

# Transactions (Partie I)

G. RASCHIA

dernière mise à jour : le 20 mars 2026

## 1 Sur quelques propriétés des exécutions concurrentes

1. Pour chacune des histoires suivantes, déterminer son type<sup>1</sup> parmi *réparable* (ou recouvrable) (REP), *prévention des annulations en chaîne* (PAC), *rigoureux*<sup>2</sup> (SS2PL), *sérialisable par conflit* (CSER).
  - (a)  $S_1 : w_1(a); w_2(b); r_1(a); c_1; r_2(a); c_2; w_3(b); c_3$
  - (b)  $S_2 : w_1(a); r_2(b); r_2(a); r_1(b); c_1; w_2(b); c_2$
  - (c)  $S_3 : w_1(a); r_2(b); r_2(a); r_1(a); c_2; w_1(b); c_1$
  - (d)  $S_4 : w_1(a); w_3(b); c_1; r_2(a); r_3(b); w_3(a); c_3; w_2(b); c_2$

### Solution :

En préambule, on peut établir les chaînes d'inclusion  $SS2PL \subset PAC \subset REP$ , ainsi que  $SS2PL \subset CSER$ . En revanche, les ensembles PAC/REP et CSER sont incomparables entre eux.

En effet, un plan 2PL rigoureux (SS2PL) empêche la formation de cycles dans un graphe de sérialisabilité (preuve à fournir, en exercice complémentaire). Donc  $SS2PL \subset CSER$ .

En outre, un plan 2PL rigoureux (SS2PL) ne libère les verrous qu'au commit de la transaction. Cela implique qu'aucune autre transaction ne peut lire ou écrire l'objet avant que la transaction qui possède la ressource ne soit terminée. C'est la condition requise pour la propriété PAC. Donc  $SS2PL \subset PAC$ . Enfin, nous savons déjà que  $PAC \subset REP$  puisqu'un plan réparable n'exige que d'ordonner les commits  $c_i <_P c_j$  en fonction des dépendances de type  $w_i \dots r_j$ .

- (a)  $S_1$  est *rigoureuse* :

$$S_1 : x_1(a); w_1(a); x_2(b); w_2(b); r_1(a); \ell_1(a); c_1; \\ s_2(a); r_2(a); \ell_2(a); \ell_2(b); c_2; x_3(b); w_3(b); \ell_3(b); c_3$$

Elle est donc SS2PL/CSER/PAC/REP.

- (b)  $S_2$  est CSER car le graphe de sérialisabilité ne comporte qu'un arc  $T_1 \rightarrow T_2$  suite à l'existence des 2 paires d'actions conflictuelles :  $(w_1(a), r_2(a))$  et  $(r_1(b), w_2(b))$ . En outre,  $S_2$  est réparable (REP) car  $T_2$ , qui lit  $T_1$  sur  $a$ , ne valide qu'après  $T_1$ . Elle n'est pas PAC car  $T_2$  lit  $a$  avant que  $T_1$  ne soit définitivement validée. Par conséquent  $S_2$  n'est pas rigoureusement 2PL (SS2PL) non plus.

---

1. Une histoire admet plusieurs types.

2. Les plans rigoureux (*Strong Strict 2PL*) sont des plans 2PL pour lesquels les verrous sont libérés à la validation de la transaction.

- (c)  $S_3$  ne vérifie aucune des propriétés ! Elle n'est en particulier, pas REP étant donné que  $T_2$  lit  $a$  ( $r_2(a)$ ) précédemment écrit par  $T_1$  ( $w_1(a)$ ), et que  $T_2$  est validée ( $c_2$ ) avant qu'une décision définitive ne soit prise pour  $T_1$ , et donc avec le risque que  $T_1$  soit rejetée. Par ailleurs,  $S_3$  n'est pas conflit-sérialisable (CSER) car les paires d'action conflictuelles ( $w_1(a), r_2(a)$ ) et ( $r_2(b), w_1(b)$ ) induisent un cycle dans le graphe de sérialisation.
- (d)  $S_4$  n'est pas CSER, car il existe un cycle ( $T_2, T_3$ ) dans le graphe de sérialisabilité, d'après les paires d'actions ( $r_2(a), w_3(a)$ ), et ( $w_3(b), w_2(b)$ ). Ainsi,  $S_4$  ne peut être STR.
- $S_4$  est en revanche REP/PAC. En effet,  $T_2$  lit  $T_1$  sur  $a$ ; or  $T_1$  est validée avant la lecture  $r_2(a)$ . Donc le plan  $S_4$  est bien PAC.

## 2 SI, représentant des protocoles MVCC

On considère l'histoire suivante :

$$S : r_1(a); w_2(a); w_3(b); w_2(b); c_2; r_3(a); c_3; r_1(b); c_1$$

On suppose que l'horodatage des transactions est donné par leur index ( $T_i$  commence au temps  $i$ ). Plus généralement, le plan  $S$  commence au temps 1, et chaque action incrémente le compteur de temps. Il y a donc 9 instants dans l'exécution de  $S$ .

