# *BASES*
# *DE*
# *DONNÉES RELATIONNELLES*
## *Approfondissement*

# Identificateurs

Mot commençant par une lettre et contenant des lettres, des chiffres ou le caractère « _ ».

Toute suite de caractères encadrée par le caractère « " ».

# Chaînes de caractères

Deux jeux de caractères sont disponibles :
Le jeu de caractères de base
Le jeu de caractères national

Expressions littérales :

| | |
|---|---|
| 'chaîne' | Utilise le jeu de caractères de base |
| N'chaîne' | Utilise le jeu de caractères national |

Codage en longueur fixe par remplissage avec des blancs :

| | |
|---|---|
| **CHAR(n)** | Utilise le jeu de caractères de base. |
| **NCHAR(n)** | Utilise le jeu de caractères national. |

Codage en longueur variable :

| | |
|---|---|
| **VARCHAR(n)** | Utilise le jeu de caractères de base. |
| **NVARCHAR(n)** | Utilise le jeu de caractères national. |



# Les nombres

Deux types de codage :

- Valeur exacte : Le type possède une précision et une échelle. La précision indique le nombre de chiffres significatifs. Une échelle égale à 0 indique que le nombre est un entier. Pour une échelle donnée S, la valeur d'un nombre est la valeur de l'entier formé des chiffres significatifs multipliée par $10^{-S}$.

- Valeur approximée : Codage avec une mantisse et un exposant. Le type possède une précision qui indique de nombre de chiffres binaires significatifs de la mantisse. La valeur du nombre est la valeur de la mantisse multipliée par $10^{\text{exposant}}$.

Types avec codage en valeur exacte :

| | |
|---|---|
| **INT, INTEGER** | La précision est soit binaire soit décimale et dépend de l'implémentation. |
| **SMALLINT** | La précision doit être plus petite ou égale à celle de **INTEGER**, dépend de l'implémentation. |
| **NUMERIC(p,s)** | La précision et l'échelle sont exactement **p** et **s**. |
| **DECIMAL(p,s)** | L'échelle est **s** et la précision est supérieur ou égale à **p**, dépend de l'implémentation. |

Types avec codage en valeur approximée :

| | |
|---|---|
| **FLOAT(p)** | La précision est binaire, supérieure ou égale à p |
| **REAL** | La précision dépend de l'implémentation. |
| **DOUBLE PRECISION** | La précision est supérieur à celle de **REAL**, dépend de l'implémentation. |

# Date, Temps et Intervalle

◆ **Les types :**

**DATE**
Contient les champs **YEAR**, **MONTH** et **DAY**

**TIMESTAMP[(p)] [WITH TIME ZONE]**
Contient les champs **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE**, **SECOND** et éventuellement **TIMEZONE_HOUR** et **TIMEZONE_MINUTE**. La précision p indique le nombre de chiffres significatifs pour les fractions de seconde (ex : 3 pour milliseconde, 6 pour microseconde, 9 pour nanoseconde), la valeur par défaut est 6.

**TIME[(p)] [WITH TIME ZONE]**
Contient les champs **HOUR**, **MINUTE**, **SECOND** et éventuellement **TIMEZONE_HOUR** et **TIMEZONE_MINUTE**. La précision p indique le nombre de chiffres significatifs pour les fractions de seconde.

**INTERVAL YEAR[(p)] TO MONTH**
Spécifie une durée en nombre d'années et de mois. La précision p indique le nombre de chiffres significatif pour le nombre d'années, la valeur par défaut est 2.

**INTERVAL DAY[(p)] TO SECOND[(q)]**
Spécifie une durée en nombre de jours, d'heures, de minutes et de secondes. La précision p indique le nombre de chiffres significatif pour le nombre de jours, la valeur par défaut est 2. La précision q indique le nombre de chiffres significatifs pour les fractions de secondes, la valeur par défaut est 6.



# Date, Temps et Intervalle

◆ **Les valeurs littérales :**

Syntaxe générale :

**NomDuType valeur [QualificatifIntervalle]**

Exemples :

```
DATE '2019-02-01'

TIMESTAMP '2019-02-01 10:30:10'
TIMESTAMP '2019-02-01 10:30:10 +1:30'

INTERVAL '10' YEAR
INTERVAL '-10' YEAR
INTERVAL '10' MONTH
INTERVAL '10-3' YEAR TO MONTH

INTERVAL '30' DAY
INTERVAL '-30' DAY
INTERVAL '300' DAY(3)
INTERVAL '10' HOUR
INTERVAL '-10' MINUTE
INTERVAL '30' SECOND
INTERVAL '12:10.222' MINUTE TO SECOND(3)
INTERVAL '4 5:12' DAY TO MINUTE
INTERVAL '4 5:12:10.222444' DAY TO SECOND
```

# Date, Temps et Intervalle

◆ **Arithmétique :**

| Opérande 1 | Opérateur | Opérande 2 | Résultat |
|---|---|---|---|
| DateTemps | - | DateTemps | Intervalle |
| DateTemps | + ou - | Intervalle | DateTemps |
| Intervalle | + | DateTemps | DateTemps |
| Intervalle | + ou - | Intervalle | Intervalle |
| Intervalle | * ou / | Nombre | Intervalle |
| Nombre | * | Intervalle | Intervalle |

◆ **Extraction d'un composant :**

```
EXTRACT(champ FROM valeur)
```

Où `champ` est l'un des mots clefs suivant :

**YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE**, **SECOND**,
**TIMEZONE_HOUR**, **TIMEZONE_MINUTE**

Exemples :

```
SELECT EXTRACT(YEAR FROM DateNais)
       AS AnneeNais
FROM Personnes ;

SELECT EXTRACT(DAY FROM DureeSejour)
       AS NbJours
FROM Sejours ;
```



# Date, Temps et Intervalle

◆ **Comparaison de périodes :**

```
période1 OVERLAPS période2
```

**où** `période1` et `période2` sont spécifiées par un des couples suivants :

```
(DateTemps_debut, DateTemps_fin)
```

Ou

```
(DateTemps_debut, intervalle)
```

Exemple :
Pour connaître les vols qui ont lieu en même temps

```
SELECT  V1.NumVol, V2.NumVol
FROM    Vols V1, Vols V2
WHERE   (V1.DateD,V1.DateA) OVERLAPS
        (V2.DateD,V2.DateA)
AND     V1.NumVol < V2.NumVol;
```

◆ **Date et temps courant :**

```
CURRENT_TIMESTAMP
CURRENT_DATE
CURRENT_TIME
```

# Requête

```
SELECT [DISTINCT] <liste d'expressions>
FROM <liste d'expressions de tables>
[WHERE <condition>]
[GROUP BY <liste d'expressions>]
[HAVING <condition>]
[ORDER BY <liste d'expressions>]
```

La clause **SELECT** produit les colonnes de la table résultat. Chaque expression correspond à une colonne.

La clause **FROM** produit une table dérivée à partir des expressions de tables qu'elle contient.

La clause **WHERE** contient une condition qui permet de sélectionner certaines lignes dans la table produite par la clause **FROM**. Seules les lignes qui rendent vrai la condition sont sélectionnées.

La clause **GROUP BY** contient une liste d'expressions qui une fois évalué sur une ligne donne une liste de valeurs. Les lignes qui donnent la même liste de valeurs sont regroupées. Il y a autant de groupes formés qu'il y a de listes de valeurs différentes. Chaque groupe de lignes doit produire une seule ligne dans le résultat de la requête. Pour cela, on utilise en clause **SELECT** des expressions figurant dans la clause **GROUP BY** et/ou des fonctions de groupe.

La clause **HAVING** contient une condition qui permet de sélectionner certains groupes de lignes formés par la clause **GROUP BY**.

La clause **ORDER BY** contient une ou plusieurs d'expressions qui spécifient les critères de tri. Les lignes du résultat de la requête sont triées suivant ces critères.



# Les expressions

Dans la clause **SELECT**, le caractère « ***** » désigne toutes les colonnes de la table produite par la clause **FROM**. Il est possible de désigner toutes les colonnes d'une table avec l'expression : `nomTable.*`.

Dans la clause **SELECT**, il est possible de donner un nom de colonne à une expression : **exp [AS] nom**

Quand une même colonne apparaît dans plusieurs tables, pour l'utiliser sans ambiguïté, il faut la préfixer par le nom de table.

Dans la clause **FROM**, une expression de table (nom de table, expression de jointure, requête) produit une table qu'il est possible de nommer : **<exp table> [AS] nomTable**

Les fonctions de groupe **COUNT**, **SUM**, **AVG**, **MIN** et **MAX** s'applique à une colonne pour produire l'agrégat sur cette colonne. Les valeurs **NULL** sont ignorées pour former l'agrégat. Si la colonne est vide ou ne contient que des **NULL**, le résultat est **NULL**. La clause **DISTINCT** permet de former l'agrégat à partir de valeurs distinctes, les doublons sont éliminés. Le cas particulier de **COUNT(*)** où le caractère « ***** » désigne toutes les lignes et non pas toutes les colonnes.

# Prédicats

Les prédicats sont évalués dans la logique à 3 valeurs, vrai, faux, indéfini.

Les prédicats sont utilisés en clause **WHERE** pour sélectionner des lignes ou en clause **HAVING** pour sélectionner des groupes.

