```
/* Correction du TD sur les transactions */
----------------
-- EXERCICE 1 --
----------------

-- question 1.1
insert into T values (4,4);
insert into T values ('a','b');
--ERROR:  invalid input syntax for type integer: "a"
--LINE 1: insert into T values ('a','b');

/*
 seul le second insert est annulé
 Comme on n'a pas défini de transaction, chaque insert est considéré comme 1 transaction.
 Remarque : ce n'est pas demandé mais si vous mettez ces 2 lignes dans une transaction
 (i.e. à l'intérieur d'un begin; ... commit;) le premier insert est également annulé.
*/
begin;
insert into T values (5,5);
insert into T values (6,6);
select * from T;
/*
 a | b
---+---
 0 | 0
 2 | 3
 0 | 1
 4 | 4
 5 | 5
 6 | 6
*/

rollback;
select * from T;
/*
 a | b
---+---
 0 | 0
 2 | 3
 0 | 1
 4 | 4
 */
/*
 Ici on a défini une transaction qui commence avec l'instruction begin;
 le rollback annule donc les 2 insertions (une transaction est atomique)).
 L'instruction rollback marque la fin de la transaction.
*/
```

```
-- Question 1.2

/*
  La session 1 voit les modifications faites par la session 2 dès que celle-ci a fait un commit (validation).
  ça confirme bien le mode de fonctionnement READ COMMITTED.
  On peut aussi remarquer que la transaction de la session 1 fait 3 instructions select identiques et
  n'obtient pas 3 fois le même résultat. Donc elle n'est pas vraiment isolée.
  En particulier, elle n'est pas dans un mode REPEATABLE READ
*/

-- Question 1.3
-- session 1
begin;
-- session 2 :
begin ;
-- session 1
update T set a=a+1;
select * from T;
/*
 a | b
---+---
 2 | 0
 4 | 3
 2 | 1
 6 | 4
*/
-- session 2 :
insert into T values(3,7);
select * from t;
/*
 a | b
---+---
 1 | 0
 3 | 3
 1 | 1
 5 | 4
 3 | 7
 */
delete from t where a=2;
-- aucune ligne supprimée
delete from t where a=1;
-- attente (session 1 a le verrou)
-- session 1 :
commit ;
--> ça débloque session 2
-- session 2 :
-- son second delete ne supprime aucune ligne
```

```
commit ;
/* Conclusion : cet ordonnancement n'est pas sériable, parce que pas équivalent à une exécution en série
On compare avec les exécutions en série possibles :
--1- si on fait session1 avant session2, on supprime les lignes tq a=2
--2- si on fait session2 avant session1 on supprime les lignes tq a=1
Ici, aucune instruction delete ne supprime de ligne.
*/

-- Question 1.4
-- session 1
begin ;
-- session 2 :
begin ;
-- session 1 :
update T set a=3 where a=2;
-- 2 lignes modifiées
-- session 2 :
update T set b=2 where b=3;
-- 1 ligne modifiée
select * from t;
/*
 a | b
---+---
 2 | 0
 2 | 1
 6 | 4
 3 | 7
 4 | 2
*/
-- session 1 :
select * from t;
/*
a | b
---+---
 4 | 3
 6 | 4
 3 | 7
 3 | 0
 3 | 1
*/

-- session 2 :
update T set a=3 where a=2;
--> attente, session1 a le verrou
-- session 1
update T set b=2 where b=3;
--> attente donc deadlock détecté
/*
```

```
ERROR:  deadlock detected
DETAIL:  Process 18956 waits for ShareLock on transaction 659; blocked by process 18978.
Process 18978 waits for ShareLock on transaction 658; blocked by process 18956.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,12) in relation "t"
*/
--> ça libère la session2
-- 2 lignes modifiées par la session 2
-- session 1 :
commit ;
--> répond rollback
-- session 2 :
commit ;
-- table après question 1.4 :
/*
 a | b
---+---
 6 | 4
 3 | 7
 4 | 2
 3 | 0
 3 | 1

Résumé :
dead lock lorsque session1 fait son second update.
session 1 pose un verrou sur 2 lignes (2,0) et (2,1) puis fait son update
session 2 pose un verrou sur 1 ligne (4,3) puis fait son update
session 2 attend pour poser un verrou sur des lignes bloquées par session1
session 1 attend pour poser un verrou sur une ligne bloquée par session 2 ==> dead lock
La session 1 est avortée (rollback) et la session 2 continue.
