

Module 7 – Gestion de la concurrence

Cours IGE487 – Modélisation de bases de données

Chargé de cours Tudor Antohi

Objectifs

- Concurrency
- Protocole 2PL
- Protocole TS
- Protocoles MVCC
- Granularité de verrous
- Autres défis

Introduction

On a discuté les conflits, les dangers de parallélisme transactionnels, les problèmes transactionnels.

Comment se protéger ?

Quels procédures on peut mettre en place pour assurer qu'on est recouvrables et c- sérialisable ?

Réponse : *en utilisant des protocoles de contrôle de concurrence (ensembles de règles) qui garantissent la sérialisabilité.*

Protocoles

- **verrouillage en deux phases 2PL**, utilise la technique de verrouillage des données pour empêcher plusieurs transactions d'accéder simultanément aux mêmes tuples. Les protocoles de verrouillage simple 2PL ont une surcharge élevée.
- **contrôle de la concurrence par horodatages TS**. Un identifiant unique pour chaque transaction, généré par le système. Les valeurs d'horodatage sont générées dans le même ordre de début de transaction.
- **contrôle de concurrence multiversion MVCC** qui utilisent plusieurs versions d'un élément de données. Un protocole multiversion étend l'ordre d'horodatage à l'ordre d'horodatage multiversion et un autre étend l'ordre d'horodatage au verrouillage 2PL.
- Il existe aussi le protocole optimiste basé sur le concept **d'isolement d'instantané SI et SSI**, qui peut utiliser des techniques similaires à celles proposées

2PL – verrouillage en 2 phases (*2 Phase lock*)

Un verrou est une variable associée à un élément de données qui décrit l'état de l'élément par rapport aux opérations possibles qui peuvent lui être appliquées.

1. **LOCK(X) = 1**, toute autre transaction est forcée d'attendre.

2. **UNLOCK (X)**, libère le X

- Une transaction T doit émettre l'opération **lock_item(X)** avant que toute opération **lire(X)** ou **écrire(X)** ne soit effectuée dans T.
- Un élément **verrouillé en lecture (*read-lock*)** est également appelé **verrouillé en partage (*shared locks*)** car d'autres transactions sont autorisées à lire l'élément, alors que un élément **verrouillé en écriture (*write-lock*)** est appelé à **verrouillage exclusif**.

2PL – verrouillage en 2 phases (*2 Phase lock*)

1. Une transaction T doit émettre l'opération **read_lock(X)** ou **write_lock(X)** avant que toute opération **read_item(X)** ne soit effectuée dans T.
2. Une transaction T doit émettre l'opération **write_lock(X)** avant que toute opération **write_item(X)** ne soit effectuée dans T.
3. Une transaction T doit émettre l'opération **unlock(X)** une fois que toutes les opérations **read_item(X)** et **write_item(X)** sont terminées dans T

2PL – mise a niveau et déclassification

- Il est possible qu'une transaction qui détient déjà un verrou sur l'élément X soit autorisée, sous certaines conditions, à convertir le verrou d'un état verrouillé à un autre.

Ex.: il est possible pour une transaction T d'émettre un **read_lock(X)** et ensuite de mettre à niveau le verrou en émettant une opération **write_lock(X)**. Si T est la seule transaction détenant un verrou en lecture sur X au moment où il émet l'opération **write_lock(X)**, le verrou peut être mis à niveau ; sinon, la transaction doit attendre.

- Il est également possible pour une transaction T d'émettre un **write_lock(X)** et ensuite de rétrograder le verrou en émettant une opération **read_lock(X)**.

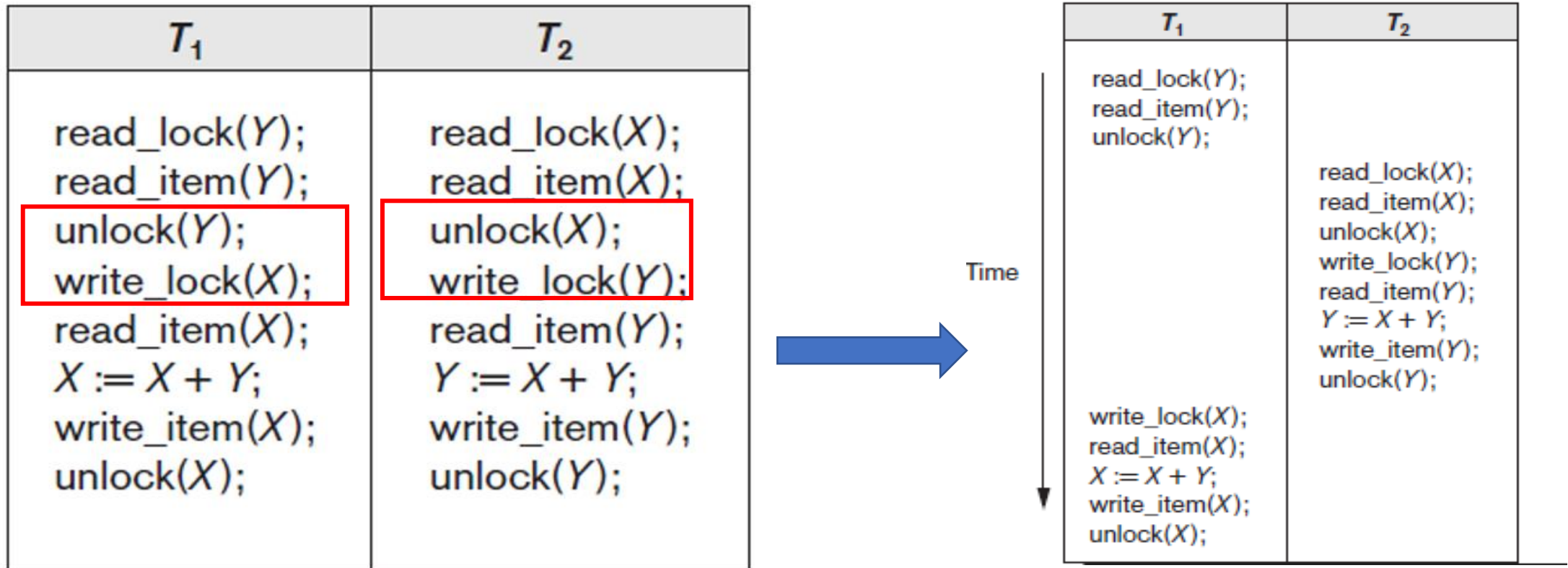
2PL – expansion et réduction

Une transaction suit le 2PL si toutes les opérations de verrouillage (*read_lock*, *write_lock*) précèdent la première opération de déverrouillage dans la transaction.

