TP Schéma Physique et Optimisation

March 18, 2024

1 TP BD Avancée

ANAS CHAKIR 2A ISI G4

ABDELHADI NASMANE 2A ISI G4

MAROUANE AKASSAB 2A ISI G4 Ce travail a été réalisé en équipe de 3.

1.1 EXERCICE 1

Question A:

[ ]:
```
CREATE TABLE clientsA AS SELECT * FROM clients;
CREATE TABLE commandesA AS SELECT * FROM commandes;
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     9 |   369 |    10   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |            |     9 |   369 |    10   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| CLIENTSA   |     1 |    22 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| COMMANDESA |   830 | 15770 |     7   (0)| 00:00:01 |
-------------------------------------------------------------------------------


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

   1 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
   2 - filter("CL"."CONTACT"='Thomas Hardy')
```

Interpretation: On utilise pas d'index, donc on a pas d'optimistion.

Question B:

[ ]:
```
CREATE TABLE clientsB AS SELECT * FROM clients;
CREATE TABLE commandesB AS SELECT * FROM commandes;
CREATE INDEX index_sec_trie1 ON clientsB(codeclient);
CREATE INDEX index_sec_trie2 ON clientsB(contact);
CREATE INDEX index_sec_trie3 ON commandesB(codeclient);
```

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                |     9 |   369 |     9   (0)| 00:00:01 |
|*  1 |  HASH JOIN                          |                |     9 |   369 |     9   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTSB       |     1 |    22 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | INDEX_SEC_TRIE2 |     1 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL                 | COMMANDESB     |   830 | 15770 |     7   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------

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

   1 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
   3 - access("CL"."CONTACT"='Thomas Hardy')
```

Interpretation: On remarque que l'utilisation de l'index secondaire sur l'attribut clientB.contact car la condition de sélection est une égalité, donc en utilisant l'index secondaire on trouve bien le résultat avec un coût optimal (ex: recherche dichotomique).

index_sec_trie1 est négligé car le chemin d'accès choisi pour accéder à la table clients est index_sec_tri2 car il est de cardinalité 1.

index_sec_trie3 négligé puisque le Hash Join nécessitera dans ce cas de traverser des pointeurs supplémentaires, donc c'est mieux de le faire par un parcours séquentiel sans avoir à faire la traversée de pointeurs.

Question C:

[ ]:
```
CREATE cluster cluster1 (codeclientindex CHAR(5));
CREATE INDEX index_primaire_1 ON CLUSTER cluster1;
CREATE TABLE clientsC CLUSTER cluster1(codeclient) AS SELECT * FROM clients;
CREATE INDEX index_sec_trie1 ON clientsC(contact);
CREATE TABLE commandesC AS SELECT * FROM commandes;
CREATE INDEX index_sec_trie2 ON commandesC(codeclient);
```

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                |     9 |   387 |    10   (0)| 00:00:01 |
|*  1 |  HASH JOIN                          |                |     9 |   387 |    10   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTSC       |     1 |    24 |     3   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | INDEX_SEC_TRIE1 |     1 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL                 | COMMANDESC     |   830 | 15770 |     7   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------

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

   1 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
   3 - access("CL"."CONTACT"='Thomas Hardy')
```

Interpretation: Même justification que la question précédente.

Question D:

```
[ ]: CREATE cluster cluster1 (contactindex VARCHAR2(30));
CREATE INDEX index_primaire_1 ON CLUSTER cluster1;
CREATE TABLE clientsD CLUSTER cluster1(contact) AS SELECT * FROM clients;
CREATE INDEX index_sec_trie1 ON clientsD(codeclient);
CREATE TABLE commandesD AS SELECT * FROM commandes;
CREATE INDEX index_sec_trie2 ON commandesD(codeclient);
```

```
-----------------------------------------------------------------------------------------
| Id  | Operation             | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                  |     8 |   344 |     9   (0)| 00:00:01 |
|*  1 |  HASH JOIN            |                  |     8 |   344 |     9   (0)| 00:00:01 |
|   2 |   TABLE ACCESS CLUSTER| CLIENTSD         |     1 |    24 |     2   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN  | INDEX_PRIMAIRE_1 |     1 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL   | COMMANDESD       |   830 | 15770 |     7   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------------------

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

   1 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
   3 - access("CL"."CONTACT"='Thomas Hardy')
```

Interpretation: Même justification que la question précédente, la différence est qu' ici on a utilisé l'index primaire au lieu de l'index secondaire sur clientsD.contact.