**{exp | (requête)} op {exp | (requête)}**
Où **op** est : **= , <>, >, >=, <, <=**

**exp IS [NOT] NULL**

**exp [NOT] LIKE motif [ESCAPE carac]**
Les caractères jokers sont « % » et « _».

**exp [NOT] BETWEEN $exp_1$ AND $exp_2$**

**exp [NOT] IN {($exp_1$, …, $exp_n$) | (requête)}**

**exp {ANY | ALL} op {($exp_1$, …, $exp_n$) | (requête)}**
Où **op** est : **= , <>, >, >=, <, <=**

**EXISTS (requête)**

**période OVERLAPS période**



# Logique à 3 valeurs

◆ **Tables de vérité des opérateurs logiques :**

| AND | VRAI | FAUX | INDÉFINI |
|---|---|---|---|
| VRAI | VRAI | FAUX | INDÉFINI |
| FAUX | FAUX | FAUX | FAUX |
| INDÉFINI | INDÉFINI | FAUX | INDÉFINI |

| OR | VRAI | FAUX | INDÉFINI |
|---|---|---|---|
| VRAI | VRAI | VRAI | VRAI |
| FAUX | VRAI | FAUX | INDÉFINI |
| INDÉFINI | VRAI | INDÉFINI | INDÉFINI |

| NOT | |
|---|---|
| VRAI | FAUX |
| FAUX | VRAI |
| INDÉFINI | INDÉFINI |

Attention :

p OR NOT(p) est indéfini lorsque p est NULL

NULL = NULL est indéfini

# L'expression CASE

```
CASE expression
    WHEN expression THEN expression
    [WHEN expression THEN expression] …
    [ELSE expression]
END
```

Ou

```
CASE
    WHEN condition THEN expression
    [WHEN condition THEN expression] …
    [ELSE expression]
END
```

Lorsque la clause **ELSE** est absente, la valeur par défaut retournée est **NULL**.

**Formes abrégées :**

**NULLIF(**exp1,exp2**)**
Equivalent à :

```
CASE WHEN exp1=exp2 THEN NULL
ELSE exp1 END
```

**COALESCE(**exp1,exp2**)**
Equivalent à :

```
CASE WHEN exp1 IS NOT NULL THEN exp1
ELSE exp2 END
```



# Jointures

◆ **Opérateurs implicites (SQL norme 1) :**

Produit cartésien :

```
SELECT …
FROM Table1, Table2
[WHERE …]
```

Dans la clause **WHERE** il n'y a pas de condition de jointure

Jointure interne :

```
SELECT …
FROM Table1, Table2
WHERE <condition de jointure>
```

La jointure est décrite comme un produit cartésien suivi d'une condition de sélection liant les 2 tables.

◆ **Opérateurs explicites (SQL norme 2) :**

Produit cartésien :

```
SELECT …
FROM Table1 CROSS JOIN Table2
```

Jointure interne :

```
SELECT …
FROM Table1 [INNER] JOIN Table2
     ON <condition de jointure>
```

# Jointures

Jointure externe :

```
SELECT …
FROM  T1 {LEFT|RIGHT|FULL} [OUTER] JOIN T2
      ON <condition de jointure>
```

Formes particulières des équijointures (internes ou externes) lorsque les 2 tables ont des colonnes en commun :

Equijointure sur toutes les colonnes communes

```
SELECT …
FROM T1 NATURAL <opérateur jointure> T2
```

Equijointure sur quelques colonnes communes

```
SELECT …
FROM T1 <opérateur jointure> T2
     USING (colonne,…, colonne)
```

Dans les 2 cas, une seule occurrence des colonnes communes jointes apparait dans le résultat.



# Union, Intersection et Différence

```
<requête>
{UNION|INTERSECT|EXCEPT} [ALL]
<requête>
```

Par défaut, les résultats sont ensemblistes. Les doublons sont supprimés.

Si **ALL** est spécifié, les multiples occurrences d'une même ligne sont prises en compte. S'il y a N occurrences dans la première requête et M occurrences dans la deuxième requête, dans le résultat on aura :

N+M occurrences pour l'union
min(N,M) occurrences pour l'intersection
max(N-M,0) occurrences pour la différence

# Clause WITH

```
WITH nomReq AS (requête)
       [, nomReq AS (requête)] ...
requête
```

Pour connaître les pilotes qui assurent le plus de vols.

```
WITH NbVolsParPilote
AS (SELECT   NumPil, count(*) AS NbreVols
   FROM      Vols V
   WHERE NumPil is not null
   GROUP BY NumPil)
SELECT   NumPil  FROM NbVolsParPilote
WHERE    NbreVols =(SELECT max(NbreVols)
                    FROM   NbVolsParPilote);
```

Pour connaître les pilotes qui utilisent le plus souvent des Airbus.

```
WITH
NbVolsAirbus
AS (SELECT   NumPil, count(NumVol) AS NbreVols
   FROM      Vols V, Avions A
   WHERE V.NumAv = A.NumAv AND NumPil is not null
   AND NomAv LIKE 'Airbus%'
   GROUP BY NumPil),
VolsNonAirbus
AS (SELECT   NumVol, NumPil
   FROM      Vols V, Avions A
   WHERE V.NumAv = A.NumAv AND NumPil is not null
   AND NomAv NOT LIKE 'Airbus%'),
NbVolsNonAirbus
AS (SELECT P.NumPil, count(NumVol) AS NbreVols
   FROM Pilotes P LEFT JOIN VolsNonAirbus V
        ON P.NumPil = V.NumPil
   GROUP BY P.NumPil)
SELECT A.NumPil,  A.NbreVols as NbVolsAirbus,
N.NbreVols AS NbVolsNonAirbus
FROM NbVolsAirbus A, NbVolsNonAirbus N
WHERE A.NumPil = N.NumPil
AND A.NbreVols > N.NbreVols;
```



# Requête récursive

```
WITH RECURSIVE
    nomRequêteRécursive AS (
    <sous requête non récursive>
    UNION [ALL]
    <sous requête récursive>
    )
<Requête qui exploite la requête récursive>
```

Exemple :

```
WITH RECURSIVE
TousLesComposants
AS (SELECT Piece, Composant
    FROM Compositions
    UNION ALL
    SELECT T.Piece, C.Composant
    FROM TousLesComposants T, Compositions C
    WHERE T.Composant = C.Piece)
SELECT * FROM TousLesComposants;
```

**Compositions**

| Pièce | Composant |
|---|---|
| P3 | P1 |
| P3 | P2 |
| P5 | P3 |
| P5 | P4 |
| P6 | P5 |

**TouslesComposants**

| Pièce | Composant |
|---|---|
| P3 | P1 |
| P3 | P2 |
| P5 | P3 |
| P5 | P4 |
| P6 | P5 |
| P5 | P1 |
| P5 | P2 |
| P6 | P3 |
| P6 | P4 |
| P6 | P1 |
| P6 | P2 |

# Requête récursive

1) Evaluation de <sous requête non récursive> :
   La table résultat est égale à Composition.

2) Evaluation de <sous requête récursive> en prenant comme valeur pour la référence récursive la table précédente :

| Pièce | Composant |
|---|---|
| P5 | P1 |
| P5 | P2 |
| P6 | P3 |
| P6 | P4 |

3) Evaluation de <sous requête récursive> en prenant comme valeur pour la référence récursive la table précédente :

| Pièce | Composant |
|---|---|
| P6 | P1 |
| P6 | P2 |

4) Evaluation de <sous requête récursive> en prenant comme valeur pour la référence récursive la table précédente : La table résultat est vide. Arrêt de la récursion.

5) Le résultat est l'union des tables obtenues précédemment.



# Création d'une table

```
CREATE TABLE [schéma.]table
({def_colonne |
 [CONSTRAINT nom] def_contrainte_table},
  ..................................... )
[AS requête]
```

• Clause de définition d'une colonne

```
colonne type [DEFAULT exp]
             [ [CONSTRAINT nom] def_contrainte_colonne]…
```

• Clause de définition d'une contrainte de colonne

```
NULL | NOT NULL
```
ou
```
UNIQUE | PRIMARY KEY
```
ou
```
REFERENCES [schéma.]table [(colonne)]
[ON DELETE [CASCADE | SET NULL | SET DEFAULT]]
[ON UPDATE [CASCADE | SET NULL | SET DEFAULT]]
```
ou
```
CHECK (condition)
```

• Clause de définition d'une contrainte de table

```
{UNIQUE | PRIMARY KEY} (colonne, ..., colonne)
```
ou
```
FOREIGN KEY (colonne, ..., colonne)
REFERENCES [schéma.]table [(colonne, ..., colonne)]
[MATCH SIMPLE | MATCH FULL | MATCH PARTIAL]
[ON DELETE [CASCADE | SET NULL | SET DEFAULT]]
[ON UPDATE [CASCADE | SET NULL | SET DEFAULT]]
```
ou
```
CHECK (condition)
```

# Suppression d'une table

```
DROP TABLE [schéma.]table
```

Quand une table est supprimée, le SGBD :
- efface tous les index qui y sont attachés quel que soit le propriétaire
- efface tous les privilèges qui y sont attachés

Attention : les vues et les synonymes se référant à cette table ne sont pas supprimés.