Une telle transaction peut être divisée en deux phases :

1. **une (première) phase d'expansion** ou de croissance, au cours de laquelle de nouveaux verrous sur des objets peuvent être acquis mais aucun ne peut être libéré ;
2. **et une phase seconde réduction**, au cours de laquelle les verrous existants peuvent être libérés mais aucun nouveau verrou ne peut être acquis.

Non 2PL – mauvais – conflit c-sérialisable



Valeurs initiales : $X=20$, $Y=30$ Résultat du calendrier séquentiel T_1 suivi de T_2 : $X=50$, $Y=80$

Résultat du programme séquentiel T_2 suivi de T_1 : $X=70$, $Y=50$

Résultat de l'ordonnancement S : $X=50$, $Y=50$ (non sérialisable, aucune de 2 valeurs !)

2PL – bon , mais possible deadlock

T_1'	T_2'
read_lock(Y); read_item(Y);	read_lock(X); read_item(X);
write_lock(X); unlock(Y)	write_lock(Y); unlock(X)
read_item(X); $X := X + Y$; write_item(X); unlock(X);	read_item(Y); $Y := X + Y$; write_item(Y); unlock(Y);

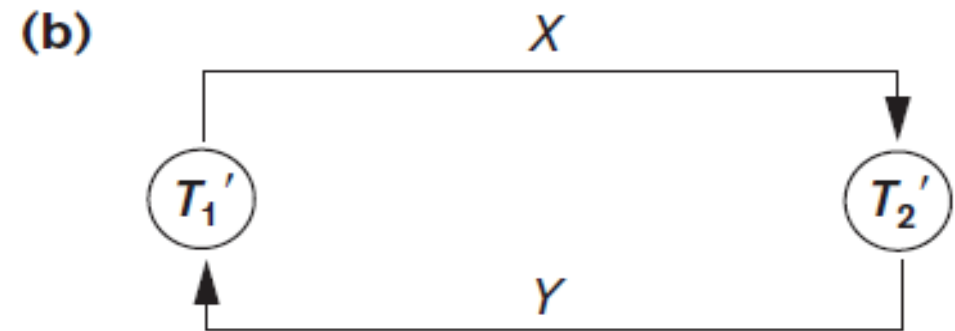
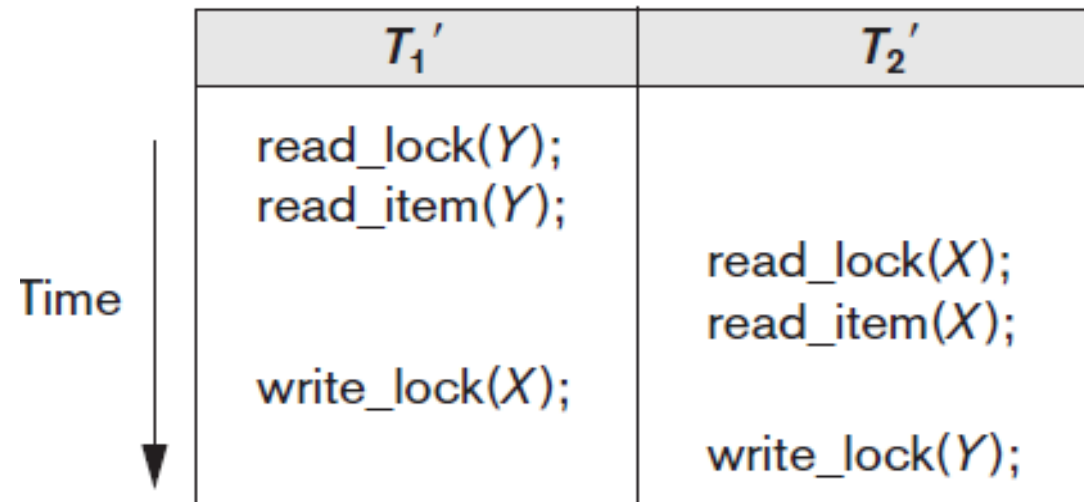
Le verrouillage en deux phases peut limiter la simultanéité qui peut se produire dans un programme car une transaction T peut ne pas être en mesure de libérer un élément X après l'avoir utilisé si T doit verrouiller un élément supplémentaire Y ultérieurement ;
ou, inversement, T doit verrouiller l'élément supplémentaire Y avant d'en avoir besoin pour pouvoir libérer X. Ainsi, **X reste verrouillé par T** jusqu'à ce que tous les éléments que la transaction doit lire ou écrire aient été verrouillés.

Versions de 2PL

1. Le **2PL conservateur** (ou 2PL statique) nécessite qu'une transaction **verrouille** tous les éléments auxquels elle accède **avant** que la transaction commence son exécution.
2. Le **2PL strict**, une transaction T **ne libère aucun de ses verrous exclusifs (en écriture) tant qu'elle n'a pas été validée ou abandonnée.** Par conséquent, aucune autre transaction ne peut lire ou écrire un élément écrit par T pendant cette période, ce qui conduit à un calendrier strict de récupération.
3. Le **2PL rigoureux**. Dans cette variante, une transaction T **ne libère** aucun de ses verrous (**exclusifs ou partagés**) avant d'avoir validé ou abandonné, plus facile à mettre en œuvre que le 2PL strict.

Deadlock (Interblocage)

L'interblocage se produit lorsque chaque transaction T dans un ensemble de deux transactions ou plus attend un élément qui est verrouillé par une autre transaction T' dans l'ensemble. Chaque transaction de l'ensemble est dans une file d'attente, attendant que l'une des autres transactions de l'ensemble libère le verrou sur un élément.



Règles correctives d'interblocage

Attendez-mourez (Wait-Die). Si $TS(T_i) < TS(T_j)$, alors (T_i plus ancien que T_j) **T_i est autorisé à attendre** ; sinon (**T_i plus jeune que T_j**) abandon T_i (**T_i meurt**) et redémarre plus tard avec le même horodatage.

Blessure-attendre (Wound-wait). Si $TS(T_i) < TS(T_j)$, alors (T_i plus ancien que T_j) abandon T_j (**T_i tue T_j**) et le redémarrer plus tard avec le même horodatage ; sinon (**T_i plus jeune que T_j**) T_i est autorisé à attendre.

Attente prudente (Cautious-Waiting). Si T_j n'est pas bloqué (n'attend pas un autre élément verrouillé), alors **T_i est bloqué et autorisé à attendre** ; sinon abandonne de T_i .

Obs.: *les mécanismes sont implantés par les bases de données*

Horodatage (Timestamp)

Un horodatage peut être considéré comme l'heure de début de la transaction. Nous désignerons l'horodatage de la transaction T par **TS(T)**.

