

COMPTE-RENDU TP2 BDA

Exercise 1:

1/

The screenshot shows the SQL Developer interface with the following details:

- SQL Commands** tab: The query is `SELECT dept.dept_name
from department dept
where dept.budget = (SELECT MAX(budget) from department);`
- Schema**: WKSP_CDZLTGASL2ROCA
- Language**: SQL
- Rows**: 10
- Buttons**: Clear Command, Find Tables, Save, Run
- Results** tab: The result is a single row with the value **Finance** under the column **DEPT_NAME**.
- Footer**: 1 rows returned in 0.01 seconds, Download

2/

The screenshot shows the SQL Developer interface with the following details:

- SQL Commands** tab: The query is `SELECT tea.name,tea.salary from teacher tea
WHERE salary > (SELECT AVG(tea.salary) FROM teacher tea);`
- Schema**: WKSP_CDZLTGASL2ROCA
- Language**: SQL
- Rows**: 10
- Buttons**: Clear Command, Find Tables, Save, Run
- Results** tab: The result is a table with 7 rows and 2 columns: **NAME** and **SALARY**.
- Footer**: 7 rows returned in 0.01 seconds, Download

NAME	SALARY
Wu	90000
Einstein	95000
Gold	87000
Katz	75000
Singh	80000
Brandt	92000
Kim	80000

3/

APEX App Builder SQL Workshop Team Development Gallery Search ES

SQL Commands Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

```

1 SELECT tea.id as professor_id,tea.name as professor_name,
2 stu.id as student_id, stu.name as student_name,
3 COUNT(*) as total_courses FROM teacher tea, student stu, takes ts, teaches teas
4 WHERE tea.id = teas.id
5 and teas.course_id = ts.course_id
6 and ts.sec_id = teas.sec_id
7 and ts.semester = teas.semester
8 and ts.year = teas.year
9 and stu.id = ts.id
10 group by stu.id, stu.name, tea.id, tea.name
11 HAVING COUNT(*) > 2;

```

Results Explain Describe Saved SQL History

PROFESSOR_ID	PROFESSOR_NAME	STUDENT_ID	STUDENT_NAME	TOTAL_COURSES
10101	Srinivasan	12345	Shankar	3

rows returned in 0.05 seconds Download

4/

APEX App Builder SQL Workshop Team Development Gallery Search ES

SQL Commands Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

```

1 SELECT tea.id as professor_id,tea.name as professor_name,
2 stu.id as student_id, stu.name as student_name, total_courses
3 from (
4     SELECT teas.id as teacher_id,
5           ts.id as student_id,
6           COUNT(*) as total_courses
7     from teaches teas, takes ts
8     WHERE teas.course_id = ts.course_id
9           and teas.sec_id = ts.sec_id
10           and teas.semester = ts.semester
11           and teas.year = ts.year
12     group by teas.id, ts.id
13 ) number_of_courses, teacher tea, student stu
14 WHERE tea.id = number_of_courses.teacher_id
15 and stu.id = number_of_courses.student_id
16 and number_of_courses.total_courses > 2;

```

Results Explain Describe Saved SQL History

PROFESSOR_ID	PROFESSOR_NAME	STUDENT_ID	STUDENT_NAME	TOTAL_COURSES
10101	Srinivasan	12345	Shankar	3

rows returned in 0.03 seconds Download

5/

APEX

App Builder

SQL Workshop

Team Development

Gallery

Search

ES

SQL Commands

SchemaWKSP_CDZLTGASL2ROCA

LanguageSQL

Rows10

Clear Command

Find Tables

Save

Run

Az

```
1 SELECT stu.id, stu.name from student stu WHERE
2 stu.id not in (
3 | SELECT distinct ts.id from takes ts WHERE ts.year < 2010
4 )
```

Results

Explain

Describe

Saved SQL

History

ID	NAME
19991	Brandt
23121	Chavez
55739	Sanchez
70557	Snow

4 rows returned in 0.01 secondsDownload

6/

APEX

App Builder

SQL Workshop

Team Development

Gallery

Search

ES

SQL Commands

SchemaWKSP_CDZLTGASL2ROCA

LanguageSQL

Rows10

Clear Command

Find Tables

Save

Run

Az

```
1 SELECT *
2 from teacher tea WHERE tea.name LIKE 'E%';
```

Results

Explain

Describe

Saved SQL

History

ID	NAME	DEPT_NAME	SALARY
22222	Einstein	Physics	95000
32343	El Said	History	60000

2 rows returned in 0.01 secondsDownload

7/

APEX App Builder SQL Workshop Team Development Gallery Search

SQL Commands Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

```

1 SELECT *
2 FROM teacher tea1
3 WHERE 3 = (
4     SELECT COUNT(DISTINCT tea2.salary)
5     FROM teacher tea2
6     WHERE tea2.salary > tea1.salary
7 );
8

```

Results Explain Describe Saved SQL History

ID	NAME	DEPT_NAME	SALARY
33456	Gold	Physics	87000

1 rows returned in 0.01 seconds Download

8/

APEX App Builder SQL Workshop Team Development Gallery Search

SQL Commands Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

```

1 SELECT tea.id, tea.name, tea.salary
2 from (
3     SELECT tea.id, tea.name, tea.salary
4     from teacher tea
5     order by tea.salary asc
6 ) tea
7 where rownum <= 3;

```

Results Explain Describe Saved SQL History

ID	NAME	SALARY
15151	Mozart	40000
32343	El Said	60000
58583	Califieri	62000

3 rows returned in 0.00 seconds Download

9/

SQL Commands

Schema: WKSP_CDZLTGASL2ROCA

Language: SQL Rows: 10

Clear Command Find Tables Save Run

```
1 SELECT stu.name
2 FROM student stu
3 WHERE stu.id IN (
4     SELECT ts.id
5     from takes ts WHERE
6     ts.semester = 'Fall' and ts.year = 2009
7 );
```

Results Explain Describe Saved SQL History

NAME
Zhang
Shankar
Peltier
Levy
Williams
Brown
Bourikas

7 rows returned in 0.02 seconds Download

10/

SQL Commands

Schema: WKSP_CDZLTGASL2ROCA

Language: SQL Rows: 10

Clear Command Find Tables Save Run

```
1 SELECT stu.name
2 FROM student stu
3 WHERE stu.id = SOME(
4     SELECT ts.id
5     from takes ts WHERE
6     ts.semester = 'Fall' and ts.year = 2009
7 );
```

Results Explain Describe Saved SQL History

NAME
Zhang
Shankar
Peltier
Levy
Williams
Brown
Bourikas

7 rows returned in 0.02 seconds Download

11/

SQL Commands Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

```

1 SELECT distinct stu.name
2 from student stu
3 NATURAL INNER JOIN takes ts
4 WHERE ts.semester = 'Fall' and ts.year = 2009

```

Results Explain Describe Saved SQL History

NAME
Brown
Zhang
Levy
Bourikas
Shankar
Peltier
Williams

7 rows returned in 0.02 seconds Download

12/

SQL Commands Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

```

1 SELECT stu.name
2 from student stu
3 WHERE EXISTS (
4     SELECT *
5     from takes ts
6     WHERE ts.semester = 'Fall'
7           and ts.year = 2009
8           and ts.id = stu.id
9 );