# Modification d'une table

• Ajouter une ou plusieurs colonnes ou contraintes de table

```
ALTER TABLE [schéma].table
ADD ( {def_colonne | def_contrainte_table}
    [, {def_colonne | def_contrainte_table} ] ... )
```

• Modifier la définition d'une ou plusieurs colonnes

```
ALTER TABLE [schéma].table
MODIFY ( def_colonne [, def_colonne ] ... )
```

• Supprimer une ou plusieurs contraintes de table

```
ALTER TABLE [schéma].table
DROP CONSTRAINT contrainte
[DROP CONSTRAINT contrainte] ...
```

• Renommer une table, une colonne ou une contrainte

```
ALTER TABLE [schéma].table
RENAME [COLUMN|CONSTRAINT] ancienNom TO nouveauNom
```



# Insertion de lignes

```
INSERT INTO table [ (col,...,col) ]
VALUES (val,..,val)
```

Ou

```
INSERT INTO table [ (col,...,col) ]
requête
```

# Suppression de lignes

```
DELETE [FROM] table [alias]
[WHERE condition]
```

# Modification de lignes

```
UPDATE table [alias]
SET col = exp, ..., col = exp
[WHERE condition]
```

ou

```
UPDATE table [alias]
SET (col,...,col) = (requête)
    [ (col,...,col) = (requête) ]...
[WHERE condition]
```

# Modification ou Insertion

```
MERGE   INTO table [alias]
USING   {table|(requête)} [alias]
ON   (condition)
WHEN MATCHED THEN
   UPDATE SET col = exp, …, col = exp
   [WHERE condition]
   [DELETE WHERE condition ]
WHEN NOT MATCHED THEN
   INSERT [(col,…,col)] VALUES (exp,…, exp)
   [WHERE condition];
```

Il est impossible de modifier plusieurs fois la même ligne.

Les colonnes de la clause SET ne peuvent pas figurer dans la condition de la clause ON.

Exemples :

Mise à jour du stock avec les nouveaux achats :

```
merge into Stock S
using Achat A on (S.produit = A.produit)
when matched then
update set S.quantité = S.quantité + A.quantité
when not matched then
insert values (A.produit, A.quantité);
```

Mise à jour du stock avec les nouvelles ventes :

```
merge into Stock S
using (select produit, sum(quantité) as quantité
       from Vente group by produit) V
on (S.produit = V.produit)
when matched then
update set S.quantité = S.quantité - V.quantité
delete where S.quantité = 0;
```



# Les vues

◆ **Rôles d'une vue :**

(1) Réduire la complexité syntaxique des requêtes ou tout simplement en permettre l'expression.
(2) Définir les schémas externes.
(3) Définir des contraintes d'intégrité.
(4) Définir un niveau additionnel de sécurité en restreignant l'accès à un sous ensemble de lignes et/ou de colonnes.

◆ **Structure physique :**

Une vue n'utilise aucun espace de stockage autre que celui nécessaire à sa définition dans le dictionnaire.

◆ **Opérations sur les vues :**

**INSERT** : insertion de lignes dans la table sous-jacente à la vue.
**UPDATE** : mise à jour des lignes de la table sous-jacente à la vue.
**DELETE** : suppression de lignes dans la table sous-jacente à la vue.

Attention : Ces instructions ne s'appliquent pas aux vues qui contiennent :

- une jointure
- une clause **GROUP BY**
- la clause **DISTINCT**, une expression de calcul ou une pseudo-colonne dans la clause **SELECT**.

# Les vues

◆ **Création d'une vue de schéma externe :**

```
CREATE VIEW vue [(alias,...,alias)]
AS requête
```

Exemples :

• Création de la vue qui servira à renseigner le public sur les vols existants.

```
CREATE VIEW Public_Vols
AS SELECT NumVol, VilleD, DateD, VilleA, DateA
FROM Vols
WHERE (VilleD, DateD, VilleA, DateA) =
       ((VilleD, DateD, VilleA, DateA));
```

Utilisations possibles (Interrogation seulement) :

```
SELECT * FROM Public_Vols ;
```

Ou

```
SELECT NumVol, DateD, DateA
FROM Public_Vols
WHERE VilleD='Marseille' AND VilleA='Paris'
AND DateD  BETWEEN current_timestamp
    AND current_timestamp + interval '1' day;
```



# Les vues

• Création de la vue qui servira à la personne qui définit les vols.

```
CREATE VIEW Definition_Vols
AS SELECT NumVol, VilleD, DateD, VilleA, DateA
FROM Vols
WHERE NumAv IS NULL AND NumPil IS NULL ;
```

Utilisations possibles (Toutes opérations possibles) :

Définir un nouveau vol :

```
INSERT INTO Definition_Vols VALUES
('V1000', 'Marseille',
timestamp '2015-11-01 10:30:00',
'Paris',
timestamp '2015-11-01 11:30:00');
```

Supprimer un vol non affecté :

```
DELETE FROM Definition_Vols
WHERE NumVol = 'V1000';
```

Modifier un vol non affecté :

```
UPDATE Definition_Vols
SET DateD=DateD + interval '1' hour,
    DateA=DateA + interval '1' hour
WHERE NumVol='V1000';
```

Connaître les vols non affectés :

```
SELECT * FROM Definition_Vols ;
```

# Les vues

• Création de la vue qui servira à la personne qui affecte un avion et un pilote à un vol.

```
CREATE VIEW Affectation_Vols
AS SELECT NumVol, NumAv, NumPil
FROM Vols ;
```

Utilisations possibles (Modification seulement) :

Affecter un avion et un pilote à un nouveau vol :

```
UPDATE Affectation_Vols
SET NumAv=101 , NumPil=5050
WHERE NumVol='V1000'
AND NumAv IS NULL AND NumPil IS NULL;
```

Affecter un nouvel avion à un vol :

```
UPDATE Affectation_Vols
SET NumAv=202
WHERE NumVol='V1000';
```

Permuter l'affectation des pilotes de 2 vols :

```
UPDATE Affectation_Vols A1
SET NumPil =
    (SELECT NumPil
     FROM Affectation_Vols A2
     WHERE
     (A1.NumVol='V100' AND A2.NumVol='V200')
     OR
     (A1.NumVol='V200' AND A2.NumVol='V100'))
WHERE NumVol='V100' OR NumVol='V200' ;
```



# Les vues

◆ **Création d'une vue de vérification :**

```
CREATE VIEW vue [(alias,...,alias)]
AS requête
WITH CHECK OPTION
```

Ce genre de vues s'utilise pour contrôler l'insertion ou la modification des lignes d'une table.

Les lignes qui seraient sélectionnées par la requête si elles étaient présentes dans la table sont les seules lignes qu'il est possible ajouter par insertion ou par modification à travers la vue.

Exemples : contraintes de domaine
Utilisation d'une vue pour exprimer les domaines des attributs de la relation "avions".

```
CREATE VIEW A_Avions
AS SELECT * FROM Avions
WHERE NumAv > 0
AND CapAv > 1
AND NomAv IN ('Airbus','Boeing')
WITH CHECK OPTION ;
```

Attention : la vue ci-dessus interdit les valeurs nulles.
Pour autoriser les valeurs nulles :

```
CREATE VIEW A_Avions
AS SELECT * FROM Avions
WHERE NumAv > 0
AND (CapAv IS NULL OR CapAv > 1)
AND (NomAv IS NULL OR
     NomAv IN ('Airbus','Boeing'))
WITH CHECK OPTION ;
```

# Les vues

Exemple :
Pour interdire de réserver sur un vol qui a déjà eu lieu

```
CREATE VIEW A_Reservations
AS SELECT * FROM Reservations
WHERE numvol IN
        (SELECT numvol FROM vols
         WHERE dated > current_timestamp)
WITH CHECK OPTION ;
```

Exemple :
Pour interdire l'affectation d'un même avion à deux vols différents dont les périodes de vols se chevauchent.

```
CREATE VIEW A_Vols
AS SELECT * FROM Vols V1
WHERE NOT EXISTS (
    SELECT * FROM Vols V2
    WHERE V1.NumVol <> V2.NumVol
    AND V1.NumAv = V2.NumAv
    AND (V1.DateD,V1.DateA) overlaps
        (V2.DateD,V2.DateA)
    )
WITH CHECK OPTION ;
```

Attention : Cette vue ne doit pas être utilisée avec une instruction d'insertion ou de modification qui concerne plus d'une ligne.



# La sécurité

Le modèle de sécurité de SQL est basé sur l'octroi de privilèges aux utilisateurs.

Un privilège donne le droit d'exécuter une ou plusieurs instructions SQL.

Il y a deux sortes de privilèges :

- Les privilèges système : Ils donnent le droit d'exécuter des actions sur la base de données ou sur tout objet d'un certain type. Par exemple, le privilège **`CREATE SESSION`** donne le droit de se connecter à la base de données. Le privilège **`CREATE TABLE`** donne le droit de créer, supprimer ou modifier des tables.

- Les privilèges sur un objet : Ils donnent le droit d'exécuter une action spécifique sur un objet. Par exemple, le privilège **`SELECT`** sur une table permet de l'interroger.