L'idée est d'appliquer **l'ordre de transactions en fonction de leurs horodatages**.

Un ordonnancement **des transactions est alors sérialisable**, et le seul programme en série équivalent autorise **les transactions dans l'ordre de leurs valeurs d'horodatage**.

C'est ce qu'on appelle **l'ordre d'horodatage (TO)**.

Lire_TS (*Read_TS*) et Écrire_TS (*Write_TS*)

1. **lire_TS(X)**. L'horodatage de lecture de l'élément X est le plus grand horodatage parmi tous les horodatages des transactions qui ont lu avec succès l'élément X, c'est-à-dire **read_TS(X) = TS(T)**, où **T est la transaction la plus récente qui a lu X** avec succès.
2. **ecrire_TS(X)**. L'horodatage d'écriture de l'élément X est le plus grand de tous les horodatages des transactions qui ont écrit avec succès l'élément X, c'est-à-dire, **write_TS(X) = TS(T)**, où **T est la plus jeune transaction qui a écrit X** avec succès.

Algorithme pour Write_Item(X)

1. Si $\text{read_TS}(X) > \text{TS}(T)$ ou si $\text{write_TS}(X) > \text{TS}(T)$, annuler T et rejeter l'opération. Cela devrait être fait parce **qu'une transaction plus jeune** avec un horodatage supérieur à TS (T), après T dans l'ordre d'horodatage - **a déjà lu ou écrit** la valeur de l'élément **X** avant que T n'ait eu la chance d'écrire X, violant ainsi ordre.
2. Si la condition de la partie (1) ne se produit pas, alors exécutez l'opération **write_item(X)** de T et mettre le **TS(T)** en X avec **Write_TS(X)**

Algorithme pour Read_Item(X)

1. Si $\text{write_TS}(X) > \text{TS}(T)$, annuler T. Cela devrait être fait parce qu'une transaction plus jeune a déjà écrit la valeur de l'élément X avant que T n'ait eu la chance de lire X.
2. Si $\text{write_TS}(X) \leq \text{TS}(T)$, exécutez l'opération $\text{read_item}(X)$ de T et définissez $\text{read_TS}(X) = \max(\text{TS}(T), \text{read_TS}(X))$.

MVCC

Le protocole a été proposé pour la première fois en 1978 dans la thèse de doctorat du MIT.

Les premières implémentations étaient Rdb/VMS et InterBase au DEC au début des années 1980.

→ Les deux étaient de Jim Starkey, co-fondateur de NuoDB.

→ DEC Rdb/VMS devient "Oracle Rdb"

→ InterBase était open source sous le nom de Firebird.



MVCC – problèmes de conception

- Protocole de contrôle de concurrence
- Stockage des versions
- Nettoyage des versions inutiles (Garbage collection)
- Gestion des index (juste savoir que c'est compliqué)
- Gestion de suppressions

MVCC – version horodatées

1. **lire_TS(Xi)** = le plus grand de tous les horodatages des transactions qui ont **lu avec succès la version Xi**.
2. **écrire_TS(Xi)** = l'horodatage de la transaction qui **écrit la valeur de la version Xi**.

Les deux acétates suivants présente les concepts de livre Elmasri.
Et après on va regarder une implantation concrete.

MVCC TS – règles de sérialisabilité - écriture

1. If transaction T issues a $\text{write_item}(X)$ operation, and version i of X has the highest $\text{write_TS}(X_i)$ of all versions of X that is also *less than or equal to* $\text{TS}(T)$, and $\text{read_TS}(X_i) > \text{TS}(T)$, then abort and roll back transaction T ; otherwise, create a new version X_j of X with $\text{read_TS}(X_j) = \text{write_TS}(X_j) = \text{TS}(T)$.

Si on execute le $\text{write_item}(X)$ et la version i de X a le plus élevé $\text{write_TS}(X_i)$ de toutes les versions de X égales ou inférieurs à $\text{TS}(T)$, et $\text{read_TS}(X_i) > \text{TS}(T)$, abandonner et annuler la transaction T ;

Sinon, créer une nouvelle version X_j de X avec $\text{read_TS}(X_j) = \text{write_TS}(X_j) = \text{TS}(T)$.

MVCC TS – règles de sérialisabilité - lecture

2. If transaction T issues a $\text{read_item}(X)$ operation, find the version i of X that has the highest $\text{write_TS}(X_i)$ of all versions of X that is also *less than or equal to* $\text{TS}(T)$; then return the value of X_i to transaction T , and set the value of $\text{read_TS}(X_i)$ to the larger of $\text{TS}(T)$ and the current $\text{read_TS}(X_i)$.

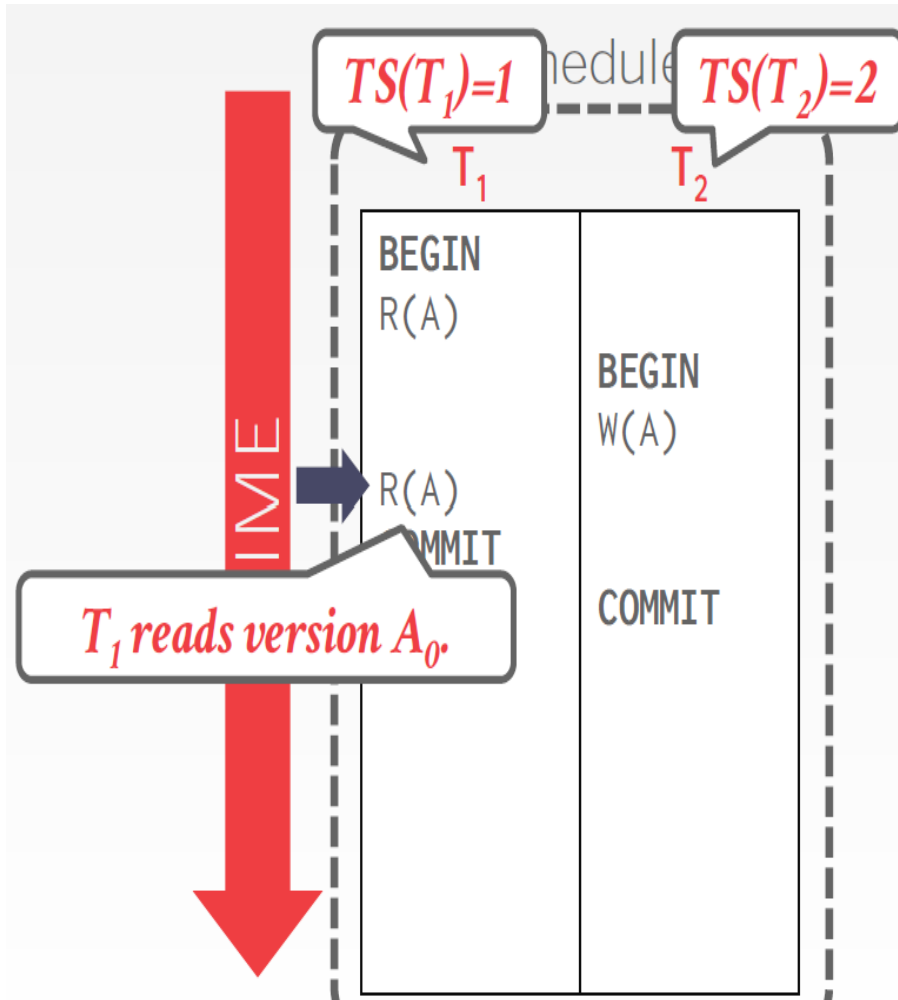
Si $\text{read_item}(X)$, cherchez la version i de X qui a le $\text{write_TS}(X_i)$ le plus élevé de toutes les versions de X inférieures ou égales à $\text{TS}(T)$;