```

Results Explain Describe Saved SQL History

NAME
Zhang
Shankar
Peltier
Levy
Williams
Brown
Bourikas

7 rows returned in 0.01 seconds Download

13/

SQL Commands

SchemaWKSP_CDZLTGASL2ROCA

LanguageSQLRows10Clear CommandFind TablesSaveRun

↶↷🔍🔗A⌘⚙️

```
1 SELECT stu1.name as student1, stu2.name AS student2
2 from student stu1, student stu2, takes ts_stu1, takes ts_stu2
3 WHERE stu1.id >| stu2.id
4 and ts_stu1.course_id = ts_stu2.course_id
5 and ts_stu1.sec_id = ts_stu2.sec_id
6 and ts_stu1.semester = ts_stu2.semester
7 and ts_stu1.year = ts_stu2.year
8 and ts_stu1.id = stu1.id
9 and ts_stu2.id = stu2.id;
10
```

Results

ExplainDescribeSaved SQLHistory

STUDENT1	STUDENT2
Shankar	Zhang
Shankar	Zhang
Levy	Shankar
Levy	Zhang
Williams	Levy
Williams	Shankar
Williams	Zhang
Williams	Shankar
Brown	Williams
Brown	Levy

More than 10 rows available. Increase rows selector to view more rows.

10 rows returned in 0.02 secondsDownload

14/

SQL Commands

SchemaWKSP_CDZLTGASL2ROCA

LanguageSQLRows10Clear CommandFind TablesSaveRun

↶↷🔍🔗A⌘⚙️

```
1 SELECT tea.name, COUNT(*) AS number_of_students
2 from takes ts,teaches teas, teacher tea
3 WHERE ts.course_id = teas.course_id
4 and ts.sec_id = teas.sec_id
5 and ts.semester = teas.semester
6 and ts.year = teas.year
7 and teas.id = tea.id
8 GROUP BY tea.name, tea.id
9 ORDER BY number_of_students DESC;
```

Results

ExplainDescribeSaved SQLHistory

NAME	NUMBER_OF_STUDENTS
Srinivasan	10
Brandt	3
Crick	2
Katz	2
Einstein	1
El Said	1
Mozart	1
Wu	1
Kim	1

9 rows returned in 0.06 secondsDownload

15/

↑ SQL Commands

Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

↶ ↷ 🔍 ↵ A:: ⚙️

```
1 SELECT tea.name, COUNT(ts.id) as number_of_students
2 FROM teacher tea
3 LEFT JOIN teaches teas on tea.id = teas.id
4 LEFT JOIN takes ts on ts.course_id = teas.course_id
5 and ts.sec_id = teas.sec_id
6 and ts.semester = teas.semester
7 and ts.year = teas.year
8 group by tea.name, tea.id
9 order by number_of_students desc;
```

Results Explain Describe Saved SQL History

NAME	NUMBER_OF_STUDENTS
Srinivasan	10
Brandt	3
Katz	2
Crick	2
Wu	1
El Said	1
Mozart	1
Einstein	1
Kim	1
Singh	0

More than 10 rows available. Increase rows selector to view more rows.

10 rows returned in 0.02 seconds Download

16/

↑ SQL Commands

Schema WKSP_CDZLTGASL2ROCA

Language SQL Rows 10 Clear Command Find Tables Save Run

↶ ↷ 🔍 ↵ A:: ⚙️

```
1 SELECT tea.name, COUNT(*) AS number_of_A_grades
2 from teacher tea,teaches teas,takes ts
3 WHERE tea.id = teas.id
4 and ts.course_id = teas.course_id
5 and ts.sec_id = teas.sec_id
6 and ts.semester = teas.semester
7 and ts.year = teas.year
8 and ts.grade = 'A'
9 group by tea.name, tea.id
10 order by number_of_A_grades desc;
```

Results Explain Describe Saved SQL History

NAME	NUMBER_OF_A_GRADES
Srinivasan	4
Brandt	2
Crick	1

3 rows returned in 0.05 seconds Download

17/

```

1 SELECT tea.id AS teacher_id, tea.name AS teacher_name,
2       stu.id AS student_id, stu.name AS student_name,
3       COUNT(*) AS times_taken
4 from teacher tea, teaches teas, takes ts, student stu
5 WHERE tea.id = teas.id
6 and teas.course_id = ts.course_id
7 and teas.sec_id = ts.sec_id
8 and teas.semester = ts.semester
9 and teas.year = ts.year
10 and ts.id = stu.id
11 group by tea.id, tea.name, stu.id, stu.name
12 order by teacher_name, student_name;

```

Results Explain Describe Saved SQL History

TEACHER_ID	TEACHER_NAME	STUDENT_ID	STUDENT_NAME	TIMES_TAKEN
45565	Katz	45678	Levy	2
98345	Kim	76653	Aoi	1
15151	Mozart	55739	Sanchez	1
10101	Srinivasan	98765	Bourikas	2

More than 10 rows available. Increase rows selector to view more rows.

10 rows returned in 0.11 seconds [Download](#)

18/

```

1 SELECT tea.id AS teacher_id, tea.name AS teacher_name,
2       stu.id AS student_id, stu.name AS student_name,
3       COUNT(*) AS times_taken
4 from teacher tea, teaches teas, takes ts, student stu
5 WHERE tea.id = teas.id
6 and teas.course_id = ts.course_id
7 and teas.sec_id = ts.sec_id
8 and teas.semester = ts.semester
9 and teas.year = ts.year
10 and ts.id = stu.id
11 group by tea.id, tea.name, stu.id, stu.name
12 having COUNT(*) >= 2
13 order by teacher_name, student_name;

```

Results Explain Describe Saved SQL History

TEACHER_ID	TEACHER_NAME	STUDENT_ID	STUDENT_NAME	TIMES_TAKEN
70700	Clark	70700	Clark	2
45565	Katz	45678	Levy	2
10101	Srinivasan	98765	Bourikas	2
10101	Srinivasan	12345	Shankar	3
10101	Srinivasan	00128	Zhang	2

5 rows returned in 0.06 seconds [Download](#)

Exercice 2:

1/ $R(A,B,C)$ et $F = \{A \rightarrow B; B \rightarrow C\}$.

1FN:

A, B et C sont atomiques donc $R(A,B,C)$ est en **1FN**.

2FN:

Une relation est en **2FN** si elle est en **1FN** et qu'aucun attribut non clé ne dépend d'une partie de la clé candidate.

Dans notre cas, on a $A \rightarrow B$ et $B \rightarrow C$. On peut ainsi conclure par la règle de transitivité que $A \rightarrow C$. Par conséquent, A est la clé candidate de la relation.

B et C dépendent tous les deux de la clé candidate entière (qui est A).

On conclut donc que $R(A,B,C)$ est en **2FN**.

3FN:

Une relation est en **3FN** si elle est en **2FN** et qu'aucun attribut non clé ne dépend transitivement d'une clé candidate. Ce n'est pas le cas pour cette relation.

On a B est un attribut non clé et puisque $B \rightarrow C$ et $A \rightarrow B$ donc on a une dépendance transitive. Donc la **3FN** n'est pas respectée. Pour que la **3FN** soit respectée, il faut que $R(A,B,C)$ soit décomposée en $R_1(A,B)$ et $R_2(B,C)$ en vue d'éliminer la dépendance transitive. On conserve ainsi la totalité de l'information et on élimine la dépendance.