*/

-- Question 1.5
-- Le select for update permet d'empêcher les autres transactions de modifier les lignes lues par le select

-- session 1
begin ;
-- session 2 :
begin ;
-- session 1 :
select * from t where b=7 for update;
/*
 a | b
---+---
 3 | 7
*/
-- session 2 :
update t set b=10 where b=0 ;
```

```
-- 1 ligne modifiée
select * from t;
/*
 a | b
---+----
 6 |  4
 3 |  7
 4 |  2
 3 |  1
 3 | 10
*/
update t set a=5 where a=3 ;
-- en attente, session 1 a le verrou
-- session 1 :
update t set b=6 where b=7 ;
commit ;
--> ça libère la session 2
-- session 2 :
-- le update modifie 3 lignes
commit ;
-- la session 2 a pu faire le 1er update donc le verrou posé par le select ... for update
-- n'est bloquant que pour les modifications des lignes sélectionnées
-- Table après la question 1.5 :
/*
select * from t;
 a | b
---+----
 6 |  4
 4 |  2
 5 |  6
 5 |  1
 5 | 10
*/

-- Question 1.6

/*
 La session 2 est avortée parce qu'elle veut écrire des lignes qui ont été modifiées par session1.
 C'est pour éviter un ordonnancement comme celui de la question 1.3
 Après question 1.6 :
  a | b
---+----
 4 |  2
 5 |  6
 5 |  1
 5 | 10
 5 |  4
*/
```

```
-- Question 1.7
/*
     Comme la session 1 est annulée (rollback), il n'y a plus de conflit avec la session 2.
     Donc la session 2 termine normalement sans être avortée (c'est la différence avec 1.6)
*/

-- Question 1.8
/*
ici on a résolu le problème des lectures non reproductibles.
La session 2 fait plusieurs fois le même select et obtient toujours le même résultat,
indépendamment de la session 1. La session 2 est bien isolée tout au long de son exécution (pas comme Q1.2)
*/

----------------
-- EXERCICE 2 --
----------------
/*
Dans la norme SQL, on peut activer/désactiver les contraintes,
On peut aussi attendre la fin d'une transaction pour vérifier les contraintes (on dit qu'elles sont "différées")

Avec Postgresql, seules les contraintes d'intégrité référentielles (foreign key) peuvent être différées.
*/

-- si on essaie de modifier les clés avec un update :
update Ecrivain set pid = pid+1, influence=influence+1;
-- ERROR:  duplicate key value violates unique constraint "ecrivain_pkey"
-- DETAIL:  Key (pid)=(202) already exists.

/*
 on en déduit que la contrainte de clé primaire est vérifiée à chaque ligne modifiée.
 Il faut donc modifier les pid par ordre décroissant,
 pour pas qu'il y ait de chevauchement entre les anciennes et nouvelles valeurs.
 Remarque : à partir du moment où la clé primaire est modifiée, on peut aussi avoir une erreur sur la clé étrangère
 lorsqu'elle est vérifiée à chaque ligne.
*/


/*
1ère solution :
   on diffère la contrainte de clé étrangère,
   on modifie les lignes (colonnes pid et influence) par ordre décroissant de pid
   pour éviter que la contrainte de clé primaires soit invalidée.
   Comme la contrainte de clé étrangère est vérifiée à la fin de la transaction, on n'a pas d'erreur.
*/
alter table Ecrivain alter constraint ecrivain_influence_fkey deferrable initially deferred;

DO $$
```

```
DECLARE
  r record ;
BEGIN
  for r in (select * from Ecrivain order by pid desc) loop
    update Ecrivain set pid = pid+1, influence=influence+1
    where r.pid = pid ;
  end loop ;
END ;
$$ LANGUAGE plpgsql;

/*
2ème solution :
    on met un "on update cascade" sur la contrainte de clé étrangère,
    on modifie les lignes (colonnes pid uniquement) par ordre décroissant de pid.
    Les clé étrangères sont modifiées automatiquement en fonction des modifications de la clé primaires.
*/
alter table Ecrivain drop constraint ecrivain_influence_fkey,
 add constraint ecrivain_influence_fkey foreign key(influence) references Ecrivain on update cascade;

-- on ne modifie pas la colonne influence dans le programme, juste pid (influence est modifiée grâce au CASCADE)
DO $$
DECLARE
  r record ;
BEGIN
  for r in (select * from Ecrivain order by pid desc) loop
    update Ecrivain set pid = pid+1
    where r.pid = pid ;
  end loop ;
END ;
$$ LANGUAGE plpgsql;

/*
 bien sûr, dans un SGBD où les contraintes de clé primaires sont aussi différables, il suffit
 de différer la clé primaire et de mettre un cascade sur la clé étrangère.
 Cette troisième solution permet de modifier les clés avec l'instruction update :
     update Ecrivain set pid = pid+1, influence=influence+1;
 Mais Postgresql refuse de différer les clé primaires :
alter table Ecrivain alter constraint ecrivain_pkey deferrable initially deferred;
-->ERROR:  constraint "ecrivain_pkey" of relation "ecrivain" is not a foreign key constraint
*/
```