Question E:

```
[ ]: CREATE cluster cluster1 (codeclientindex CHAR(5));
CREATE INDEX index_primaire_jointure_trie ON CLUSTER cluster1;
CREATE TABLE clientsE CLUSTER cluster1(codeclient) AS SELECT * FROM clients;
CREATE TABLE commandesE CLUSTER cluster1(codeclient) AS SELECT * FROM commandes;
CREATE INDEX index_sec_trie1 ON clientsE(contact);
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                 |    14 |   742 |     4   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                       |                 |    14 |   742 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTSE        |     1 |    24 |     3   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | INDEX_SEC_TRIE1 |     1 |       |     1   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS CLUSTER              | COMMANDESE      |    14 |   406 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------------------

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

   3 - access("CL"."CONTACT"='Thomas Hardy')
   4 - filter("CL"."CODECLIENT"="CO"."CODECLIENT")
```

Interpretation: On accède à clientsE par l'index secondaire trié puisque la cardinalité estimé du résultat est égale à 1 ce qui est optimal. De plus, l'optimiseur estime que l'accès par l'index primaire de jointure trié à commandesE donnera 15 lignes, ce qui est beacoup mieux que les 830

lignes données par le parcours sequentiel, car elle est dans le même cluster que clientsE et grâce à l'index on pourra faire la jointure rapidement.

L'utilisation du Nested Loop car c'est la plus simple et la plus rapide. Un hash Join perdra du temps à construire la table de hachage et que le join n'implique pas d'autres entrée/sortie supplémentaires.

Question F:

[ ]:
```
CREATE CLUSTER cluster1(contactindex VARCHAR2(30)) SINGLE TABLE HASHKEYS 80;
CREATE TABLE clientsF CLUSTER cluster1(contact) AS SELECT * FROM clients;
CREATE INDEX index_sec_trie1 ON clientsF(codeclient);
CREATE TABLE commandesF AS SELECT * FROM commandes;
CREATE INDEX index_sec_trie2 ON commandesF(codeclient);
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |    13 |   559 |     7   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |            |    13 |   559 |     7   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| COMMANDESF |   830 | 15770 |     7   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS HASH| CLIENTSF   |     1 |    24 |            |          |
-------------------------------------------------------------------------------


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

   3 - access("CL"."CONTACT"='Thomas Hardy')
       filter("CL"."CODECLIENT"="CO"."CODECLIENT")
```

Interpretation: La différence entre cette démarche et celle de la question D, c'est qu'on utilise un index hashé sur clientsF.contact au lieu d'un index trié.

L'utilisation d'une Nested Loop est optimal, puisqu'on doit faire un parcours inévitable des 830 lignes.

Question G:

[ ]:
```
CREATE CLUSTER cluster1(codeclientindex CHAR(5)) SINGLE TABLE HASHKEYS 80;
CREATE TABLE clientsG CLUSTER cluster1(codeclient) AS SELECT * FROM clients;
CREATE INDEX index_sec_trie1 ON clientsG(contact);
CREATE TABLE commandesG AS SELECT * FROM commandes;
CREATE INDEX index_sec_trie2 ON commandesG(codeclient);
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     9 |   387 |     7   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |            |     9 |   387 |     7   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| COMMANDESG |   830 | 15770 |     7   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS HASH| CLIENTSG   |     1 |    24 |            |          |
-------------------------------------------------------------------------------


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

   3 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
       filter("CL"."CONTACT"='Thomas Hardy')
```

Interpretation: La différence entre cette démarche et celle de la question C, c'est que ici on utilise un index hashé sur clientsG.contact au lieu d'un index trié.