BCNF:

Une relation est en **BCNF** si, pour toute dépendance fonctionnelle $A \rightarrow B$, A est une super-clé.

Pour la relation $R_1(A,B)$, on a $A \rightarrow B$, donc A est une super-clé.

Pour la relation $R_2(B,C)$, on a $B \rightarrow C$, donc B est une super-clé.

2/ $R(A,B,C)$ et $F = \{A \rightarrow C; A \rightarrow B\}$.

1FN:

A, B et C sont atomiques donc $R(A,B,C)$ est en **1FN**

2FN:

Une relation est en **2FN** si elle est en **1FN** et qu'aucun attribut non clé ne dépend d'une partie de la clé candidate.

Dans notre cas, on a $A \rightarrow C$ et $A \rightarrow B$. Donc B et C dépendent tous les deux de A.

A est donc la clé candidate. Par conséquent, tous les attributs dépendent de la clé candidate entière (qui est A) donc $R(A,B,C)$ est en **2FN**.

3FN:

Une relation est en **3FN** si elle est en **2FN** et qu'aucun attribut non clé ne dépend transitivement d'une clé candidate. Ce n'est pas le cas pour cette relation.

Dans notre cas, on a $A \rightarrow C$ et $A \rightarrow B$. Puisqu'il n'y a aucune relation transitive, et que les deux attributs non clé (B et C) dépendent de la clé qui est A alors $R(A,B,C)$ est en **3FN**.

BCNF:

Une relation est en **BCNF** si, pour toute dépendance fonctionnelle $A \rightarrow B$, A est une super-clé.

Dans notre cas, pour l'ensemble des dépendances fonctionnelles A est une super-clé ($A \rightarrow C$ et $A \rightarrow B$) alors $R(A,B,C)$ est en **BCNF**.

3/ $R(A,B,C)$ et $F = \{A,B \rightarrow C; C \rightarrow B\}$.

1FN:

A , B et C sont atomiques donc $R(A,B,C)$ est en **1FN**

2FN:

Les clés candidates potentielles sont $\{A,C\}$ et $\{A,B\}$ car elles sont minimales et déterminent l'ensemble des attributs.

Une relation est en **2FN** si elle est en **1FN** et qu'aucun attribut non clé ne dépend d'une partie de la clé candidate.

Dans notre cas, on a $A,B \rightarrow C$ et $C \rightarrow B$. Donc $\{A,B\}$ est une clé candidate car elle permet de déterminer C ($\{A,B\}^+ = \{A,B,C\}$). C'est-à-dire qu'elle est minimale et qu'elle permet de déterminer l'ensemble des attributs.

On a aussi $\{A,C\}$ est une clé candidate car $C \rightarrow B$ donc $\{A,C\}$ détermine tous les attributs aussi et elle est minimale ($\{A,C\}^+ = \{A,B,C\}$).

Pour la clé $\{A,B\}$: C dépend de $\{A,B\}$ entièrement, pas d'une partie seulement

Pour la clé $\{A,C\}$: B dépend de C (qui fait partie de la clé), donc il y a une dépendance partielle

La relation n'est donc pas en 2NF à cause de la dépendance $C \rightarrow B$.

On décompose alors $R(A,B,C)$ en deux relations $R_1(A,C)$ avec la clé $\{A,C\}$ et $R_2(B,C)$ avec la clé C .

3FN:

Une relation est en **3FN** si elle est en **2FN** et qu'aucun attribut non clé ne dépend transitivement d'une clé candidate. Ce n'est pas le cas pour cette relation.

On a $R_1(A,C)$ n'a aucune dépendance fonctionnelle transitive donc R_1 est en **BCNF**.

On a $R_2(C,B)$ n'a aucune dépendance fonctionnelle transitive donc R_2 est en **BCNF**.

BCNF:

Une relation est en **BCNF** si, pour toute dépendance fonctionnelle $A \rightarrow B$, A est une super-clé.

Pour la relation $R_1(A,C)$, on a $C \rightarrow B$ donc $A,C \rightarrow B$.

Pour la relation $R_2(B,C)$, on $C \rightarrow B$ donc $C,B \rightarrow B$.

Ainsi $R_1(A,C)$ et $R_2(B,C)$ sont en **BCNF**.

Exercice 3:

1/

1/ $BC \rightarrow B$ (Réflexivité)

2/ $BC \rightarrow C$ (Réflexivité)

3/ $A \rightarrow B$ (Transitivité de $A \rightarrow BC$ et $BC \rightarrow B$)

4/ $A \rightarrow C$ (Transitivité de $A \rightarrow BC$ et $BC \rightarrow C$)

5/ $A \rightarrow D$ (Transitivité avec $A \rightarrow B$ et $B \rightarrow D$)

6/ $E \rightarrow B$ (Transitivité avec $E \rightarrow A$ et $A \rightarrow B$)

7/ $E \rightarrow C$ (Transitivité avec $E \rightarrow A$ et $A \rightarrow C$)

8/ $CD \rightarrow A$ (Transitivité avec $CD \rightarrow E$ et $E \rightarrow A$)

9/ $CD \rightarrow B$ (Transitivité avec $CD \rightarrow A$ et $A \rightarrow B$)

10/ $E \rightarrow D$ (Transitivité avec $E \rightarrow B$ et $B \rightarrow D$)
 11/ $AC \rightarrow D$ (Transitivité de $AC \rightarrow B$ et $B \rightarrow D$)
 12/ $BC \rightarrow DC$ (Augmentation de $B \rightarrow D$)
 13/ $BC \rightarrow E$ (Transitivité $BC \rightarrow DC$ et $DC \rightarrow E$)
 14/ $BC \rightarrow A$ (Transitivité de $BC \rightarrow E$ et $E \rightarrow A$)
 15/ $AD \rightarrow CD$ (Augmentation de $A \rightarrow C$)
 16/ $AD \rightarrow E$ (Transitivité $AD \rightarrow CD$ et $CD \rightarrow E$)

2/(a)

$K = \{B\}$

Itération 1:

$B \rightarrow D$ et on a $B \in K$.

$K^+ = \{B\} \cup \{D\} = \{B, D\}$

Itération 2:

$D \rightarrow A$ et on a $D \in K$

$K^+ = \{B, D\} \cup \{A\} = \{B, D, A\}$

Itération 3:

$A \rightarrow BCD$ et on a $A \in K$

$K^+ = \{B, D, A\} \cup \{B, C, D\} = \{B, D, C, A\}$

Itération 4:

$C \rightarrow DE$ et on a $C \in K$

$K^+ = \{B, D, C, A\} \cup \{E\} = \{B, D, C, A, E\}$

$\Rightarrow K^+ = \{A, B, C, D, E\}$ car il n'y a aucune règle qui nous permet de déduire F et on a déjà ajouté tous les autres attributs donc K^+ ne changera plus à ce niveau.

$K = \{A, B\}$

Itération 1:

$A \rightarrow BCD$ et on a $A \in K$ donc $K^+ = \{A, B\} \cup \{B, C, D\} = \{A, B, C, D\}$

Itération 2:

$BC \rightarrow DE$ et on a $B \in K$ et $C \in K$ donc $K^+ = \{A, B, C, D\} \cup \{D, E\} = \{A, B, C, D, E\}$

$\Rightarrow K^+ = \{A, B, C, D, E\}$ car il n'y a aucune règle qui nous permet de déduire F et on a déjà ajouté tous les autres attributs donc K^+ ne changera plus à ce niveau.