Pour faciliter la gestion des privilèges, la notion de rôle est introduite. Il est possible de donner des privilèges à un rôle et ensuite assigner ce rôle à un utilisateur. L'utilisateur obtient ainsi tous les privilèges du rôle.

# La sécurité

◆ **Pour créer un utilisateur :**

```
CREATE USER nom IDENTIFIED BY motdepasse ;
```

Une fois créé, l'utilisateur existe mais il ne possède aucun privilège.

◆ **Pour créer un rôle :**

```
CREATE ROLE nom [IDENTIFIED BY motdepasse] ;
```

◆ **Attribuer des privilèges système ou des rôles :**

```
GRANT {<liste de priviléges ou de rôles> |
      ALL [PRIVILEGES]}
TO {<liste d'utilisateurs ou de rôles> |
    PUBLIC}
[WITH ADMIN OPTION]
```

La clause **WITH ADMIN OPTION** autorise les utilisateurs bénéficiaires à transmettre ou à retirer les privilèges et les rôles aux autres utilisateurs.

◆ **Retirer des privilèges système ou des rôles :**

```
REVOKE {<liste de priviléges ou de rôles> |
        ALL [PRIVILEGES]}
FROM {<liste d'utilisateurs ou de rôles> |
      PUBLIC}
```



# La sécurité

◆ **Attribuer des privilèges objet :**

```
GRANT {<liste privilèges objet> |
<liste privilèges colonne> (col,…,col)|
       ALL [PRIVILEGES]}
ON objet
TO {<liste d'utilisateurs ou de rôles> |
    PUBLIC}
[WITH GRANT OPTION]
```

Les privilèges sur une table ou une vue :
**SELECT, DELETE, INSERT, UPDATE, INDEX, REFERENCES, ALTER.**

Les privilèges sur les colonnes d'une table ou d'une vue :
**UPDATE, REFERENCES**

La clause **WITH GRANT OPTION** autorise les utilisateurs bénéficiaires à transmettre les privilèges aux autres utilisateurs.

◆ **Retirer des privilèges objet :**

```
REVOKE {<liste privilèges objet> |
<liste privilèges colonne> (col,…,col)|
       ALL [PRIVILEGES]}
ON objet
FROM {<liste d'utilisateurs ou de rôles> |
     PUBLIC}
```

# La sécurité

Exemple : Pour rendre effective la définition des schémas externes supportés par les 3 vues Public_Vols, Definition_Vols et Affectation_Vols présentées précédemment, il faut définir des autorisations.

Pour cela, on peut définir 3 rôles :

```
CREATE ROLE Role_Public_Vols ;
CREATE ROLE Role_Def_Vols ;
CREATE ROLE Role_Aff_Vols ;
```

Puis on doit attribuer les privilèges adéquats à ces rôles.

Pour autoriser l'interrogation de la vue Public_Vols :

```
GRANT SELECT
ON Public_Vols TO Role_Public_Vols ;
```

Pour autoriser l'interrogation, l'insertion, la suppression et la modification à travers la vue Defintion_Vols :

```
GRANT SELECT, INSERT, DELETE, UPDATE
ON Definition_Vols TO Role_Def_Vols ;
```

Pour autoriser la modification à travers la vue Affectation_Vols :

```
GRANT UPDATE
ON Affectation_Vols TO Role_Aff_Vols ;
```

Enfin, il convient d'affecter ces rôles aux utilisateurs.

Important : aucun privilège ne doit être accordé directement sur la table Vols.



# PL/SQL

- PL/SQL est un langage procédural

- PL/SQL comprend une partie de SQL :
  - Les types de données
  - Les expressions
  - Les requêtes
  - Les instructions de mise à jour des données (`INSERT, DELETE, UPDATE`)
  - Les instructions de gestion des transactions (`COMMIT, ROLLBACK, SAVEPOINT, LOCK TABLE, SET TRANSACTION`)

- PL/SQL se compose d'une partie procédurale de base (que l'on retrouve dans tout langage de programmation) :
  - Variables
  - Affectation
  - Instructions conditionnelles
  - Instructions itératives
  - Instructions de branchement
  - Blocs, Procédures, Fonctions
  - Packages
  - Exceptions
  - Instructions de gestion des exceptions

- PL/SQL se compose d'une partie procédurale spécifique aux bases de données :
  - Attributs de variable
  - Curseurs
  - Instructions de gestion des curseurs

# Structure d'un bloc PL/SQL

```
[ << nom_du_bloc >> ]

[ DECLARE
{       déclaration_variable ;    |
        déclaration_curseur ;     |
        déclaration_exception ;   }
        ........................  ]
BEGIN
        instruction ;
        ............
[ EXCEPTION
        traitement_exception ;
        ........................  ]
END [ nom_du_bloc ] ;
```

Exemple :

```
DECLARE
   numero_avion NUMBER(3) ;
   nom_avion    CHAR(10) ;
   cap_avion    NUMBER(3) ;
   lieu_avion   CHAR(10) ;
BEGIN
.................................
IF lieu_avion = 'Paris' THEN
   INSERT INTO Avions_Paris VALUES
   (numero_avion,nom_avion,cap_avion) ;
END IF ;
INSERT INTO Avions VALUES
(numero_avion,nom_avion,cap_avion,lieu_avion) ;
COMMIT ;
END ;
```



# Les types simples

- Les types SQL :
  `NUMBER, DATE`
  `CHAR` (max 32Ko), `VARCHAR2` (max 32Ko),
  `LONG, RAW, LONG RAW, ROWID`

- Le type `BOOLEAN` :
  `TRUE, FALSE, NULL`

- Le type `BINARY_INTEGER` :
  $-2^{31} - 1 \leq I \leq +2^{31} - 1$ $(2^{31} - 1 = 214748647)$

  Sous type `NATURAL` :
  $0 \leq N \leq +2^{31} - 1$

  Sous type `POSITIVE` :
  $1 \leq P \leq +2^{31} - 1$

  Intérêt : Pas de conversion lors des calculs.

- Le type `variable%TYPE` qui désigne le type de la variable `variable`.

- Le type `table.colonne%TYPE` qui désigne le type de la colonne.

# Conversions des types simples

Conversions explicites

| De \ Vers | CHAR | DATE | NUMBER | RAW | ROWID |
|---|---|---|---|---|---|
| CHAR | | to_date | to_number | hextoraw | chartorowid |
| DATE | to_char | | | | |
| NUMBER | to_char | to_date | | | |
| RAW | rawtohex | | | | |
| ROWID | rowidtochar | | | | |

Conversions implicites

| De \ Vers | BINARY_ INTEGER | CHAR | DATE | LONG | NUMBER | RAW | ROWID | VARCHAR2 |
|---|---|---|---|---|---|---|---|---|
| BINARY_ INTEGER | | X | | X | X | | | X |
| CHAR | X | | X | X | X | X | X | X |
| DATE | | X | | X | | | | X |
| LONG | | X | | | | X | | X |
| NUMBER | X | X | | X | | | | X |
| RAW | | X | | X | | | | X |
| ROWID | | X | | | | | | X |
| VARCHAR2 | X | X | X | X | X | X | X | |

- Passage entre PL/SQL et SQL

  Il faut une conversion explicite dans les cas suivants :

  En interrogation :

  - De DATE vers CHAR ou VARCHAR2
  - De RAW ou LONG RAW vers CHAR ou VARCHAR2

  En insertion :

  - De CHAR ou VARCHAR2 vers DATE
  - De CHAR ou VARCHAR2 vers RAW ou LONG RAW



# Variables

### ♦ Déclaration

```
nom [CONSTANT] type [NOT NULL]
    [:= exp | DEFAULT exp ];
```

Le mot clef CONSTANT impose la présence de :=exp et signifie que la valeur stockée dans la variable ne peut être modifiée.

Le mot clef NOT NULL impose la présence de :=exp ou de DEFAULT exp et signifie que la variable ne peut pas prendre de valeurs nulles. L'affectation de la variable à une valeur nulle produit une erreur.

### ♦ Portée

Toute variable est accessible à partir des blocs contenus syntaxiquement dans le bloc qui la déclare.

### ♦ Visibilité

Tout bloc qui déclare une variable masque les variables de même nom déclarées dans les blocs le contenant. Néanmoins, il peut y faire référence avec l'expression :

```
nom_du_bloc.nom_de_la_variable
```

# Les types structurés (1/2)

- Le type ligne d'une table ou d'un curseur

```
[table|curseur]%ROWTYPE
```

Faire référence à une colonne :

```
DECLARE
  ligne table%ROWTYPE ;
  ......................
BEGIN
  ......................
  ligne.col1 := ligne.col2 ;
  ......................
END;
```

- Le type enregistrement utilisateur :

```
TYPE nomtype IS RECORD
(champ touttype [NOT NULL][:=exp],
 ..................................... )
```

Faire référence à un champ :

```
DECLARE
  TYPE Tenreg IS RECORD (
    nom CHAR(10) NOT NULL := '',
    datenais date );
  enreg Tenreg ;
  ......................
BEGIN
  ......................
  enreg.nom := 'Dupond' ;
  ......................
END;
```

Attention, une affectation entre un variable de type ligne et une variable de type enregistrement est illégale.