L'utilisation d'une Nested Loop est optimal, puisqu'on doit faire un parcours inévitable des 830 lignes.

Question H:

[ ]:
```
CREATE CLUSTER cluster1(codeclientindex CHAR(5)) HASHKEYS 80;
CREATE TABLE clientsH CLUSTER cluster1(codeclient) AS SELECT * FROM clients;
CREATE TABLE commandesH CLUSTER cluster1(codeclient) AS SELECT * FROM commandes;
CREATE INDEX index_sec_trie1 ON clientsH(contact);
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |    14 |   742 |    27   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |            |    14 |   742 |    27   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| CLIENTSH   |     1 |    24 |    27   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS HASH| COMMANDESH |    14 |   406 |            |          |
-------------------------------------------------------------------------------


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

   2 - filter("CL"."CONTACT"='Thomas Hardy')
   3 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
```

Interpretation: On accéde à clientsH.contact par l'index secondaire trié puisque la cardinalité estimé du résultat est égale à 1 ce qui est optimal.

L'utilisation de l'index hashé est pire que l'index trié de la Question E, puisqu'on doit hasher puis chercher le tuple correspondant dans l'autre table, dans le paquet correspondant, ce ce qui explique le coût élevé par rapport au schéma de la Question E.

L'utilisation du Nested Loop car c'est la plus simple et la plus rapide. Un hash Join perdrait du temps à construire la table de hachage et que le join n'implique pas d'autre entrée/sortie.

Question I:

```
[ ]: CREATE TABLE clientsI AS SELECT * FROM clients;
     CREATE INDEX index_sec_trie1 ON clientsI (codeclient);
     CREATE INDEX index_sec_trie2 ON clientsI (contact);
     CREATE TABLE commandesI AS SELECT * FROM commandes;
     CREATE INDEX index_sec_trie3 ON commandesI (codeclient, datecomm, nocomm);
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                |     9 |   369 |     3   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                       |                |     9 |   369 |     3   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTSI       |     1 |    22 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | INDEX_SEC_TRIE2 |     1 |       |     1   (0)| 00:00:01 |
|*  4 |   INDEX RANGE SCAN                  | INDEX_SEC_TRIE3 |     9 |   171 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------

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

   3 - access("CL"."CONTACT"='Thomas Hardy')
   4 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
```

Interpretation: L'optimiseur ici a choisi d'utiliser une Nested Loop au lieu de Hash join qui était utilisé dans la Question B, cela explique l'utilisation de l'index multiattributs puisque la condition de jointure va être appliqué sur la table commandesI avant de faire la jointure.

D'où le coût minimal.

Question J:

```
[ ]: CREATE TABLE clientsJ AS SELECT * FROM clients;
     CREATE INDEX index_sec_trie1 ON clientsJ (codeclient);
     CREATE INDEX index_sec_trie2 ON clientsJ (contact);
     CREATE TABLE commandesJ AS SELECT * FROM commandes;
     CREATE INDEX index_sec_trie3 ON commandesJ (datecomm, nocomm, codeclient);
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                |     9 |   369 |     5   (0)| 00:00:01 |
|*  1 |  HASH JOIN                          |                |     9 |   369 |     5   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTSJ       |     1 |    22 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | INDEX_SEC_TRIE2 |     1 |       |     1   (0)| 00:00:01 |
|   4 |   INDEX FAST FULL SCAN              | INDEX_SEC_TRIE3 |   830 | 15770 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------

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

   1 - access("CL"."CODECLIENT"="CO"."CODECLIENT")
   3 - access("CL"."CONTACT"='Thomas Hardy')
```

Interpretation: Les performances du schéma de la Question I sont mieux que celles de la Question J. Cela est expliqué par le changement de l'ordre des attributs de l'index trié puisque la

condition de jointure nécessite un tri de l'attribut commandeI.codeclient mais le tri des autres attributs n'est pas important.