2/(b)

On calcule la fermeture de $\{A, F\}$

Itération 1:

$A \rightarrow BCD$ et on a $A \in K$ donc $K^+ = \{A, F\} \cup \{B, C, D\} = \{A, F, B, C, D\}$

Itération 2:

$BC \rightarrow DE$ et on a $B \in K$ et $C \in K$ donc $K^+ = \{A, F, B, C, D\} \cup \{D, E\} = \{A, B, C, D, E, F\}$

$\{A, F\}$ détermine tous les attributs ($\{A, B, C, D, E, F\}$) donc c'est une super-clé.

2/(c)

Non, cette relation n'est pas en BCNF.

Décomposition:

Dans notre cas, les super-clés sont $\{A,F\}$, $\{B,F\}$ et $\{D,F\}$.

1FN:

R est **1FN** car tous les attributs sont atomiques.

2FN:

Une relation est en **2FN** si elle est en **1FN** et qu'aucun attribut non clé ne dépend d'une partie de la clé candidate.

On a $A \rightarrow BCD$: A est une partie de la clé candidate $\{A,F\}$ et $\{B,C,D\}$ sont des attributs non-clés.

On a $B \rightarrow D$ et $BC \rightarrow DE$: B est une partie de la clé candidate $\{B,F\}$ et D sont des attributs non-clés.

On a $D \rightarrow A$ et : D est une partie de la clé candidate $\{D,F\}$ et A est un attribut non-clé
Donc, R n'est pas en **2FN**.

La décomposition permettant de résoudre les dépendances partielles est la suivante:

R1(A,B,C,D)

R2(B,D)

R3(A,D)

R4(B,C,E)

R5(A,F)

3FN:

Une relation est en **3FN** si elle est en **2FN** et qu'aucun attribut non clé ne dépend transitivement d'une clé candidate.

Pour R1, A est la clé-candidate.

On a $A \rightarrow BCD$ et $B \rightarrow D$ (dépendance transitive) donc on décompose.

R1(A,B,C)[Représente $F=\{A \rightarrow BC\}$]

R2(B,D)[Représente $F=\{B \rightarrow D\}$]

R3(A,D) [Représente $F=\{A \rightarrow D, D \rightarrow A\}$]

R4(B,C,E)[Représente $F=\{BC \rightarrow DE\}$]

R5(A,F)[Utilisée pour conserver tous les attributs pour les jointures].

BCNF:

Une relation est en **BCNF** si, pour toute dépendance fonctionnelle $A \rightarrow B$, A est une super-clé.

Ceci est le cas pour R1. Pour $A \rightarrow BC$, on a A est une super-clé.

Ceci est le cas pour R2. Pour $B \rightarrow D$, on a B est une super-clé.

Ceci est le cas pour R3 (Équivalence).

Ceci est le cas pour R4. Pour $BC \rightarrow E$, on a BC est une super-clé.

Ceci est le cas pour R5 (Pas de dépendance fonctionnelle).

3/(a)

On a $R1 \cap R2 = \{A\}$

Si A est une clé de R1 ou A est une clé de R2.

On déduit donc dans ce cas que $A \rightarrow BC$ dans ce cas ou alors $A \rightarrow DE$.

Si l'une de ces conditions est satisfaite, nous pouvons affirmer que la décomposition est sans perte d'information. Si on utilise les dépendances fonctionnelles de 1/ ou 2/. On a $A \rightarrow BC$. Donc $R1 \cap R2 = R1$. Alors la décomposition est sans perte.

3/(b)

On a $R1 \cap R2 = \{C\}$.

En utilisant les dépendances fonctionnelles de 1 et 2. On trouve que C ne détermine ni {A,B} ni {D,E} à elle seule.

Ainsi, on ne peut pas déduire que $R1 \cap R2 = R1$ ou $R1 \cap R2 = R2$. Même si C est une super-clé, il faut qu'on ait soit $C \rightarrow AB$ ou $C \rightarrow DE$ ce qui n'est pas le cas.

Donc, il y a une perte d'informations.

Exercice 4:

Le langage utilisé pour cet exercice est Python.

1/

```
#1/Ecrire une procédure qui permet de prendre en paramètre une liste de dépendances fonctionnelles et les affiche.
def print_dependencies(dependencies):
    """usage new"""
    for left, right in dependencies:
        string_left = ','.join(left)
        string_right = ','.join(right)
        print(f"{string_left} -> {string_right}")
```

Le code commence par définir une fonction `print_dependencies` qui prend en paramètre une liste de dépendances fonctionnelles. Ensuite, on utilise une boucle `for` pour parcourir chaque couple (left, right) dans cette liste. À chaque itération, on utilise la méthode `join` pour concaténer les éléments des listes `left` et `right` en chaînes de caractères, séparées par des virgules. Enfin, on utilise une f-string pour afficher ces chaînes sous la forme "`left -> right`", ce qui donne une représentation lisible de chaque dépendance fonctionnelle.

Exécution:

```
mydependencies = [
    [{'A'}, {'B'}],
    [{'A'}, {'C'}],
    [{'C'}, {'G'}, {'H'}],
    [{'C'}, {'G'}, {'I'}],
    [{'B'}, {'H'}]
]
```

```
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
A -> B
A -> C
G,C -> H
G,C -> I
B -> H
Process finished with exit code 0
```

2/

```
#2/Ecrire une procédure qui permet de prendre en paramètre un ensemble de relations 'relations' et les affiche.
def print_relations(relations): 1 usage new *
    for R in relations:
        print("\t", R)
```

Cette procédure va afficher les relations en les préfixant d'une tabulation horizontale.

Exécution:

```
myrelations = [
    {'A', 'B', 'C', 'G', 'H', 'I'},
    {'X', 'Y'}
]
```

```
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
    {'G', 'A', 'I', 'B', 'C', 'H'}
    {'Y', 'X'}
Process finished with exit code 0
```

3/

```
def power_set(inputset): 5 usages new *
    listset = list(inputset)
    result = []
    for r in range(1, len(listset) + 1):
        for comb in itertools.combinations(listset, r):
            result.append(set(comb))
    return result
```

La fonction `power_set` génère l'ensemble des sous-ensembles non vides d'un ensemble donné (`inputset`). D'abord, elle convertit l'entrée en liste pour pouvoir accéder aux éléments par index. Ensuite, elle utilise une boucle pour générer toutes les combinaisons possibles d'éléments de taille 1 à n (n étant la taille de l'ensemble), en utilisant `itertools.combinations`. Chaque combinaison est transformée en ensemble (`set`) puis ajoutée à la liste `result`. À la fin, la fonction retourne tous les sous-ensembles non vides de l'ensemble initial.