puis renvoyez la valeur de X_i à la transaction T , et définissez la valeur de $\text{read_TS}(X_i)$ sur le $\max(\text{TS}(T), \text{read_TS}(X_i))$ actuel.

MVCC TS – implantation exemple

- Le SGBD ajoute un champ **Read-TS** supplémentaire dans l'en-tête du tuple pour garder une trace de l'**horodatage de la dernière transaction qui l'a lu**.
- Pour les lectures, une transaction est autorisée à lire la version si le verrou n'est pas défini et que son identifiant de transaction (Tid) est situé entre **Begin-TS** et **End-TS**. Les verrous ne sont pas nécessaires pour les opérations de lecture.
- Pour les écritures, une transaction crée une nouvelle version si aucune autre transaction ne détient de verrou et si Txn-Id est supérieur à Read-TS. L'opération d'écriture exécute MAJ sur le champ Txn-Id qui fournit un verrou sur ce tuple de données.
- **Après avoir créé une nouvelle version, il met à jour le champ End-TS** avec l'horodatage de la transaction.

EX.1 MVCC TO



Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

La première lecture de A en T_1 met à jour la table de versions avec la version A_0

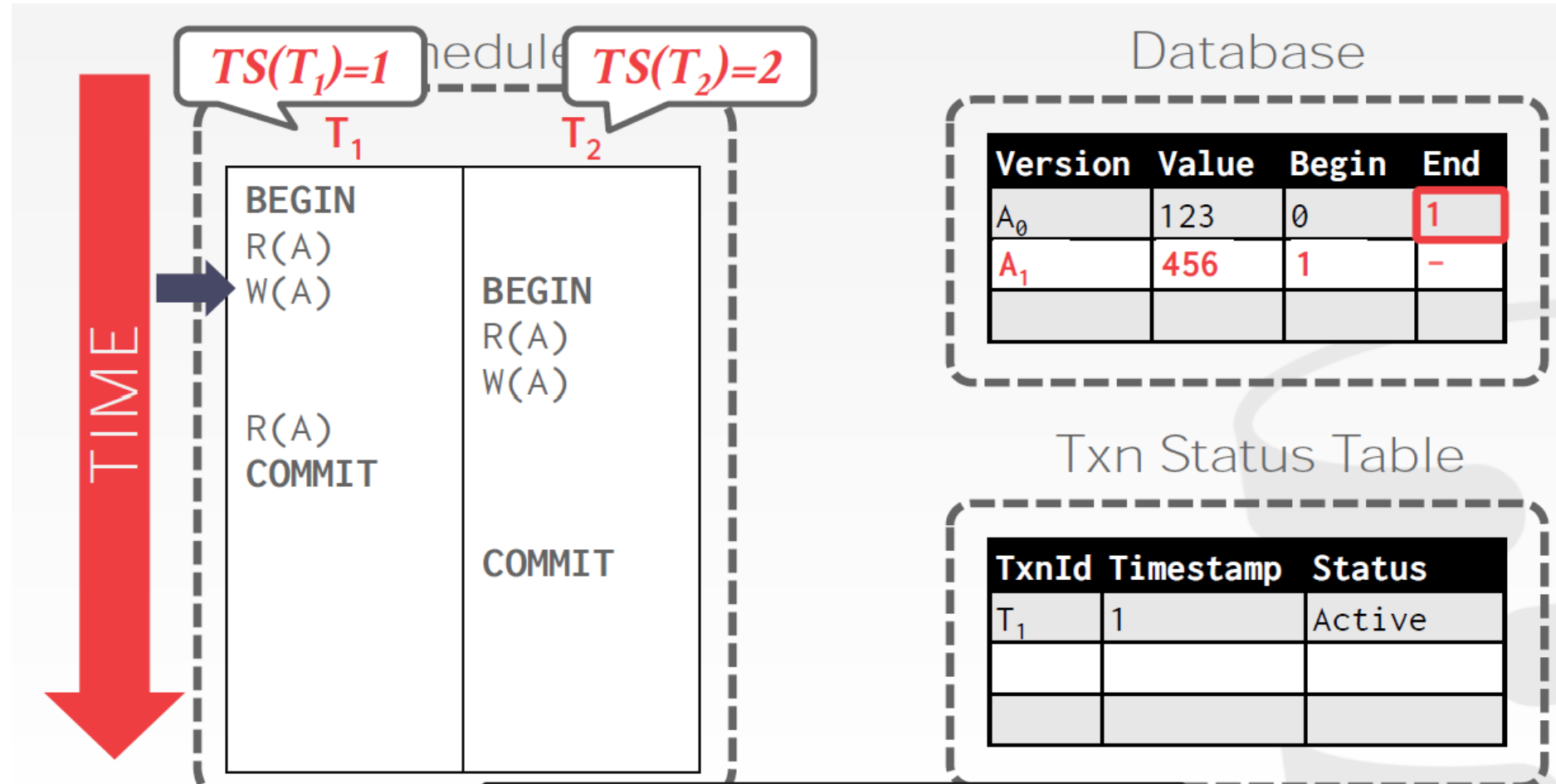
L'écriture $WA(A)$ en T_2 va écrire la deuxième version de A en A_1 et va mettre la fin de A_0 .