1. Rappeler la séquence qui compose chacune des 3 transactions jouées dans le plan  $S$ .

**Solution :**

$$\begin{aligned} T_1 &= r_1(a); r_1(b); c_1 \\ T_2 &= w_2(a); w_2(b); c_2 \\ T_3 &= w_3(b); r_3(a); c_3 \end{aligned}$$

Chacune de ces 3 séquences d'actions est immuable. Seul un entrelacement des actions de plusieurs transactions est possible dans une exécution concurrente.

2. Déterminer le comportement de l'exécution du plan  $S$  avec SI (*Snapshot Isolation*) selon la règle FUW<sup>3</sup> (*First Updater Wins*).

**Solution :**

On suppose que tous les objets ont une version validée au temps 0. L'exécution de  $S$  en suivant SI selon la règle FUW (*First Updater Wins*) fournit la séquence suivante :

01.  $T_1$  lit  $a_0$  (la version au temps 0 de  $a$ );
02.  $T_2$  verrouille  $a$  pour une écriture au temps de la validation (une nouvelle version de  $a$  au **commit** de  $T_2$ );
03.  $T_3$  verrouille  $b$  pour préparer une nouvelle version de  $b$  au **commit** de  $T_3$ ;
- 04-05.  $T_2$  veut écrire  $b$ , qui est verrouillé par  $T_3$ .  $T_2$  est mise en attente du **commit** ou **rollback** de  $T_3$ ;
06.  $T_3$  lit  $a_0$ . Il n'y a en effet aucune version *validée* de  $a$  entre le temps 0 ( $a_0$ ) et le temps 3 ( $T_3$ );
07.  $T_3$  est validée ( $c_3$ ) et une nouvelle version  $b_7$  de  $b$  est créée. Le verrou  $b$  est relâché;

---

3. FUW correspond à « l'algorithme de P. Rigaux ».

- .  $T_2$ , en attente de  $b$ , peut maintenant progresser. Néanmoins, il existe une version validée  $b_7$  ultérieure au temps 2 (début de la transaction  $T_2$ ), donc  $T_2$  doit être annulée!

08-09.  $T_1$  lit  $b_0$  puis se termine ( $c_1$ ).

SI joue  $T_1$  sans difficulté, puisqu'elle ne comporte que des lectures.  $T_2$  perd contre  $T_3$  à la règle du FUW : elle est logiquement annulée.  $T_3$  réalise son écriture et produit une nouvelle version validée de  $b$ . Il est à noter que l'écriture de  $a$  par  $T_2$  (version  $a_5$ ) n'aura en fait jamais lieu.

3. Rejouer  $S$  selon SI, en suivant la règle FCW<sup>4</sup> (*First Committer Wins*).

**Solution :**

Quel que soit le mode (FCW ou FUW), toute transaction en lecture seule lit toujours une version validée des données et n'est jamais ni annulée, ni même retardée. C'est le cas de  $T_1$  qui lit  $a_0$  et  $b_0$ .

Dans le mode FCW, le point important est de noter l'ordre des `commit` de chaque transaction.

$T_2$  crée  $a_5$  et  $b_5$  au temps 5 de sa validation ( $c_2$ ).

La transaction  $T_3$  lit sans problème  $a_0$ , bien qu'il existe une version validée plus récente,  $a_5$ , provenant de l'écriture de  $T_2$ . Cette version  $a_5$  est en effet trop fraîche par rapport au début de la transaction  $T_3$  : elle ne fait donc pas partie du « cliché » de la base pour  $T_3$ .

Quoi qu'il en soit,  $T_3$  est finalement annulée car elle souhaite écrire  $b_7$  alors qu'il existe à l'instant 7 une version  $b_5$  ultérieure au début de  $T_3$ .

En conclusion, quel que soit le mode de SI (FUW ou FCW), l'une des transactions en concurrence ( $T_2$  ou  $T_3$ ) est annulée, et la transaction en lecture seule ( $T_1$ ) n'est jamais perturbée.

4. Dans la norme ANSI SQL, il existe quatre niveaux d'isolation des transactions. Reproduire le tableau croisé des anomalies corrigées en fonction de chacun des 4 niveaux. Imaginons que l'on ajoute un cinquième niveau **SNAPSHOT**<sup>5</sup> (de type SI), où se situerait-il dans ce tableau ?

**Solution :**

	dirty read	unrep. read	phantom read	lost update	write skew
1. UNCOMMITTED READ	oui	oui	oui	oui	?
2. COMMITED READ	non	oui	oui	oui/ non	?
3. REPEATABLE READ	non	non	oui	oui/ non	?
3.5 SNAPSHOT	non	non	non	non	oui
4. SERIALIZABLE	non	non	non	non	non

**SNAPSHOT** se retrouverait coincé entre **REPEATABLE READ** (lvl3) et **SERIALIZABLE** (lvl4). En effet, on accorde généralement plus d'importance aux lectures fantômes qu'à l'écriture biaisée (*write skew*); cela positionne le niveau **SNAPSHOT** un peu « au-dessus » de **REPEATABLE READ**, sans toutefois être parfaitement sérialisable.

4. le mode FCW de SI consiste à différer les écritures au moment du commit et, en cas de conflit, à privilégier la transaction qui valide la première.

5. En pratique, ce niveau est implémenté dans la plupart des SGBD.