Exécution:

```
155 print(power_set({'A','B','C'}))
156

Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
[{'A'}, {'B'}, {'C'}, {'A', 'B'}, {'A', 'C'}, {'B', 'C'}, {'A', 'B', 'C'}]
Process finished with exit code 0
```

4/

```
#4/ Ecrire une fonction qui permet, étant donné un ensemble de dépendances fonctionnelles F et un ensemble d'attributs
def compute_attributes_closure(dependencies, attributes): 3 usages new *
    attributes = set(attributes)
    while True:
        changed = False
        for left, right in dependencies:
            if left.issubset(attributes) and not right.issubset(attributes):
                attributes.update(right)
                changed = True
        if not changed:
            break
    return attributes
```

Cette fonction calcule la clôture d'un ensemble d'attributs par rapport à un ensemble de dépendances fonctionnelles. Elle commence par transformer les attributs donnés en un ensemble, puis applique de manière répétée les dépendances : à chaque itération, si les attributs du côté gauche d'une dépendance sont inclus dans la clôture actuelle et que le côté droit ne l'est pas encore entièrement, elle ajoute les attributs du côté droit à la clôture. Ce processus continue jusqu'à ce qu'aucune nouvelle information ne puisse être ajoutée, garantissant que la clôture finale contient tous les attributs déductibles à partir de l'ensemble initial.

Exécution pour l'ensemble de dépendances défini dans l'énoncé avec {'A'} comme ensemble d'attributs initial.

```
155 closure_result = compute_attributes_closure(mydependencies, {'A'})
156 print(closure_result)
157

Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
{'B', 'H', 'C', 'A'}
Process finished with exit code 0
```

5/

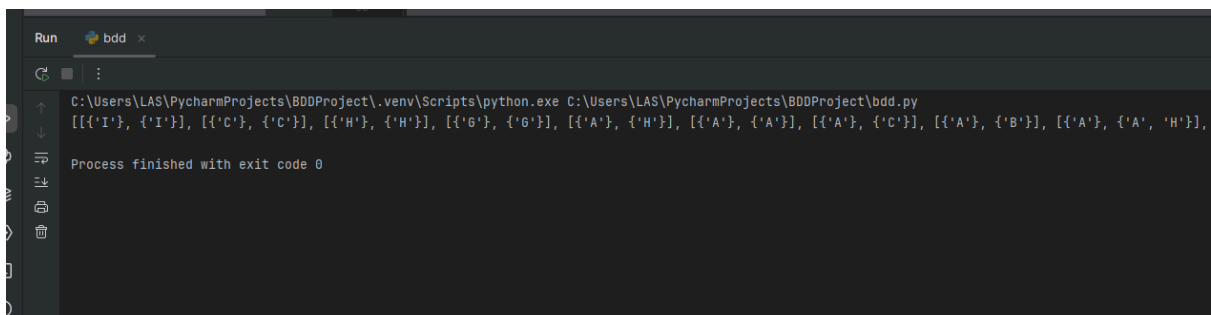

```
#5/Ecrire une fonction qui permet, étant donné un ensemble de dépendances fonctionnelles F, de retourner la clôture de F.
def compute_dependencies_closure(dependencies): 2 usages new *
    r = set()
    for left, right in dependencies:
        r.update(left | right)

    f_plus = [[item,beta] for item in power_set(r) for beta in power_set(compute_attributes_closure(dependencies, item))]

    return f_plus
```

Le code implémente un algorithme permettant de calculer la clôture d'un ensemble de dépendances fonctionnelles. Pour cela, plusieurs fonctions sont utilisées. Tout d'abord, la fonction `power_set(inputset)` génère tous les sous-ensembles possibles d'un ensemble d'attributs donné, ce qui est nécessaire pour examiner toutes les combinaisons d'attributs dans le calcul de la clôture. Ensuite, la fonction `compute_attributes_closure(dependencies, attributes)` calcule la clôture d'attributs : à partir d'un ensemble d'attributs donné, elle applique les dépendances fonctionnelles de façon itérative pour déterminer l'ensemble complet des attributs qui peuvent être déduits. Enfin, la fonction principale, souvent appelée `fd_closure_with_sets`, parcourt tous les sous-ensembles possibles d'attributs (grâce à `power_set`), calcule leur clôture (avec `compute_attributes_closure`), et en déduit les dépendances $X \rightarrow AX$ pour chaque attribut A appartenant à la clôture de X mais pas à X lui-même. Les dépendances ainsi obtenues sont stockées sous forme de couples, avec la partie gauche et la partie droite sous forme d'ensembles (set), ce qui permet une représentation flexible et cohérente. Le résultat final est une liste représentant la clôture complète de l'ensemble de dépendances initial.

Exécution:



```
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
[[{'I'}, {'I'}], [{'C'}, {'C'}], [{'H'}, {'H'}], [{'G'}, {'G'}], [{'A'}, {'H'}], [{'A'}, {'A'}], [{'A'}, {'C'}], [{'A'}, {'B'}], [{'A'}, {'A', 'H'}],
Process finished with exit code 0
```

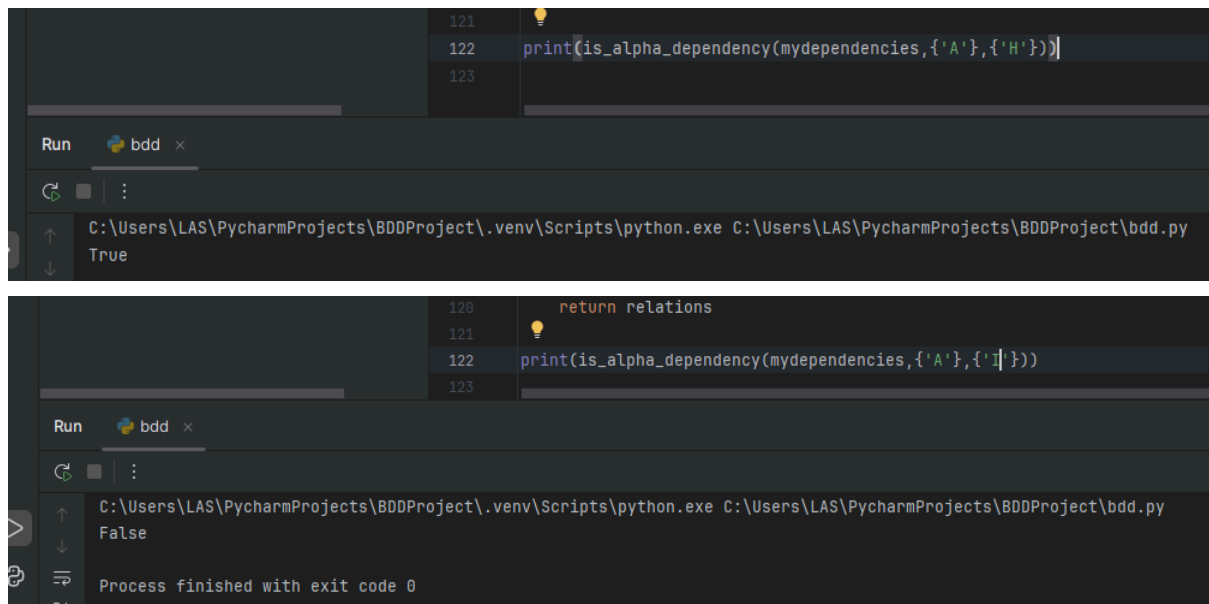
6/

```
#6/Ecrire une fonction qui permet, étant donnée un ensemble de dépendances fonctionnelles F
def is_alpha_dependency(dependencies,alpha,beta): new *
    all_alpha_potential_combinations = compute_attributes_closure(dependencies,alpha)
    return beta.issubset(all_alpha_potential_combinations)
```

Cette fonction, nommée `is_alpha_dependency`, permet de vérifier si un ensemble d'attributs alpha détermine fonctionnellement un autre ensemble beta, en utilisant un ensemble de dépendances fonctionnelles donné. Pour cela, elle commence par calculer la clôture de alpha (c'est-à-dire l'ensemble de tous les attributs que alpha peut déterminer) en appelant une fonction `compute_attributes_closure`. Ensuite, elle vérifie si tous les attributs de beta sont inclus dans cette clôture, en utilisant la

méthode `issubset`. Si c'est le cas, cela signifie que `alpha` détermine bien `beta`, et la fonction retourne `True`, sinon elle retourne `False`.