1.2 EXERCICE 2

Question A:

```
[ ]: EXPLAIN SELECT co.nocomm, co.datecomm
     FROM clientsA cl, commandesA co
     WHERE cl.codeclient = co.codeclient
     AND cl.contact = 'Thomas Hardy' ;
     SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	ALL	*NULL*	*NULL*	*NULL*	*NULL*	91	10.00	Using where
1	SIMPLE	co	*NULL*	ALL	*NULL*	*NULL*	*NULL*	*NULL*	830	10.00	Using where; Using join buffer (hash join)

2 rows (0.006 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	766.910180

1 row (0.003 s) Edit, Export

Interpretation: On utilise pas d'index, donc pas d'optimistion.

Question B:

```
[ ]: CREATE INDEX index_sec1 ON clientsB (codeclient);
     CREATE INDEX index_sec2 ON clientsB (contact);
     CREATE INDEX index_sec3 ON commandesB (codeclient);

     EXPLAIN SELECT co.nocomm, co.datecomm
     FROM clientsB cl, commandesB co
     WHERE cl.codeclient = co.codeclient
     AND cl.contact = 'Thomas Hardy' ;
     SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	ref	index_sec1,index_sec2	index_sec2	123	const	1	100.00	Using where
1	SIMPLE	co	*NULL*	ref	index_sec3	index_sec3	21	nasmanea.cl.codeclient	9	100.00	*NULL*

2 rows (0.001 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	3.613045

1 row (0.002 s) Edit, Export

Interpretation: On remarque l'utilisation de l'index secondaire sur l'attribut clientB.contact, puisque la condition de sélection est une égalité, donc en utilisant l'index secondaire on trouvera le résultat avec un coût optimal (ex: recherche dichotomique).

De même, l'optimiseur utilise l'index_sec_trie3 pour accéder à commandesB.codeclient.

index_sec_trie1 est négligé car le chemin d'accès choisi pour clients est index_sec_tri2 car il est de cardinalité 1.

Question C:

```
[ ]: ALTER TABLE clientsC ADD PRIMARY KEY (contact);
CREATE INDEX index_sec1 ON clientsC (codeclient);
CREATE INDEX index_sec2 ON commandesC (codeclient);

EXPLAIN SELECT co.nocomm, co.datecomm
FROM clientsC cl, commandesC co
WHERE cl.codeclient = co.codeclient
AND cl.contact = 'Thomas Hardy' ;
SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	const	PRIMARY,index_sec1	PRIMARY	122	const	1	100.00	*NULL*
1	SIMPLE	co	*NULL*	ref	index_sec2	index_sec2	21	const	14	100.00	*NULL*

2 rows (0.001 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	4.899000

1 row (0.002 s) Edit, Export

Interpretation: Même justification que la question précédente, à part l'utilisation de l'index primaire au lieu de l'index secondaire.

Question D:

```
[ ]: ALTER TABLE clientsD ADD PRIMARY KEY (codeclient);
CREATE INDEX index_sec1 ON clientsD (contact);

EXPLAIN SELECT co.nocomm, co.datecomm
FROM clientsD cl, commandesD co
WHERE cl.codeclient = co.codeclient
AND cl.contact = 'Thomas Hardy' ;
SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	ref	PRIMARY,index_sec1	index_sec1	123	const	1	100.00	Using index
1	SIMPLE	co	*NULL*	ALL	*NULL*	*NULL*	*NULL*	*NULL*	830	10.00	Using where; Using join buffer (hash join)

2 rows (0.004 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	85.600227

1 row (0.004 s) Edit, Export

Interpretation: Ici on a le choix entre deux chemin d'accès à la table clientsD. L'optimiseur choisit l'index secondaire sur clientsD.contact puisque la cardinalité estimée du resultat est égale à 1 ce qui est optimal, on n'utilise pas l'index primaire car on a unicité du chemin d'accès.

Le coût augmente significativement puisqu'on fait un FULL SCAN sur la deuxième table.