Txn Status Table

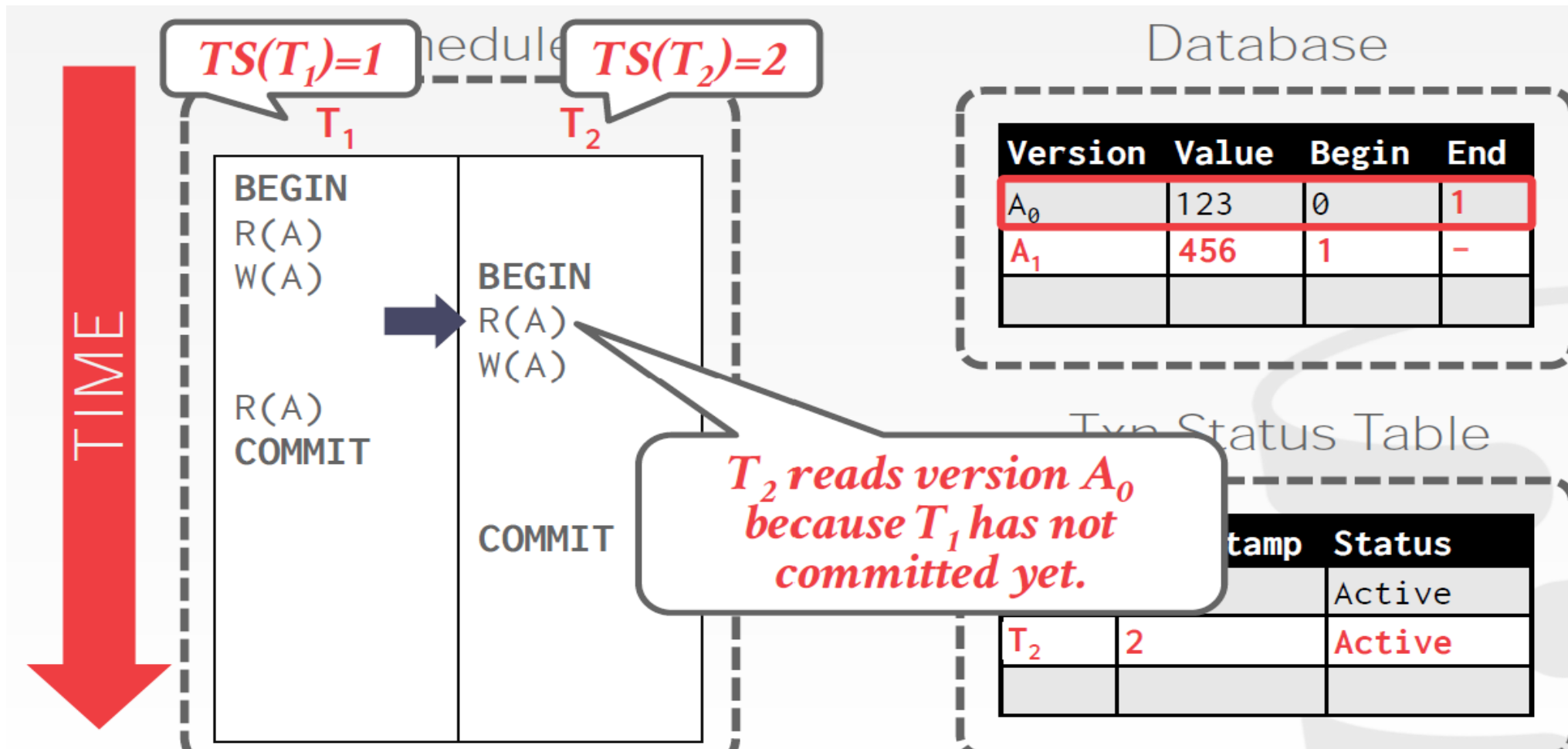
TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

La deuxième lecture de A en T_1 va lire toujours A_0 , parce le timestamp de T_1 est inférieur à T_2 selon les règles.

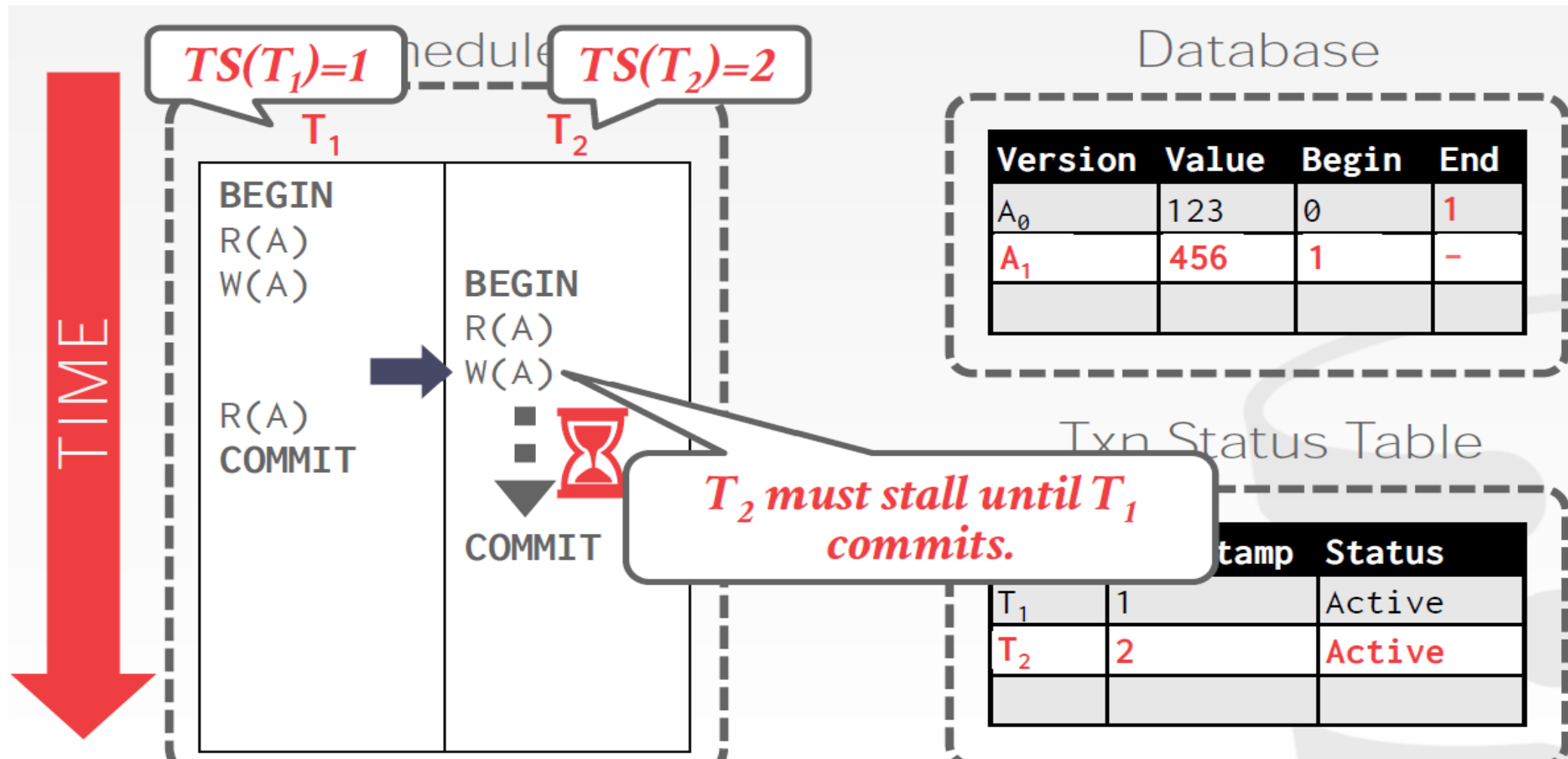
Ex.2 MVCC TO (1)