Exécution:



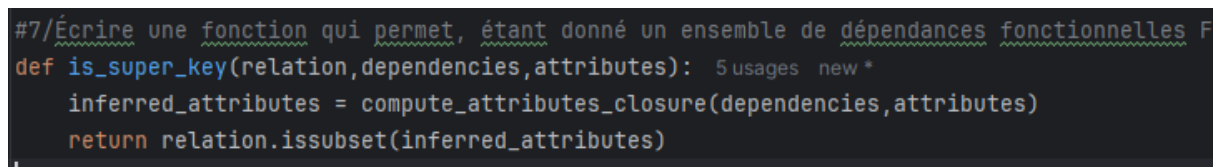
```
121
122 print(is_alpha_dependency(mydependencies,{'A'},{'H'}))
123

Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
True

120 return relations
121
122 print(is_alpha_dependency(mydependencies,{'A'},{'I'}))
123

Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
False
Process finished with exit code 0
```

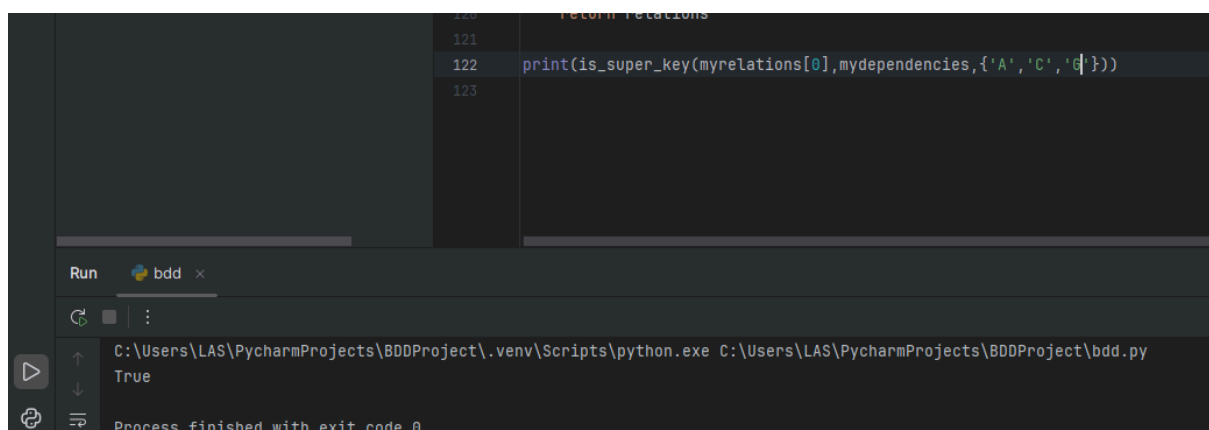
7/



```
#7/Écrire une fonction qui permet, étant donné un ensemble de dépendances fonctionnelles F
def is_super_key(relation,dependencies,attributes): 5 usages new *
    inferred_attributes = compute_attributes_closure(dependencies,attributes)
    return relation.issubset(inferred_attributes)
```

La fonction `is_super_key` permet de vérifier si un ensemble d'attributs `attributes` constitue une super-clé pour une relation `relation`, en tenant compte d'un ensemble de dépendances fonctionnelles `dependencies`. Pour cela, elle calcule d'abord la clôture de `attributes`, c'est-à-dire l'ensemble des attributs que `attributes` permet de déterminer en utilisant les dépendances données. Cette clôture est obtenue via la fonction `compute_attributes_closure`. Ensuite, la fonction vérifie si tous les attributs de la relation `relation` sont inclus dans cette clôture. Si oui, cela signifie que `attributes` permet de déterminer tous les attributs de la relation, donc c'est une super-clé, et la fonction retourne `True`. Sinon, elle retourne `False`.

Exécution:



```
120 return relations
121
122 print(is_super_key(myrelations[0],mydependencies,{'A','C','d'}))
123

Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
True
Process finished with exit code 0
```

```
122 print(is_super_key(myrelations[0],mydependencies,{'A','C'}))
123
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
False
```

8/

```
#8/Ecrire une fonction qui permet, etant donnee un ensemble de dependances fonctionnelles F,
def is_candidate_key(relation, dependencies, attributes): 2usages new *
    if not is_super_key(relation, dependencies, attributes):
        return False

    for attr in attributes:
        reduced_set = attributes - {attr}
        if is_super_key(relation, dependencies, reduced_set):
            return False

    return True
```

La fonction `is_candidate_key` permet de vérifier si un ensemble d'attributs `attributes` est une clé candidate pour une relation `relation`, en tenant compte d'un ensemble de dépendances fonctionnelles `dependencies`. Elle commence par vérifier si `attributes` est une super-clé, c'est-à-dire si elle permet de déterminer tous les attributs de la relation. Si ce n'est pas le cas, `attributes` ne peut pas être une clé candidate et la fonction retourne `False`. Ensuite, pour garantir que `attributes` est minimale, la fonction teste pour chaque attribut de `attributes` si l'ensemble réduit (obtenu en enlevant cet attribut) reste une super-clé. Si c'est le cas, cela signifie que `attributes` n'est pas minimale, donc pas candidate, et la fonction retourne `False`. Si aucun sous-ensemble strict de `attributes` n'est une super-clé, alors `attributes` est bien une clé candidate, et la fonction retourne `True`.

Exécution:

```
122 print(is_candidate_key(myrelations[0],mydependencies,{'A','G'}))
123
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
True
Process finished with exit code 0
```

```
122 print(is_candidate_key(myrelations[0],mydependencies,{'A','C','G'}))
123
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
False
Process finished with exit code 0
```

9/

```
#9/ Ecrire une fonction qui, étant donnée une relation R et un ensemble de dépendances fonctionnelles F
def find_all_candidate_keys(relation, dependencies): new *
    return [item for item in power_set(relation) if is_candidate_key(relation,dependencies,item)]
```

La fonction `find_all_candidate_keys` retourne la liste de toutes les clés candidates d'une relation `relation`, en se basant sur un ensemble de dépendances fonctionnelles `dependencies`. Elle génère d'abord l'ensemble des sous-ensembles possibles des attributs de la relation à l'aide de la fonction `power_set`. Pour chaque sous-ensemble `item`, elle vérifie s'il s'agit d'une clé candidate en appelant la fonction `is_candidate_key`. Si c'est le cas, l'ensemble est ajouté à la liste des résultats. La fonction renvoie finalement la liste complète de tous les sous-ensembles de la relation qui sont des clés candidates.