Question E:

[ ]:
```
ALTER TABLE clientsE ADD PRIMARY KEY (codeclient);
CREATE INDEX index_sec1 ON clientsE (contact);
CREATE INDEX index_sec2 ON commandesE (codeclient);

EXPLAIN SELECT co.nocomm, co.datecomm
FROM clientsE cl, commandesE co
WHERE cl.codeclient = co.codeclient
AND cl.contact = 'Thomas Hardy' ;
SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	ref	PRIMARY,index_sec1	index_sec1	123	const	1	100.00	Using index
1	SIMPLE	co	*NULL*	ref	index_sec2	index_sec2	21	nasmanea.cl.codeclient	9	100.00	*NULL*

2 rows (0.001 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	3.613045

1 row (0.002 s) Edit, Export

Interpretation: On est dans les même circonstances de la Question D, mais cette fois on a un index secondaire sur commandesE.codeclient, donc on a pas le problème du FULL SCAN, d'où le coût diminue.

Question F:

[ ]:
```
CREATE INDEX index_sec1 ON clientsF (contact);
CREATE INDEX index_sec2 ON clientsF (codeclient);
CREATE INDEX index_sec3 ON commandesF (codeclient, datecomm, nocomm);

EXPLAIN SELECT co.nocomm, co.datecomm
FROM clientsF cl, commandesF co
WHERE cl.codeclient = co.codeclient
AND cl.contact = 'Thomas Hardy' ;
SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	ref	index_sec1,index_sec2	index_sec1	123	const	1	100.00	Using where
1	SIMPLE	co	*NULL*	ref	index_sec3	index_sec3	21	nasmanea.cl.codeclient	9	100.00	Using index

2 rows (0.002 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	1.929407

1 row (0.003 s) Edit, Export

Interpretation: Ici on choisit l'index secondaire clientsF.contact pour accéder à la table clientF, car la cardinalité estimée est 1, on n'utilise pas l'index secondaire index_sec2 sur clientsF.codeclient

car on a unicité du chemin d'accès.

On utilise de plus l'index tripet pour accéder à la table commandeF. On remarque une baisse significative du coût.

Question G:

```
[ ]: CREATE INDEX index_sec1 ON clientsG (contact);
     CREATE INDEX index_sec2 ON clientsG (codeclient);
     CREATE INDEX index_sec3 ON commandesG (datecomm, nocomm, codeclient);

     EXPLAIN SELECT co.nocomm, co.datecomm
     FROM clientsG cl, commandesG co
     WHERE cl.codeclient = co.codeclient
     AND cl.contact = 'Thomas Hardy' ;
     SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	ref	index_sec1,index_sec2	index_sec1	123	const	1	100.00	*NULL*
1	SIMPLE	co	*NULL*	index	*NULL*	index_sec3	30	*NULL*	830	10.00	Using where; Using index; Using join buffer (hash join)

2 rows (0.001 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	85.600227

1 row (0.002 s) Edit, Export

Interpretation: Ici on choisit l'index secondaire clientsG.contact pour accéder à la table clientsG, car la cardinalité estimée est 1, on n'utilise pas l'index secondaire index_sec2 sur clientsG.codeclient car on a unicité du chemin d'accès.

On utilise de plus l'index tripet pour accéder à la table commandeG. Le coût se dégrade par rapport au coût de la requête de la Question E, puisque l'ordre de l'index multiattributs n'est pas bien choisi, puisqu'on a intêret à avoir commandesG.codeclient trié, les autres attributs étant non important.

Question H:

```
[ ]: ALTER TABLE clientsH ADD PRIMARY KEY (codeclient, contact);
     CREATE INDEX index_sec1 ON commandesH (codeclient);

     EXPLAIN SELECT co.nocomm, co.datecomm
     FROM clientsH cl, commandesH co
     WHERE cl.codeclient = co.codeclient
     AND cl.contact = 'Thomas Hardy' ;
     SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	index	PRIMARY	PRIMARY	142	*NULL*	91	10.00	Using where; Using index
1	SIMPLE	co	*NULL*	ref	index_sec1	index_sec1	21	nasmanea.cl.codeclient	9	100.00	*NULL*

2 rows (0.001 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	39.051810

1 row (0.002 s) Edit, Export

Interpretation: On utilise l'index secondaire sur commandesH.codeclient pour accéder à la table commandesH.

Encore une fois, le choix de l'ordre de l'index multiattribut n'est pas optimal, on devrait avoir clientH.contact comme premier attribut pour construire l'index. Le coût est élevé.