# Les types structurés (2/2)

- Le type table PL/SQL :

```
TYPE nomtype IS TABLE OF typesimple
INDEX BY BINARY_INTEGER
```

Faire référence à un élément :

```
DECLARE
  TYPE Ttableau IS TABLE OF type
  INDEX BY BINARY_INTEGER;
  tableau Ttableau;
  i BINARY_INTEGER;
BEGIN
  ..................................
  tableau(-i) := tableau(i + 1);
  ..................................
END;
```

Remarques :

- La taille d'une table PL/SQL n'a pas d'autre limite que celle imposée par l'espace de stockage.
- Une référence à une ligne qui n'existe pas produit l'exception `NO_DATA_FOUND`.
- Il est impossible de supprimer une ligne. Une assignation à `NULL` ne supprime pas la ligne.

Vider toute une table :

```
DECLARE
  table Ttable;
  table_vide Ttable;
BEGIN
  ..................................
  table(i) := exp ;
  table := table_vide ;
  ..................................
END;
```

# Les expressions PL/SQL

Important: Les noms de colonnes sont interdits dans les expressions PL/SQL. Dans ce qui suit, le terme "expression SQL" désigne une expression SQL dans laquelle on a remplacé toute occurrence d'un nom de colonne par un nom de variable PL/SQL.

- Les variables et les paramètres de curseur.
- Les fonctions spécifiques à PL/SQL :
  - `SQLCODE` retourne le code de l'erreur courante.
  - `SQLERRM[(code_err)]` retourne le message d'erreur.
- Les expressions scalaires SQL
  Remarques :
  - La chaîne vide `''` est équivalent à `NULL`.
  - L'opérateur `||` ignore les valeurs nulles.
- Les expressions booléennes SQL (i.e. les conditions de la clause `WHERE`) construites uniquement avec :
  - `=, !=, <>, <, >, <=, >=`
  - `NOT, AND, OR, LIKE, IN, BETWEEN`
  - `IS NULL, IS NOT NULL`

  Attention : La valeur d'une expression booléenne PL/SQL est soit VRAI soit FAUX soit INDÉFINI.

```
IF a = b THEN                  IF a != b THEN
     instructions1                 instructions2
ELSE                     ≠     ELSE
     instructions2                 instructions1
END IF                         END IF
```



# Les instructions procédurales

- L'affectation :

```
variable := exp
```

- Les instructions conditionnelles :

```
IF exp_bool THEN instructions
[ ELSIF exp_bool THEN instructions ] ...
[ ELSE instructions ]
END IF ;
```

- Les instructions itératives :

```
[ <<étiquette>> ]
LOOP
instructions
END LOOP [ étiquette ] ;
```

ou

```
[ <<étiquette>> ]
WHILE exp_bool LOOP
instructions
END LOOP [ étiquette ] ;
```

ou

```
[ <<étiquette>> ]
FOR index IN [REVERSE] exp .. exp LOOP
instructions
END LOOP [ étiquette ] ;
```

- Les instructions de branchement :

```
EXIT [ étiquette ] [ WHEN exp_bool ] ;
```

ou

```
GOTO étiquette
```

- L'instruction "aucune action" :

```
NULL
```

# Les instructions SQL incluses (1/2)

• Les expressions SQL étendues

Une expression SQL étendue est une expression SQL dans laquelle une ou plusieurs constantes ont été remplacées par une expression PL/SQL

• Les requêtes SQL

```
SELECT  exp_étendue,...,exp_étendue
INTO    [var,...,var | enregistrement]
FROM    ......................
[WHERE exp_étendue ] ............
```

Important : La requête doit retourner une seule ligne.

• L'insertion de lignes dans une table

```
INSERT INTO table [ (col,...,col) ]
{VALUES (exp_plsql,..,exp_plsql) | requête}
```

• La suppression de lignes dans une table

```
DELETE [FROM] table [alias]
[WHERE exp_étendue ]
```

• La modification de lignes dans une table

```
UPDATE table [alias]
SET col = exp_étendue, ...
[WHERE exp_étendue ]
```

ou

```
UPDATE table [alias]
SET (col,...,col) = (requête), ...
[WHERE exp_étendue ]
```



# Les instructions SQL incluses (2/2)

• Les indicateurs positionnés par les ordres SELECT, INSERT, DELETE et UPDATE :

Si l'ordre porte sur une ou plusieurs lignes alors
- SQL%FOUND est vrai
- SQL%NOTFOUND est faux
- SQL%ROWCOUNT est égal au nombre de lignes sur lesquelles porte l'ordre

Si l'ordre porte sur zéro ligne alors
- SQL%FOUND est faux
- SQL%NOTFOUND est vrai
- SQL%ROWCOUNT est égal à 0

Attention :
- Lorsqu'un ordre SELECT ne retourne aucune ligne, l'exception NO_DATA_FOUND est déclenchée.
- Lorsqu'un ordre SELECT retourne plusieurs lignes, l'exception TOO_MANY_ROWS est déclenchée.

• Les instructions de gestion des transactions :

```
COMMIT [WORK]
ROLLBACK [WORK] [TO [SAVEPOINT] point_de_reprise]
SAVEPOINT point_de_reprise
SET TRANSACTION READ ONLY
LOCK TABLE table IN verrou MODE [NOWAIT]
```

Fonctionnement identique à celui de SQL.

# Les curseurs (1/3)

C'est une structure qui permet de traiter une requête SQL (ordre SELECT sans INTO) qui retourne plusieurs lignes.

- Déclaration d'un curseur :

```
CURSOR cur [(para,...,para)]
IS requête_sql
```

Où la déclaration de paramètre `para` est :

```
nom [IN] type [:=exp | DEFAULT exp]
```

- Ouverture d'un curseur :

```
OPEN cur [(exp, ..., exp)]
```

ou

```
OPEN cur [(para=>exp,...,para=>exp)]
```

Assignation des valeurs aux paramètres par position (1er cas) ou par nom (2ème cas)

- Accès à la ligne suivante :

```
FETCH curseur INTO var,...,var
```

ou

```
FETCH curseur INTO enregistrement
```

Important : Le nombre de variables doit être égal au nombre de colonnes retournées par la requête.

- Fermeture d'un curseur :

```
CLOSE curseur
```



# Les curseurs (2/3)

♦ **Les indicateurs positionnées par un curseur :**

- `nom_de_curseur%NOTFOUND` est :
  - indéfini après `OPEN` et avant le premier `FETCH`
  - vrai après un `FETCH` qui ne retourne pas de lignes
  - faux après un `FETCH` qui retourne une ligne

  Si le curseur n'est pas ouvert et si cette variable est référencée, il y a déclenchement de l'exception `INVALID_CURSOR`.
- `nom_de_curseur%FOUND` est la négation logique de `nom_de_curseur%NOTFOUND`
- `nom_de_curseur%ROWCOUNT` est égal au nombre de lignes retournées par `FETCH` depuis l'ouverture du curseur. L'exception `INVALID_CURSOR` est déclenchée comme précédemment.
- `nom_de_curseur%ISOPEN` est vrai si le curseur est ouvert, faux sinon.

♦ **La clause WHERE CURRENT OF**

```
WHERE CURRENT OF curseur
```

Elle permet aux ordres `UPDATE` et `DELETE` de sélectionner la ligne retournée par le dernier `FETCH`.

♦ **La boucle FOR sur un curseur ou une requête :**

```
 [ <<étiquette>> ]
FOR enreg IN [curseur | requête_SQL] LOOP
instructions
END LOOP [ étiquette ] ;
```

# Les curseurs (3/3)

♦**Exemples :**

```
DECLARE
CURSOR curseur IS
     SELECT ....;
enreg curseur%ROWTYPE;
.......................
BEGIN
.......................
OPEN curseur ;
......<inst. 1>.........
LOOP
     FETCH curseur INTO enreg ;
     EXIT WHEN curseur%NOTFOUND ;
     .........<inst. 2>..........
END LOOP ;
CLOSE curseur ;
.......................
END ;
```

```
DECLARE
CURSOR curseur IS
     SELECT ....;
.......................
BEGIN
.......................
FOR enreg IN curseur LOOP
     .........<inst. 2>..........
END LOOP ;
.......................
END ;
```



# Les exceptions (1/2)

Une exception correspond à l'arrêt du traitement normal d'une séquence d'instructions. Cet arrêt peut être programmé par l'utilisateur ou être le résultat d'une erreur d'exécution. Une exception sera dénotée par un nom.

- Déclaration d'une exception (utilisateur)

```
nom_exception EXCEPTION
```

- Déclenchement d'une exception

```
RAISE [nom_exception]
```

Toutes les exceptions sont autorisées, utilisateurs ou pré-définies.

Lorsqu'un nom d'exception est spécifié, cette instruction réalise un branchement, dans la partie exception du bloc PL/SQL, à l'instruction de traitement de l'exception. Si une telle instruction n'existe pas, le branchement essaie de se faire dans les blocs enchâssant. Si aucun de ces blocs ne sait traiter l'exception, le programme est arrêté.

Sans nom d'exception cette instruction, lorsqu'elle est placée dans la partie qui concerne le traitement de l'exception, assure un branchement aux blocs enchâssant pour traiter l'exception.