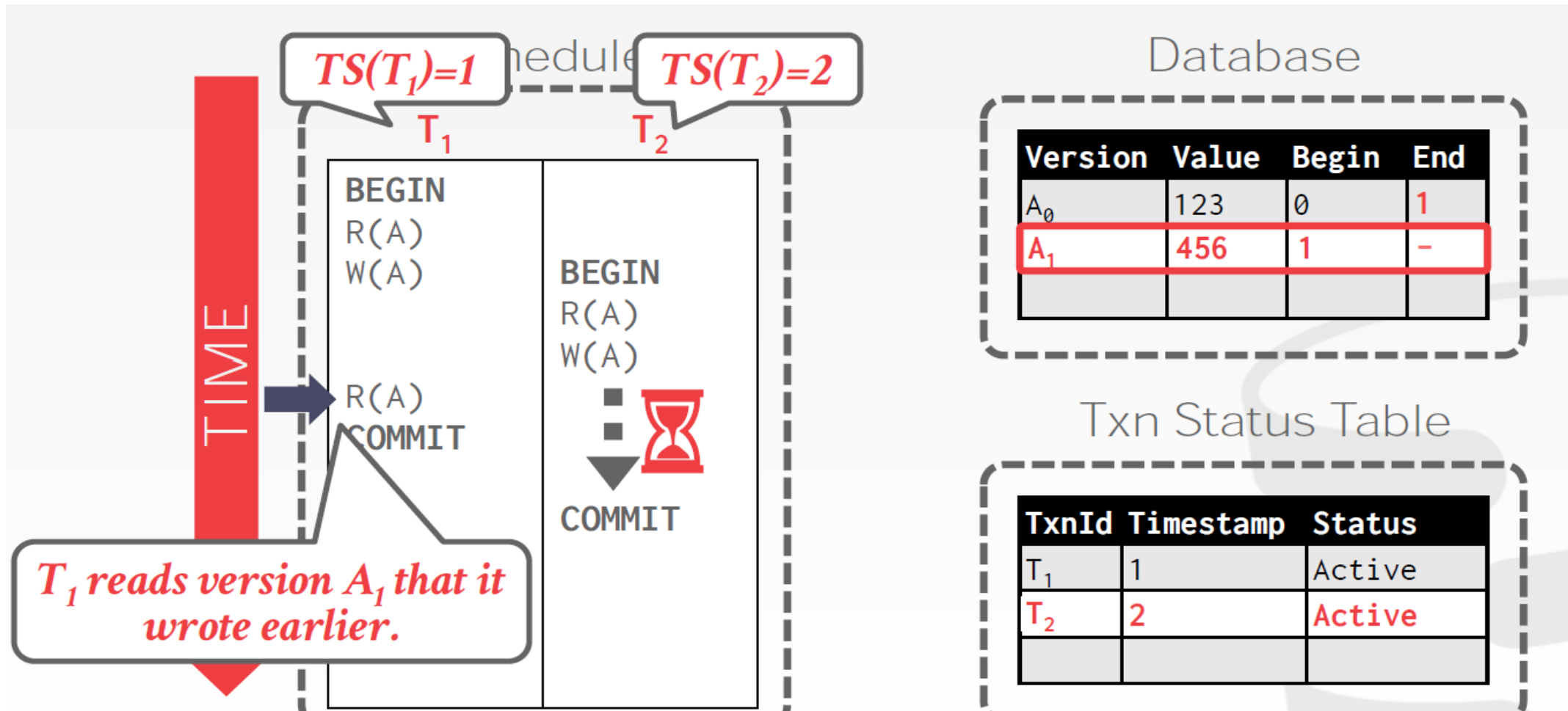
Ex.2 MVCC TO (2)