Exécution:

```
120 return relations
121
122 print(find_all_candidate_keys(myrelations[0],mydependencies))
123
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
[{'G', 'A'}]
Process finished with exit code 0
```

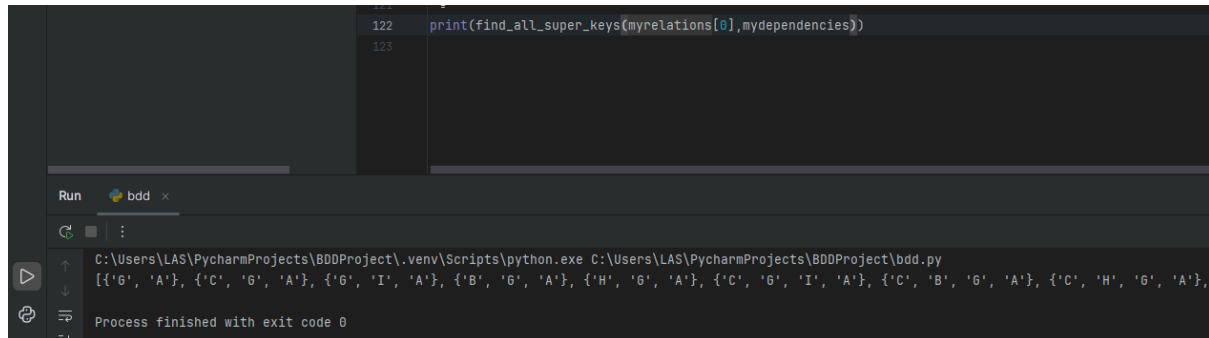
10/

```
#10/ Ecrire une fonction qui, étant donnée une relation R et un ensemble de dépendances fonctionnelles F,
def find_all_super_keys(relation, dependencies): new *
    return [item for item in power_set(relation) if is_super_key(relation, dependencies, item)]
```

La fonction `find_all_super_keys` permet de trouver toutes les super-clés d'une relation `relation`, à partir d'un ensemble de dépendances fonctionnelles `dependencies`. Elle génère l'ensemble des sous-ensembles possibles des attributs de la relation grâce à la fonction `power_set`, puis teste chacun de ces sous-ensembles avec la fonction `is_super_key`. Si un sous-ensemble est une

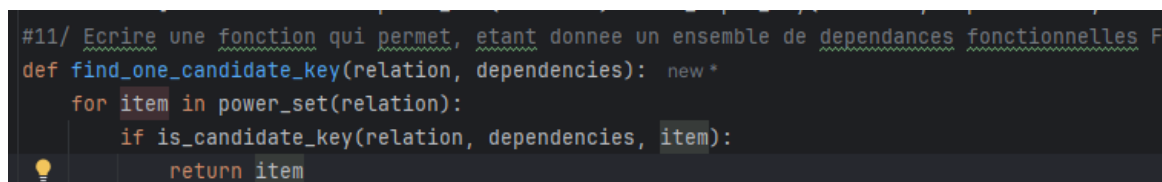
super-clé, il est ajouté à la liste des résultats. La fonction retourne donc la liste de tous les ensembles d'attributs qui sont des super-clés pour la relation donnée.

Exécution:



```
122 print(find_all_super_keys(myrelations[0],mydependencies))
123
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
[{'G', 'A'}, {'C', 'G', 'A'}, {'G', 'I', 'A'}, {'B', 'G', 'A'}, {'H', 'G', 'A'}, {'C', 'G', 'I', 'A'}, {'C', 'B', 'G', 'A'}, {'C', 'H', 'G', 'A'},
Process finished with exit code 0
```

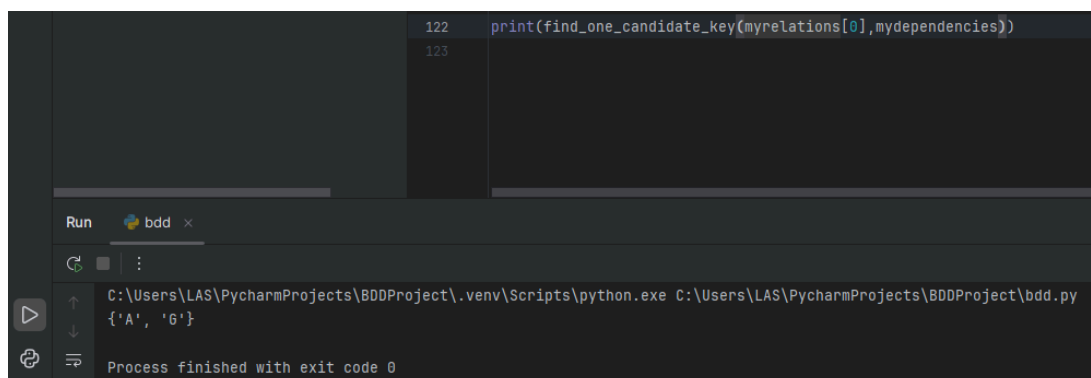
11/



```
#11/ Ecrire une fonction qui permet, étant donnée un ensemble de dépendances fonctionnelles F
def find_one_candidate_key(relation, dependencies):
    for item in power_set(relation):
        if is_candidate_key(relation, dependencies, item):
            return item
```

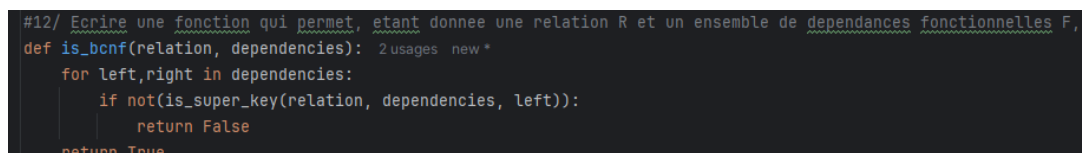
La fonction `find_one_candidate_key` permet de trouver une seule clé candidate pour une relation `relation`, à partir d'un ensemble de dépendances fonctionnelles `dependencies`. Elle parcourt tous les sous-ensembles possibles des attributs de la relation, générés via la fonction `power_set`. Pour chaque sous-ensemble `item`, elle vérifie s'il s'agit d'une clé candidate en appelant la fonction `is_candidate_key`. Dès qu'elle en trouve une, elle la retourne immédiatement, ce qui permet d'obtenir rapidement une clé candidate sans devoir toutes les lister.

Exécution:



```
122 print(find_one_candidate_key(myrelations[0],mydependencies))
123
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
{'A', 'G'}
Process finished with exit code 0
```

12/



```
#12/ Ecrire une fonction qui permet, étant donnée une relation R et un ensemble de dépendances fonctionnelles F,
def is_bcnf(relation, dependencies):
    for left, right in dependencies:
        if not(is_super_key(relation, dependencies, left)):
            return False
    return True
```

La fonction `is_bcnf` permet de vérifier si une relation `relation` est en forme normale de Boyce-Codd (BCNF), en tenant compte d'un ensemble de dépendances fonctionnelles `dependencies`. Pour cela, elle parcourt chaque dépendance