# Les exceptions (2/2)

• Traitement d'une exception

```
   WHEN except [ OR ... OR except ]
   THEN instructions
```
ou
```
   WHEN OTHERS
   THEN instructions
```

• Liste des exceptions pré-définies

| | |
|---|---|
| CURSOR_ALREADY_OPEN | erreur -6511 |
| DUP_VAL_ON_INDEX | erreur -1 |
| INVALID_CURSOR | erreur -1001 |
| INVALID_NUMBER | erreur -1722 |
| LOGIN_DENIED | erreur -1017 |
| NO_DATA_FOUND | erreur +100 |
| NO_LOGGED_ON | erreur -1012 |
| PROGRAM_ERROR | erreur -6501 |
| STORAGE_ERROR | erreur -6500 |
| TIMEOUT_ON_RESOURCE | erreur -51 |
| TOO_MANY_ROWS | erreur -1422 |
| TRANSACTION_BACKED_OUT | erreur -61 |
| VALUE_ERROR | erreur -6502 |
| ZERO_DIVIDE | erreur -1476 |

• Associer un nom d'exception à une erreur Oracle

```
PRAGMA EXCEPTION_INIT (nom,code_err)
```



# Procédures et fonctions (1/2)

♦ **Les procédures PL/SQL**

```
PROCEDURE proc [(para, ..., para)] IS
    [declarations_locales]
BEGIN
    instructions
[EXCEPTION
    traitements_exceptions ]
END [proc] ;
```

• Terminaison forcée : `RETURN ;`

♦ **Les fonctions PL/SQL**

```
FUNCTION fonct [(para, ..., para)]
RETURN type IS
    [declarations_locales]
BEGIN
    instructions
[EXCEPTION
    traitements_exceptions ]
END [fonct] ;
```

• Spécification du résultat : `RETURN exp ;`

♦ **Déclaration "forward" des procédures et fonctions**

```
PROCEDURE proc [(para, ..., para)] ;
```
ou
```
FUNCTION fonct [(para, ..., para)]
RETURN type ;
```

# Procédures et fonctions (2/2)

♦ **Déclaration des paramètres**

```
paramètre [IN | OUT | IN OUT] type
              [:=valeur | DEFAULT valeur]
```

`IN` : Le paramètre se comporte comme une constante. Il est impossible d'affecter une valeur au paramètre.

`OUT` : Le paramètre se comporte comme une variable non initialisée. Il est impossible d'utiliser le paramètre pour affecter une autre variable.

`IN OUT` : Le paramètre se comporte comme une variable initialisée.

Attention : Les types `CHAR`, `VARCHAR` et `NUMBER` sont possibles uniquement sans leurs arguments.

♦ **Passage des paramètres**

Par position :

```
proc (exp, ..., exp)
```

Ou par nom :

```
proc(param=>exp,...,param=>exp)
```

Remarque : Seul le passage par nom permet l'omission des paramètres `IN`.



# Les packages (1/3)

Un package est un groupe logique formé de types, d'objets, de procédures et de fonctions.

```
PACKAGE pack IS
   [déclarations_types_publics]
   [spécifications_objets_publics]
   [spécifications_procédures]
   [spécifications_fonctions]
END [pack] ;

PACKAGE BODY pack IS
   [déclarations_types_privés]
   [déclarations_objets_privés]
   [déclarations_procédures]
   [déclarations_fonctions]
[BEGIN
   instructions_d_initilisation ]
END [pack] ;
```

Remarque :

Un package composé uniquement de déclarations de type et ou de variables peut ne pas avoir de `PACKAGE BODY` associé.

• Faire référence à un composant d'un package :

```
nom_pack.nom_type
nom_pack.nom_objet
nom_pack.nom_procédure
nom_pack.nom_fonction
```

# Les packages (2/3)

- Spécification d'objets :

  Pour les variables :
  ```
  déclaration_variable
  ```

  Pour les curseurs :
  ```
  CURSOR curseur [(para,...,para)] RETURN type;
  ```

- Spécification de procédures ou de fonctions :

  ```
  PROCEDURE proc [(para, ..., para)] ;
  ```
  ou
  ```
  FUNCTION fonct [(para, ..., para)]
  RETURN type ;
  ```

- Déclaration des curseurs dans le `PACKAGE BODY`

  ```
  CURSOR curseur [(para,...,para)] RETURN type
  IS requête_sql ;
  ```

- Le package STANDARD :

  Il contient toutes les fonctions `ABS`, ...

  Remarque :
  Si une fonction est définie avec le même nom qu'une fonction du package standard, alors elle masque cette dernière. Il est possible d'appeler la fonction du package standard par :

  ```
  STANDARD.fonction
  ```



# Les packages (3/3)

- Le package DBMS_STANDARD :

  C'est une extension du package standard.

  Il contient notamment la procédure :

  ```
  raise_application_error(num_err, mess_err)
  ```

  Où :
  - `num_err` : un entier négatif compris entre -20000 et -20999
  - `mess_err` : une chaîne de caractères d'au plus 512 octets

**♦Stockage dans la base de données**

Pour stocker dans la base une procédure, une fonction ou un package il suffit de faire précéder sa déclaration par le mot réservé `CREATE`.

Intérêt :
L'exécution est plus rapide car l'analyse syntaxique n'est plus à faire.

**♦Appel d'un programme stocké à partir de SQL*Plus**

```
EXECUTE [pack.]prog [(para, ..., para)] ;
```

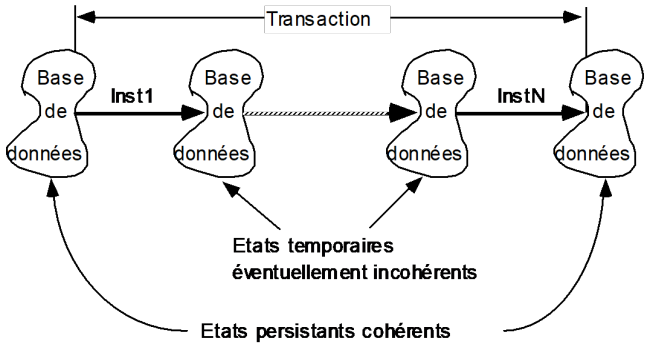
# Les transactions

Une transaction est une séquence d'instructions SQL dont les actions sur la base sont indivisibles :

- Soit toutes les actions sont exécutées
- Soit aucune action ne prend effet.

On dit que la transaction est atomique.

L'atomicité d'une transaction préserve la cohérence des données**.** En effet, une transaction permet le passage d'un état cohérent de la base à un autre état cohérent en rendant temporaire les états intermédiaires, qui ne sont pas nécessairement cohérents.




L'existence de pannes pouvant interrompre une transaction en son milieu impose des mécanismes automatiques de retour à un état cohérent.



# Les transactions

◆ **Début d'une transaction**

L'indication du début d'une transaction est implicite :

- Après la connexion à la base de données
- Après la fermeture d'une transaction.

Remarque :

Toute instruction SQL exécutée par le SGBD appartient à une transaction.

◆ **Fin d'une transaction**

Une transaction a deux façons de se terminer :

- en validant les changements
- en défaisant les changements

Une transaction se termine :

- quand une instruction **`COMMIT [WORK]`** est utilisée, les changements sont validés.
- quand une instruction **`ROLLBACK [WORK]`** est utilisée, les changements sont défaits.
- avant d'exécuter une instruction DDL, les changements sont validés.
- après avoir exécuté une instruction DDL, les changements sont validés si l'exécution ne s'est pas terminée par une erreur.
- quand l'utilisateur se déconnecte du système, les changements sont validés.
- quand le processus connexion se termine mal, les changements sont défaits.

# Les transactions

◆ **Pose d'un point de reprise**

```
SAVEPOINT point_de_reprise
```

La pose d'un point de reprise permet de défaire la transaction jusqu'à ce point.
Par défaut, le maximum de point de reprise par transaction est 5.
Si vous créez un second point de reprise avec le même nom qu'un point de reprise précédent, ce dernier est alors effacé.

◆ **Défaire partiellement les changements sans fermer la transaction**

```
ROLLBACK [WORK] TO [SAVEPOINT] point_reprise
```

Utilisé avec un point de reprise, l'instruction **ROLLBACK** réalise les actions suivantes :

- défait la transaction jusqu'au point de reprise
- le point de reprise n'est pas effacé, permettant ainsi plusieurs retours arrière jusqu'au même point
- conserve les autres points de reprise placés avant le point de reprise.
- efface tous les verrous posés après le point de reprise



# Les déclencheurs

◆ **Rôles d'un déclencheur sur table**

(1) Déclencher automatiquement des actions sur la base de données en fonction des opérations de mises à jour (**INSERT, DELETE, UPDATE**) effectuées sur les tables.
Cela permet :
- De faire un audit fin des actions utilisateurs.
- Calculer automatiquement des valeurs pour certaines colonnes.
- Maintenir des copies de tables.

(2) Exprimer des contraintes sur les opérations de mises à jour des tables.

(3) Exprimer des autorisations complexes pour renforcer la sécurité.

◆ **Rôles d'un déclencheur sur vue**

(1) Décrire les mises à jour à travers les vues.