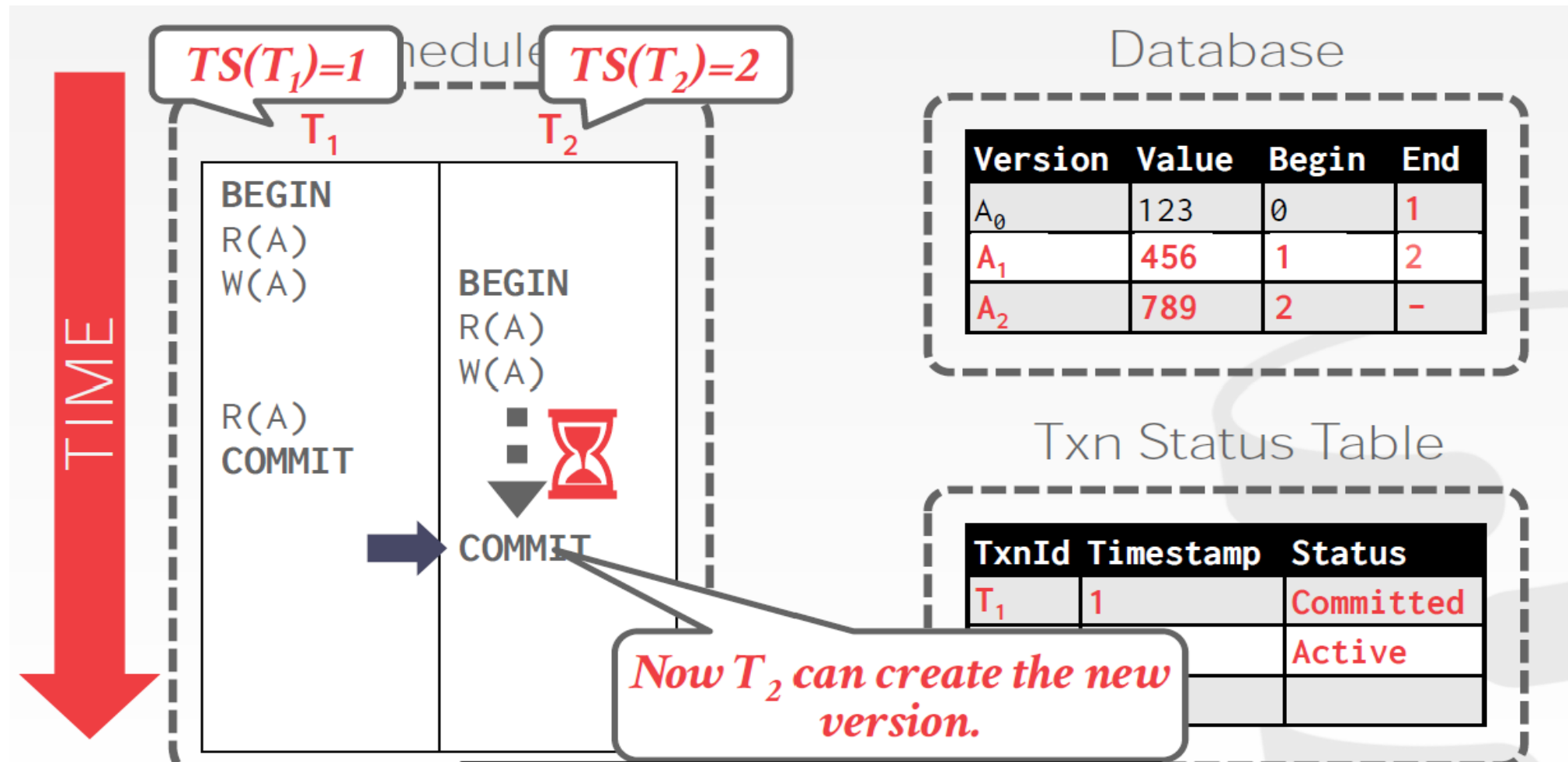
Ex.2 MVCC TO (3) – au lieu d'annuler...



Ex.2 MVCC (4) TO



Ex.2 MVCC TO (5)



MVCC – règles de verrouillage type 2PL

Dans ce schéma de verrouillage à plusieurs modes, il existe trois modes de verrouillage pour un élément (lecture, écriture et certification) au lieu des deux modes (lecture, écriture) décrits précédemment. (*Elmasri*)

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

MVCC – 2PL – sérialisation - Elmasri

L'idée derrière MV 2PL est de **permettre à d'autres transactions T de lire un élément X alors qu'une seule transaction T détient un verrou en écriture sur X.**

Ceci est accompli en autorisant deux versions pour chaque élément X ;

- **Une version, la version validée, doit toujours avoir été écrite par une transaction validée.**
- **La deuxième version locale X' peut être créée lorsqu'une transaction T acquiert un verrou en écriture sur X.**

D'autres transactions peuvent continuer à lire la version validée (COMMIT) de X pendant que T détient le verrou en écriture. La transaction T peut écrire sans validation la valeur de X' selon les besoins, sans affecter la valeur de la version validée X.

Une fois que T est prêt à valider, il faut obtenir un verrou de certification sur tous les éléments sur lesquels il détient actuellement des verrous en écriture avant de pouvoir commettre; il s'agit d'une autre forme de mise à niveau des verrous.

MVCC – 2PL – implémentation

Le SGBD ajoute un champ **Read-Cnt** supplémentaire à l'en-tête de chaque tuple qui agit comme un verrou partagé. Il s'agit d'un compteur 64 bits qui suit le nombre de transactions détenant actuellement ce verrou.

Pour les lectures, une transaction est autorisée à détenir le verrou de partage si Txn-Id est égal à zéro. Il effectue ensuite un **MAJ sur Read-Cnt** pour incrémenter le compteur.

Pour les écritures, une transaction est autorisée à détenir le verrou exclusif si Txn-Id et Read-Cnt sont à zéro. Le SGBD utilise Txn-Id et Read-Cnt ensemble comme verrou exclusif.

Lors de la validation (COMMIT), Read-Cnt et Txn-Id sont remis à zéro.

MV – 2PL Example (1)

TWO-PHASE LOCKING (MV2PL)

Thread #1

$T_{id}=10$

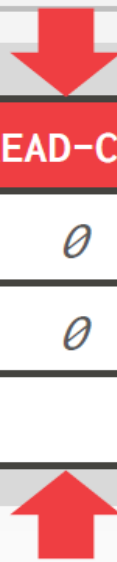


READ(A)



WRITE(B)

Txns use the tuple's *read-cnt* field as SHARED lock.
Use *txn-id* and *read-cnt* together as EXCLUSIVE lock.



	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	0	1	∞
B_1	0	0	1	∞

MV – 2PL Example (2)

TWO-PHASE LOCKING (MV2PL)

Thread #1

$T_{id}=10$



	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

Txns use the tuple's *read-cnt* field as SHARED lock. Use *txn-id* and *read-cnt* together as EXCLUSIVE lock.

If *txn-id* is zero, then the txn acquires the SHARED lock by incrementing the *read-cnt* field.

MV – 2PL Example (3)

TWO-PHASE LOCKING (MV2PL)

Thread #1

$T_{id}=10$



READ(A)



WRITE(B)



	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A ₁	0	1	1	∞
B ₁	10	1	1	∞

Txns use the tuple's **read-cnt** field as SHARED lock. Use **txn-id** and **read-cnt** together as EXCLUSIVE lock.

If **txn-id** is zero, then the txn acquires the SHARED lock by incrementing the **read-cnt** field.

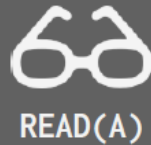
If both **txn-id** and **read-cnt** are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

MV – 2PL Example (4)

TWO-PHASE LOCKING (MV2PL)

Thread #1

$T_{id}=10$



READ(A)



WRITE(B)



	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A ₁	0	1	1	∞
B ₁	10	1	1	∞
B ₂	10	0	10	∞

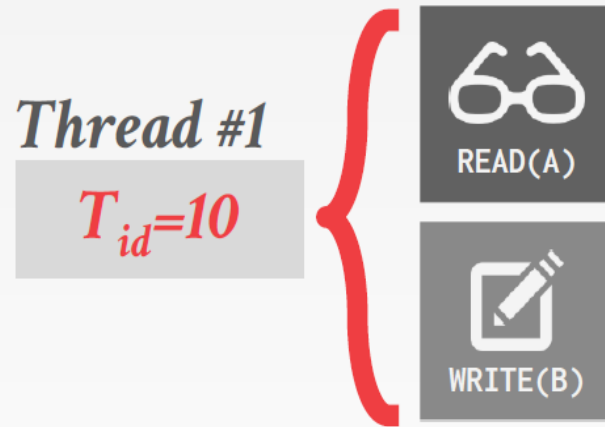
Txns use the tuple's *read-cnt* field as SHARED lock. Use *txn-id* and *read-cnt* together as EXCLUSIVE lock.