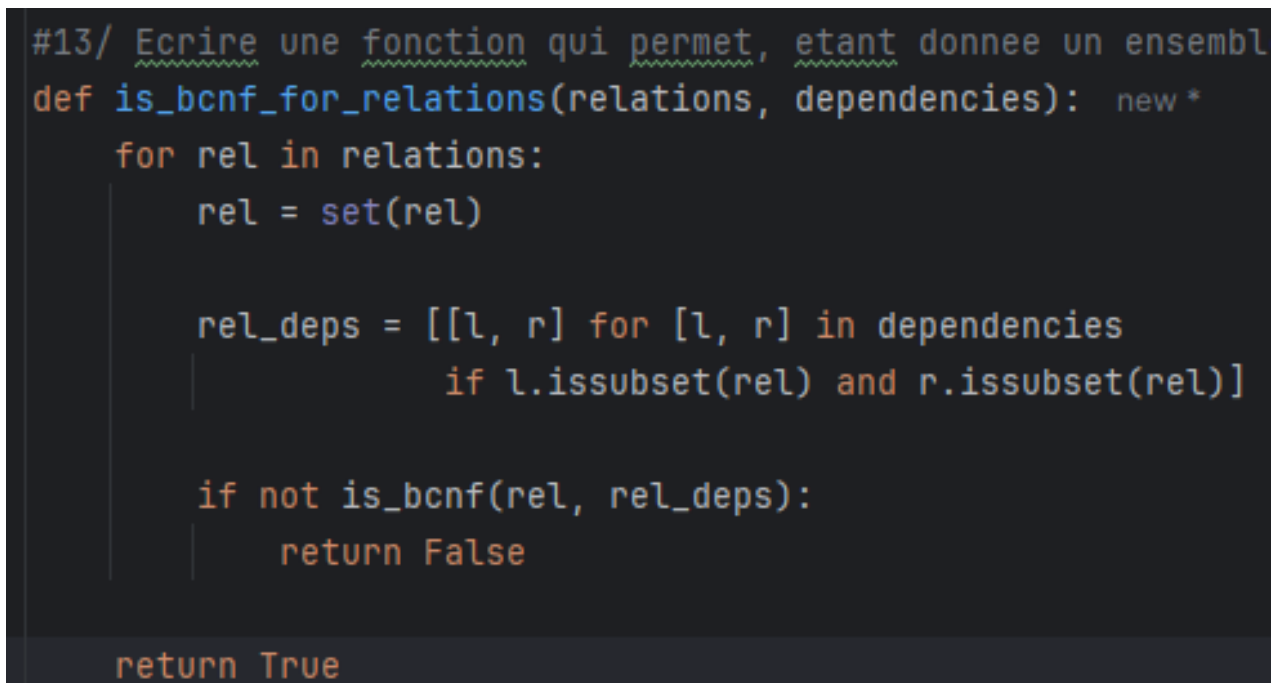
fonctionnelle de la forme left -> right dans dependencies, et vérifie si le côté gauche (left) est une super-clé de la relation. Si ce n'est pas le cas pour au moins une dépendance, la relation viole la BCNF, et la fonction retourne False. Si toutes les dépendances respectent cette condition, la relation est bien en BCNF, et la fonction retourne True.

Exécution:



```
122 print(is_bcnf(myrelations[0], mydependencies))
123
Run bdd x
C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
False
```

13/



```
#13/ Ecrire une fonction qui permet, étant donnée un ensemble
def is_bcnf_for_relations(relations, dependencies):
    for rel in relations:
        rel = set(rel)

        rel_deps = [[l, r] for [l, r] in dependencies
                      if l.issubset(rel) and r.issubset(rel)]

        if not is_bcnf(rel, rel_deps):
            return False

    return True
```

La fonction `is_bcnf_for_relations` permet de vérifier si un ensemble de relations respecte la forme normale de Boyce-Codd (BCNF), en tenant compte d'un ensemble global de dépendances fonctionnelles. Pour chaque relation du schéma, elle filtre les dépendances fonctionnelles applicables, c'est-à-dire celles dont le côté gauche et le côté droit sont inclus dans les attributs de la relation. Ensuite, elle utilise la fonction `is_bcnf` pour déterminer si cette relation individuelle est en BCNF par rapport aux dépendances qui la concernent. Si au moins une relation viole la BCNF, la fonction retourne False. Si toutes les relations respectent la BCNF, elle retourne True, indiquant que le schéma entier est conforme à cette forme normale.

Exécution:

```
122 print(is_bcnf_for_relations(myrelations,mydependencies))
123
```

Run bdd x

C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
False

```

def bcnf_decomposition(relations, dependencies): 1 usage new *
    result = [set(r) for r in relations]

    while True:
        modified = False

        for idx, R in enumerate(result):
            if is_bcnf(R, dependencies):
                continue

            violating_fd = None
            for left, right in dependencies:
                if left.issubset(R) and right.issubset(R):
                    if not is_super_key(R, dependencies, left):
                        violating_fd = (left, right)
                        break

            if violating_fd:
                left, right = violating_fd
                R1 = left.union(right)
                R2 = (R - right).union(left)

                result.pop(idx)
                result.extend([R1, R2])

                modified = True
                break

            if not modified:
                break
        cleaned = []
        for rel in result:
            if not any(rel < other for other in result):
                cleaned.append(rel)

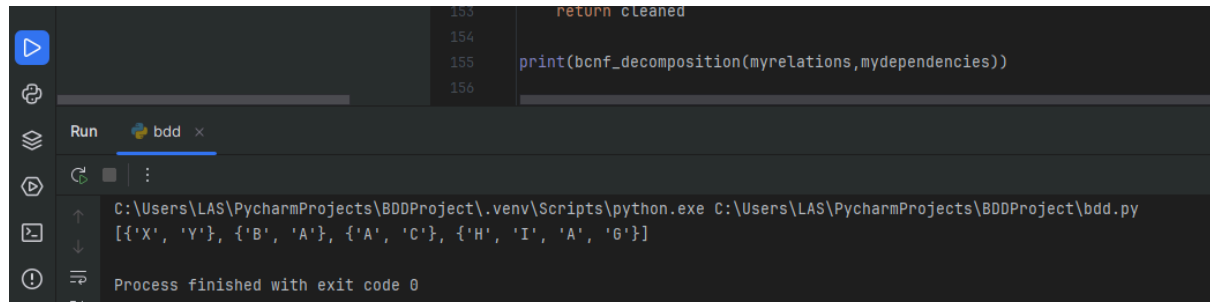
    return cleaned

```

La fonction `bcnf_decomposition` permet de décomposer un ensemble de relations en un schéma respectant la forme normale de Boyce-Codd (BCNF), en utilisant un ensemble donné de dépendances fonctionnelles. Pour chaque relation du schéma, la fonction vérifie si elle est en BCNF. Si ce n'est pas le cas, elle identifie une dépendance fonctionnelle qui viole la BCNF, c'est-à-dire une dépendance dont le côté gauche n'est pas une super-clé de la relation. La relation est alors décomposée en deux sous-relations : l'une contenant les attributs du côté gauche et du côté droit de la dépendance, l'autre contenant les attributs restants, tout en conservant le côté

gauche. Ce processus se répète jusqu'à ce que toutes les relations soient en BCNF. À la fin, un nettoyage est effectué pour supprimer les relations strictement incluses dans d'autres, afin d'éviter les redondances. La fonction retourne la liste finale des relations en BCNF.

Exécution:



```
153     return cleaned
154
155     print(bcnf_decomposition(myrelations, mydependencies))
156
```

Run bdd x

C:\Users\LAS\PycharmProjects\BDDProject\.venv\Scripts\python.exe C:\Users\LAS\PycharmProjects\BDDProject\bdd.py
[{'X', 'Y'}, {'B', 'A'}, {'A', 'C'}, {'H', 'I', 'A', 'G'}]

Process finished with exit code 0