◆ **Rôles d'un déclencheur système**

(1) Contrôler les instructions DDL.
(2) Contrôler le démarrage et l'arrêt de la base de données
(3) Contrôler les connexions à la base de données

# Les déclencheurs

◆ **Création d'un déclencheur sur table**

```
CREATE TRIGGER trigger
{ BEFORE | AFTER }
{ INSERT | DELETE | UPDATE [OF col,...,col] }
[ OR {INSERT |DELETE |UPDATE [OF col,...,col] } ...]
  ON table
[ [ REFERENCING [ OLD [AS] old ] [ NEW [AS] new ] ]
  FOR EACH ROW [WHEN (condition_pl/sql)] ]
  bloc_pl/sql ;
```

Un déclencheur concerne une seule table, spécifiée par la clause **ON**.

Les évènements déclenchants sont les instructions de mise à jour de la table (**INSERT, DELETE, UPDATE**). Toute fois dans le cas de l'**UPDATE**, il possible de restreindre le déclenchement à la modification de certaines colonnes à l'aide de la clause **OF**.

Il y a 2 types de déclencheurs :

- Les déclencheurs sur instruction : absence de clause **FOR EACH ROW**.
- Les déclencheurs sur ligne : la clause **FOR EACH ROW** est spécifiée.

Pour chaque type de déclencheur, il y a 2 moments de déclenchement : **BEFORE** et **AFTER**. Il y a donc 4 façons de déclencher le programme décrit par le bloc PL/SQL :

- avant l'instruction déclenchante
- après l'instruction déclenchante
- avant chaque ligne traitée par l'instruction déclenchante
- après chaque ligne traitée par l'instruction déclenchante



# Les déclencheurs

Remarque :

Un déclencheur peut en déclencher un autre, et ainsi de suite.

Attention :

Le bloc PL/SQL qui décrit l'action du déclencheur ne peut contenir aucun ordre de gestion des transactions (COMMIT, ROLLBACK et SAVEPOINT).

◆ **Activé ou désactivé un déclencheur**

```
ALTER TRIGGER trigger ENABLE;
```

Ou

```
ALTER TRIGGER trigger DISABLE;
```

Remarque :

À la création, un déclencheur est toujours activé.

◆ **Supprimer un déclencheur**

```
DROP TRIGGER trigger;
```

# Les déclencheurs

◆ **Caractéristiques des déclencheurs sur ligne**

Dans un déclencheur sur ligne, il est possible d'accéder aux données de chaque ligne traitée. Deux enregistrements **OLD** et **NEW**, dont les champs sont les noms des colonnes de la table donnent accès aux valeurs de la ligne avant et après traitement conformément au tableau suivant :

| Instruction déclenchante | Valeur du champ **OLD.**colonne | Valeur du champ **NEW.**colonne |
|---|---|---|
| **INSERT** | **NULL** | Valeur insérée pour la colonne |
| **DELETE** | Valeur supprimée pour la colonne | **NULL** |
| **UPDATE** | Valeur avant la modification de la colonne | Valeur après la modification de la colonne |

Il est possible de les renommer les enregistrements **OLD** et **NEW** avec la clause **REFERENCING**.

La clause **WHEN** permet de restreindre le déclenchement par une condition à satisfaire. Cette condition ne peut contenir aucune requête. Mais elle peut contenir les champs des enregistrements **OLD** et **NEW**.

A l'intérieur du bloc PL/SQL, les enregistrements **OLD** et **NEW** sont considérées comme des variables externes, ils doivent donc être préfixées par ":" .

Important : L'exception « Table en mutation » est levée par une lecture de la table qui supporte le déclencheur.



# Les déclencheurs

◆ **Prédicat de détection de l'instruction déclenchante**

Lorsqu'un déclencheur est défini sur plusieurs instructions déclenchantes, les prédicats suivants peuvent être utilisés pour déterminer l'instruction déclenchante. :

**INSERTING** vrai si **INSERT** est l'instruction déclenchante
**DELETING** vrai si **DELETE** est l'instruction déclenchante
**UPDATING** vrai si **UPDATE** est l'instruction déclenchante
**UPDATING('**colonne**')** vrai si l'**UPDATE** déclenchant concerne la colonne spécifiée.

Exemple :
Audit des actions effectuées sur la table Avions dans la table :
Audit(utilisateur, action, date_action)

```
CREATE TRIGGER Audit_Avions
BEFORE INSERT OR DELETE OR UPDATE
ON Avions
DECLARE
    action varchar(20) ;
BEGIN
    IF inserting THEN action := 'Insertion' ;
    IF deleting THEN action := 'Suppression' ;
    IF updating THEN action := 'Modification' ;
    INSERT INTO Audit
    VALUES (user, action, current_timestamp) ;
END ;
```

# Les déclencheurs

◆ **Modèle d'exécution des déclencheurs et des contraintes d'intégrités**

1) Exécution des déclencheurs "Avant instruction"

2) Pour chaque ligne traitée

   a) Exécution des déclencheurs "Avant chaque ligne"

   b) Validation des contraintes d'intégrités non différées, action sur la ligne (insertion, suppression ou modification) et verrouillage de la ligne.

   c) Exécution des déclencheurs "Après chaque ligne"

3) Validation des contraintes d'intégrités différées (i.e. contraintes de clefs et références dans certains cas)

4) Exécution des déclencheurs "Après instruction"

Important :

Lorsqu'un déclencheur lève une exception, cela produit l'annulation de l'instruction déclenchante par retour arrière au début de l'exécution.



# Les déclencheurs

◆ **Gestion d'une colonne dérivée**

Les tables :

Services (NumSer, NomSer, Total_Salaires)
Employes (NumEmp, NomEmp, Salaire, *NumSer*)

Pour chaque service, le trigger maintient automatiquement le total des salaires des employés du service.

```
CREATE OR REPLACE TRIGGER total_salaire
AFTER INSERT OR DELETE OR UPDATE OF NumSer, Salaire
ON Employes FOR EACH ROW
BEGIN
  IF INSERTING OR
     (UPDATING AND :old.NumSer != :new.NumSer)
  THEN
     UPDATE Services
     SET Total_Salaires = Total_Salaires + :new.Salaire
     WHERE NumSer = :new.NumSer;
  END IF;
  IF DELETING OR
     (UPDATING AND :old.NumSer != :new.NumSer)
  THEN
     UPDATE Services
     SET Total_Salaires = Total_Salaires - :old.Salaire
     WHERE NumSer = :old.NumSer;
  END IF;
  IF UPDATING AND :old.NumSer = :new.NumSer
     AND :old.Salaire != :new.Salaire
  THEN
     UPDATE Services
     SET Total_Salaires = Total_Salaires - :old.Salaire
                          + :new.Salaire
     WHERE NumSer = :new.NumSer;
  END IF;
END;
```

# Les déclencheurs

◆ **Contraintes de références dans une BD distribuée**

Sur le serveur 1 :

```
CREATE PUBLIC DATABASE LINK serveur2
CONNECT TO user IDENTIFIED BY mot_de_passe
USING 'serveur2';
CREATE PUBLIC SYNONYM vol FOR vol@serveur2 ;

CREATE TRIGGER supp_avion
AFTER DELETE OR UPDATE OF numav ON avion
DECLARE
    nb INTEGER;
BEGIN
    SELECT 1 INTO nb FROM dual
    WHERE NOT EXISTS
    (SELECT * FROM vol
    WHERE numav NOT IN (SELECT numav FROM avion));
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    raise_application_error(-20000,'Ref. violee');
END;
```

Sur le serveur 2 :

```
CREATE PUBLIC DATABASE LINK serveur1
CONNECT TO user IDENTIFIED BY mot_de_passe
USING 'serveur1';
CREATE PUBLIC SYNONYM avion FOR avion@serveur1 ;

CREATE TRIGGER ajout_vol
AFTER INSERT OR UPDATE OF numav ON vol
DECLARE
    nb INTEGER;
BEGIN
    SELECT 1 INTO nb FROM dual
    WHERE NOT EXISTS
    (SELECT * FROM vol
    WHERE numav NOT IN (SELECT numav FROM avion));
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    raise_application_error(-20000,'Ref. violee');
END;
```



# Les déclencheurs

◆ **Création d'un déclencheur sur vue**

```
CREATE TRIGGER trigger
INSTEAD OF
{ INSERT | DELETE | UPDATE }
[ OR {INSERT |DELETE |UPDATE } ...]
  ON vue
[ REFERENCING [ OLD [AS] old ] [ NEW [AS] new ]
  bloc_pl/sql ;
```

Un déclencheur sur vue est toujours un déclencheur de niveau ligne : Le bloc PL/SQL s'exécute pour chaque ligne traitée par l'instruction déclenchante.

Les enregistrements **OLD** et **NEW** sont disponibles uniquement en lecture.