If *txn-id* is zero, then the txn acquires the SHARED lock by incrementing the *read-cnt* field.

If both *txn-id* and *read-cnt* are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

MV – 2PL Example (5)

TWO-PHASE LOCKING (MV2PL)



	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	0	1	∞
B_1	0	0	1	10
B_2	0	0	10	∞

Txns use the tuple's *read-cnt* field as SHARED lock. Use *txn-id* and *read-cnt* together as EXCLUSIVE lock.

If *txn-id* is zero, then the txn acquires the SHARED lock by incrementing the *read-cnt* field.

If both *txn-id* and *read-cnt* are zero, then txn acquires the EXCLUSIVE lock by setting both of them.

MVCC – OCC – Snapshot - Trois phases

- 1. Phase de lecture.** Une transaction peut lire les valeurs des éléments de données validés à partir de la base de données. Cependant, les mises à jour ne sont appliquées qu'aux copies locales (versions) des éléments de données conservés dans l'espace de travail de transaction.
- 2. Phase de validation.** La vérification est effectuée pour s'assurer que la sérialisabilité ne sera pas violée si les mises à jour sont appliquées à la base de données.
- 3. Phase d'écriture.** Si la phase de validation réussit, les mises à jour des transactions sont appliquées à la base de données ; sinon, les mises à jour sont ignorées et la transaction est redémarrée.

MVCC – OCC – Validation

La phase de validation de T_i vérifie l'une des conditions suivantes est vérifiée :

1. La transaction T_j termine sa phase d'écriture sur X avant que T_i commence sa phase de lecture sur X
2. T_i a commencé sa phase d'écriture une fois que T_j a terminé sa phase d'écriture, et le **read_set de T_i n'a aucun élément en commun** avec le **write_set de T_j**
3. Le **read_set et write_set de T_i n'ont aucun élément en commun** avec le **write_set de T_j** , et T_j termine sa phase de lecture avant que T_i termine sa phase de lecture.

Revue MVCC

Approche #1 : Commande d'horodatage

→ Attribuez des horodatages txns qui déterminent l'ordre de série.

Approche #2 : Contrôle de concurrence optimiste

→ Protocole triphasé de la dernière classe.

→ Utiliser l'espace de travail privé pour les nouvelles versions.

Approche #3 : Verrouillage en deux phases 2PL – way to go !

→ Les Txns acquièrent le verrou approprié sur la version physique avant de pouvoir lire/écrire un tuple logique.

Versionnement de stockage

Approche n° 1 : Stockage en ajout uniquement (Append Only)

→ Les nouvelles versions sont ajoutées au même espace table.

Approche n° 2 : Stockage des voyages dans le temps (Time Travel)

→ Les anciennes versions sont copiées dans une table séparée.

Approche n° 3 : Stockage delta

→ Les valeurs d'origine des attributs modifiés sont copiées dans un espace d'enregistrement delta séparé, dans plusieurs tables – fichiers séparées. En Oracle , plusieurs segments dans des tables temporaires.

Nettoyage

Approche n° 1 : niveau tuple (VACUUM)

→ Trouvez les anciennes versions en examinant directement les tuples. **Vacuum complet** – niveau physique, élimine complètement l'historique non-validé.

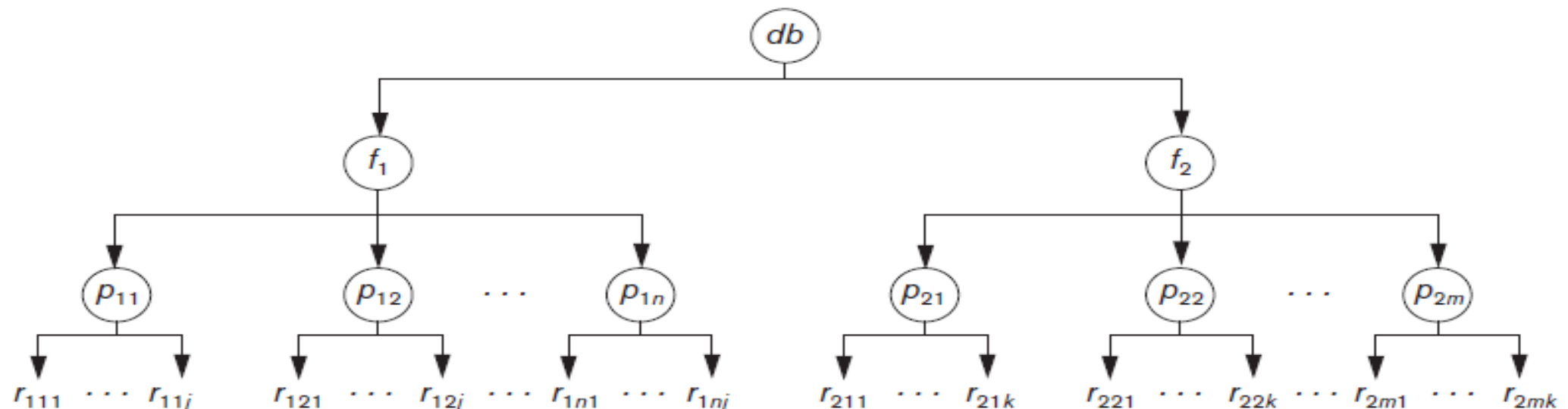
Approche n° 2 : au niveau de la transaction (TXN)

→ Les Txns gardent une trace de leurs anciennes versions afin que le SGBD n'ait pas à analyser les tuples pour déterminer la visibilité.

Granularité de verrouillage - Elmasri

Protocole de verrouillage multi-granularité permet le changement de granularité (taille de l'élément) en fonction de la transaction dans le but d'améliorer les performances du contrôle de la concurrence.

Pensez : base de données, table, rangées, colonnes, c'est rare quand le verrouillage BD respecte ce concept.



Implantations

MVCC IMPLEMENTATIONS

	<i>Protocol</i>	<i>Version Storage</i>	<i>Garbage Collection</i>	<i>Indexes</i>
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
HYRISE	MV-OCC	Append-Only	-	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical
<u>CMU's TBD</u>	MV-OCC	Delta	Txn-level	Logical