Question I:

[ ]:
```
ALTER TABLE clientsI ADD PRIMARY KEY (contact, codeclient);
CREATE INDEX index_sec1 ON commandesI (codeclient);

EXPLAIN SELECT co.nocomm, co.datecomm
FROM clientsI cl, commandesI co
WHERE cl.codeclient = co.codeclient
AND cl.contact = 'Thomas Hardy' ;
SHOW STATUS LIKE 'last_query_cost';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cl	*NULL*	ref	PRIMARY	PRIMARY	122	const	1	100.00	Using index
1	SIMPLE	co	*NULL*	ref	index_sec1	index_sec1	21	nasmanea.cl.codeclient	9	100.00	*NULL*

2 rows (0.001 s) Edit, Warnings, Export

```
SHOW STATUS LIKE 'last_query_cost'
```

Variable_name	Value
Last_query_cost	3.613045

1 row (0.002 s) Edit, Export

Interpretation: On utilise l'index secondaire sur commandesH.codeclient pour accéder à la table commandesH.

Contrairement à la Question H, le choix de l'ordre de l'index multiattribut est bon, d'où le coût diminue.

1.3 EXERCICE 3

Requête 1:

[ ]:
```
CREATE INDEX index_sec1 ON clients1 (codeclient, societe, ville, pays);
CREATE INDEX index_sec2 ON commandes1 (codeclient, datecomm);

EXPLAIN SELECT cl.codeclient, cl.societe, cl.ville, cl.pays
FROM clients1 cl, commandes1 co
WHERE cl.codeclient = co.codeclient
AND YEAR(co.datecomm) = 1997;
SHOW STATUS LIKE 'last_query_cost';
```

Coût : 152 (7564 Sans Optimisation) On est dans les même conditions de l' Exercice 2, donc on a choisi de suivre la même démarche de la Question F qui était la meilleure selon nos conclusions.

Requête 2:

[ ]:
```
ALTER TABLE produits1 ADD PRIMARY KEY (prixunit, refprod, nom);

EXPLAIN SELECT refprod, nom, prixunit
FROM produits1
WHERE prixunit BETWEEN 80 AND 100;
SHOW STATUS LIKE 'last_query_cost';
```

Coût : 2.87 (7.94 Sans Optimisation) De même, comme la question précédente, a part l'utilisation de l'index primaire au lieu de l'index secondaire (On trouve un coût de 2.92 en utilisant l'index secondaire)

Requête 3:

[ ]:
```
EXPLAIN Select p.refprod, p.nom, c.nom, f.contact
FROM produits1 p, categories1 c, fournisseurs1 f
WHERE  p.nocat = c.nocat
AND p.nofourn = f.nofourn
AND p.indisp = 0
ORDER BY c.nom ASC;
SHOW STATUS LIKE 'last_query_cost';
```

Coût : 59 Experimentalement, tous les schémas d'optimisation qu'on a essayé sont moins perfromants que le schéma sans optimisation.

Requête 4:

[ ]:
```
CREATE INDEX index_sec1 ON clients1 (pays, codeclient, societe);
CREATE INDEX index_sec2 ON detailcomm1 (nocomm, refprod);
CREATE INDEX index_sec3 ON produits1 (refprod, nom, prixunit);

EXPLAIN Select det.nocomm, p.nom, p.prixunit, e.nom, cl.societe, m.nom
FROM detailcomm1 det, commandes1 co, clients1 cl, produits1 p , messagers1 m,␣
 ↪employes1 e
WHERE p.refprod = det.refprod
AND det.nocomm = co.nocomm
AND co.noemp = e.noemp
AND co.codeclient = cl.codeclient
AND co.nomsgr = m.nomsgr
AND cl.pays = 'Italie';
SHOW STATUS LIKE 'last_query_cost';
```

Coût : 961 (3 722 884 Sans Optimisation !) Encore une fois, on a comparé différents schémas, celui qu'on a utilisé dans l'Exercice 2 Question F, était le meilleur. (C'était le premier qu'on a testé)