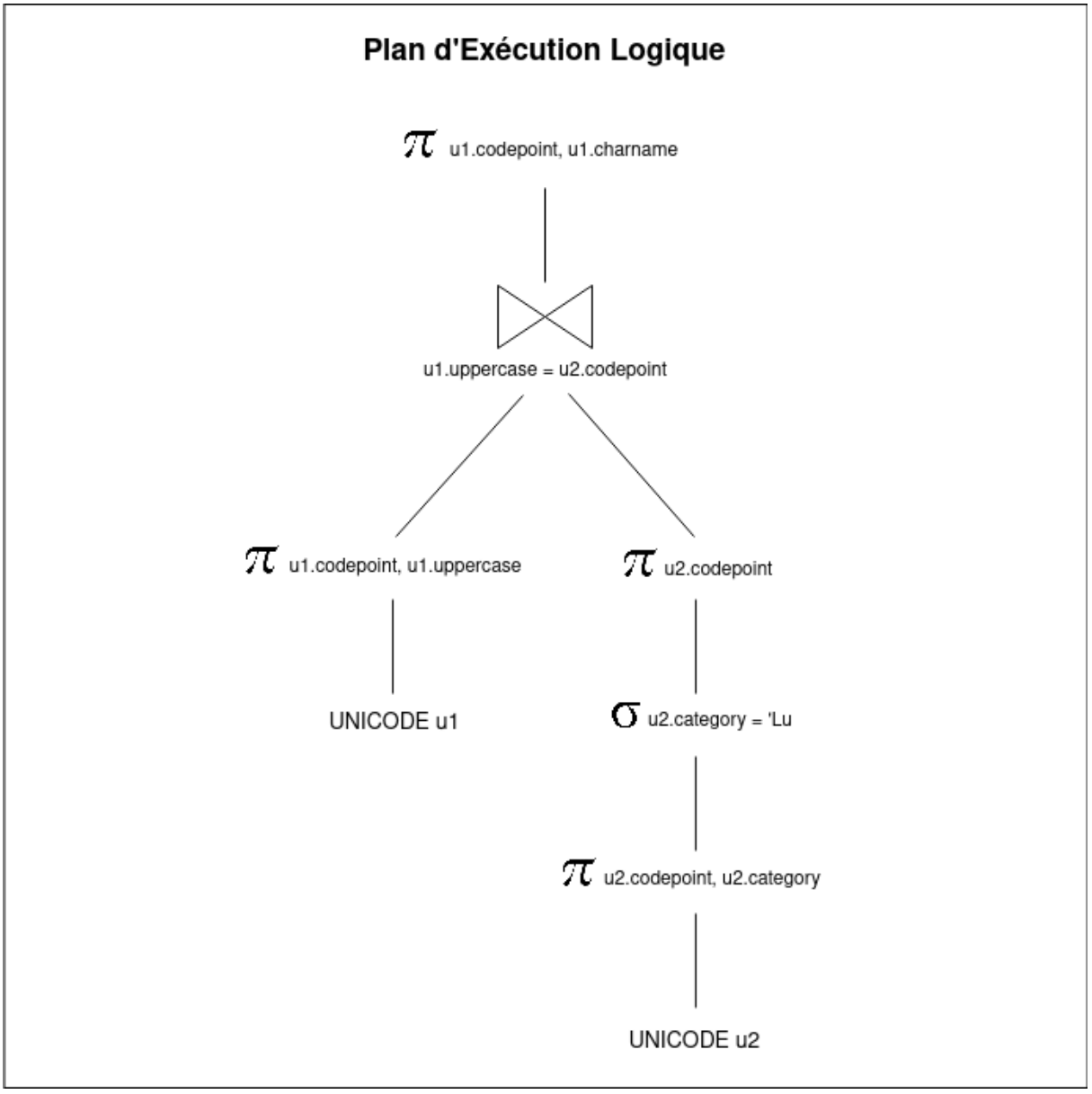
```
select distinct u1.DIGIT from unicode u1 order by u1.DIGIT;
```

```
--------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |    10 |    20 |   126   (4)| 00:00:01 |
|   1 |  SORT UNIQUE       |         |    10 |    20 |   124   (3)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| UNICODE | 32292 | 64584 |   122   (1)| 00:00:01 |
--------------------------------------------------------------------------------
```

Question 4.b : HASH GROUP BY : préparation des enregistrement pour un GROUP BY.

```
explain plan for
SELECT category_
FROM unicode
GROUP BY category_;
```

```
--------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |    29 |   145 |   124   (3)| 00:00:01 |
|   1 |  HASH GROUP BY     |         |    29 |   145 |   124   (3)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| UNICODE | 32292 |   157K|   122   (1)| 00:00:01 |
--------------------------------------------------------------------------------
```


Plan d'Exécution Logique
π u1.codepoint, u1.charname
u1.uppercase = u2.codepoint
π u1.codepoint, u1.uppercase
π u2.codepoint
UNICODE u1
σ u2.category = 'Lu
π u2.codepoint, u2.category
UNICODE u2