Exemple :

```
create or replace view Durée_Vol as
select NumVol, DateA - DateD as Durée
from Vols;

create or replace trigger modif_durée_vol
instead of update on Durée_Vol
begin
    update Vols set DateA = DateD + :new.Durée
    where NumVol = :old.NumVol;
end;

update Durée_Vol set Durée = Durée + interval '1' hour
where NumVol in ('V1', 'V2');
```

# LA CONCURRENCE

# Notion de Transaction

Point de vue utilisateur : Une transaction est une séquence d'instructions SQL qui est produit par l'exécution d'un programme.

```
create procedure trVirement (idCptDebit Integer,
            idCptCredit Integer, somme Decimal)
as
   ErreurVirement Exception;
begin
   update Comptes set solde = solde - somme
   where idCpt = idCptDebit and solde - somme >= 0;
   if SQL%RowCount <> 1
   then raise ErreurVirement; end if;
   update Comptes set solde = solde + somme
   where idCpt = idCptCredit;
   if SQL%RowCount <> 1
   then raise ErreurVirement; end if;
   commit;
exception
   when others then rollback;
end;
```

L'exécution suivante :

```
begin   trVirement(1,2,100); end;
```

peut produire la transaction :

```
update Comptes set solde = solde - 100
where idCpt = 1 and solde - 100 >= 0;
update Comptes set solde = solde + 100
where idCpt = 2;
commit;
```

Point de vue SGBD : Une transaction est une séquence d'opérations élémentaires.

**Start R($solde_1$) W($solde_1$, $solde_1$-100)**
**R($solde_2$) W($solde_2$, $solde_2$+100) Commit**

Ou

**R($solde_1$)W($solde_1$)R($solde_2$)W($solde_2$)Commit**



# Notion de Transaction

◆ **Définition :**

Une transaction est une séquence d'opérations qui doit s'exécuter de façon atomique, soit toutes les opérations sont exécutées, soit aucune ne l'est.

◆ **Les opérations élémentaires formant une transaction :**

Opérations sur une donnée :

- Lecture d'une donnée : **R(x,v)** retourne la valeur v de x, ou **R(x)** retourne la valeur de x.
- Ecriture d'une donnée : **W(x,v)** écrit la valeur v dans x, ou **W(x)** écrit une nouvelle valeur dans x.

Opérations de gestion des transactions :

- Début de transaction : **Start** marque le début de la transaction. La transaction se voit attribuer un identifiant unique par le SGBD. Cet identifiant sera attaché à chaque opération de la transaction.
- Fin de transaction en validant les changements : **Commit** indique que la transaction s'est terminée normalement et que ses effets doivent devenir permanents.
- Fin de transaction en annulant les changements : **Abort** (pour Rollback) indique que la transaction s'est terminée anormalement et que ses effets doivent être effacés.

# Exécution concurrente

◆ **Notion de concurrence :**

Une exécution concurrente est la simultanéité dans le temps de plusieurs transactions.

L'unité d'exécution concurrente est donc la transaction.

◆ **Exécution concurrente correcte :**

Une exécution concurrente est correcte si pour les transactions qui la composent, il existe une exécution indépendante de ces mêmes transactions qui produit le « même résultat » que celui produit par l'exécution concurrente.

On distingue plusieurs types d'anomalies provoquées par une exécution concurrente. Les plus représentatives sont :

(1) Mise à jour perdue : Une transaction perd une mise à jour qu'elle a effectuée.
(2) Lecture impropre : Une transaction lit une valeur temporaire, qui n'a pas d'existence dans un état cohérent et persistant de la base de données.
(3) Lecture non reproductible : Pour une même donnée, une transaction a lu une valeur qu'elle ne peut plus relire bien qu'elle ne l'ait pas modifiée.
(4) Problèmes d'annulation : L'opération d'annulation doit avoir une exécution correcte et simple.



# Problème de concurrence

◆ **Mise à jour perdue :**

$T_1$ :
```
update Employes
set salaire = salaire + 100
where nom = 'Paul';
commit;
```

$T_2$ :
```
update Employes
set salaire = salaire + 200
where nom = 'Paul';
commit;
```

On suppose l'exécution concurrente suivante :

| $T_1$ | $T_2$ |
|---|---|
| R($salaire_{paul}$) | |
| | R($salaire_{paul}$) |
| W($salaire_{paul}$, $salaire_{paul}$ +100) | |
| | W($salaire_{paul}$, $salaire_{paul}$ +200) |
| Commit | |
| | Commit |

A la fin de cette exécution, le salaire de Paul est égal à
ancien salaire + 200
alors qu'il aurait dû être égal à
ancien salaire + 100 +200

La mise à jour faite par $T_1$ est perdue.

# Problème de concurrence

◆ **Lecture impropre :**

$T_1$ :
```
update Comptes set solde = solde - 100
where IdCompte = 1;
update Comptes set solde = solde + 100
where IdCompte = 2;
commit;
```

$T_2$ :
```
select sum(solde)as balance from Comptes
where IdCompte in (1,2);
commit;
```

On suppose l'exécution concurrente suivante :

| $T_1$ | $T_2$ |
|---|---|
| $R(solde_1)$ | |
| $W(solde_1, solde_1 - 100)$ | |
| | $R(solde_1)$ |
| | $R(solde_2)$ |
| | $Afficher(solde_1+solde_2)$ |
| | Commit |
| $R(solde_2)$ | |
| $W(solde_2, solde_2 + 100)$ | |
| Commit | |

A la fin de cette exécution, la nouvelle balance ($solde_1+solde_2$) des 2 comptes est égale à l'ancienne valeur. Alors que $T_2$ affiche une balance erronée.

Les lectures faites par $T_2$ sont impropres.



# Problème de concurrence

◆ **Lecture non reproductible :**

$T_1$ :
```
select 0.3 * salaire as charge
from Employes
where nom = 'Paul';
select salaire from Employes
where nom = 'Paul';
commit;
```

$T_2$ :
```
update Employes set salaire = salaire+100
where nom = 'Paul';
commit;
```

On suppose l'exécution suivante :

| $T_1$ | $T_2$ |
|---|---|
| $R(salaire_{paul})$ | |
| $Afficher( 0.3 \times salaire_{paul})$ | |
| | $R(salaire_{paul})$ |
| | $W(salaire_{paul}, salaire_{paul}+100)$ |
| | Commit |
| $R(salaire_{paul})$ | |
| $Afficher(salaire_{paul})$ | |
| Commit | |

Dans le corps de $T_1$, la même action, la lecture du salaire de Paul, produit deux résultats différents.

# Problème de concurrence

◆ **Les fantômes :**

$T_1$ :
```
select nom from Employes where salaire > 2000;
select nom from Employes where salaire > 2000;
commit;
```

$T_2$ :
```
insert into Employes (nom,salaire)
values ('Jean',3000);
commit;
```

On suppose l'exécution suivante :

| $T_1$ | $T_2$ |
|---|---|
| select nom from Employes where salaire > 2000 ; | |
| | insert into Employes (nom, salaire) values ('Jean',3000) ; commit ; |
| select nom from Employes where salaire > 2000 ; commit ; | |

Tous les noms lus par $T_1$ la première fois ne sont pas modifiés par $T_2$. Donc il n'y pas de lecture non-reproductible des noms.
La lecture non-reproductible se situe au niveau de l'ensemble des lignes satisfaisant le prédicat de la clause where, P(salaire > 2000)



# Problème de concurrence

◆ **Annulation impossible :**

$T_1$ :
```
update Employes
set salaire = salaire + 100
where nom = 'Paul';
rollback;
```

$T_2$ :
```
update Charges C
set montant = (Select salaire*0.3
               From Employes Where nom=C.nom)
where nom = 'Paul';
commit;
```

On suppose l'exécution concurrente suivante :

| $T_1$ | $T_2$ |
|---|---|
| $R(salaire_{paul})$ | |
| $W(salaire_{paul}, salaire_{paul} +100)$ | |
| | $R(salaire_{paul})$ |
| | $W(montant_{paul}, salaire_{paul} \times 0.3)$ |
| | Commit |
| Abort | |

L'annulation de $T_1$ est impossible. En effet, si $T_1$ est annulée alors :

- Soit $T_2$ conserve sa terminaison en validant, et la sémantique de la lecture faite par $T_2$ est incorrecte (une donnée inexistante a été lue).
- Soit $T_2$ annule ses changements, et la sémantique de la validation de $T_2$ (commit) est incorrecte.

# Problème de concurrence

◆ **Annulation en cascade :**

On suppose l'exécution concurrente suivante :

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(x) | | |
| W(x) | | |
| | R(x) | |
| | W(y) | |
| | | R(y) |
| | | W(z) |
| Abort | | |

Lorsque $T_1$ est annulée, le seul moyen de corriger les écritures (liées aux lectures impropres) de $T_2$ et $T_3$ est d'annuler ces transactions et de les exécuter à nouveau.
Donc l'annulation de $T_1$ entraine par cascade l'annulation de $T_2$ et $T_3$.



# Le modèle

Une ***transaction*** $T_i$ est une séquence d'opérations de lecture $R_i(x)$ et d'écriture $W_i(x)$ qui se termine soit par l'opération de validation $C_i$ soit par celle d'annulation $A_i$. Chaque opération est marquée de l'indice de la transaction à laquelle elle appartient.

Un ***historique complet*** défini sur un ensemble de transaction $T = \{T_1,\ldots, T_n\}$ est un entrelacement de toutes les opérations des $T_i$ qui respecte l'ordre spécifié par chaque transaction. Il représente une exécution concurrente des transactions $T_1,\ldots, T_n$. L'ordre des opérations défini par l'entrelacement est noté $<$. C'est un ordre total.

Un ***historique*** est un préfixe (i.e. séquence de début) d'un historique complet H. Il est défini sur une partie $T_{i1},\ldots,T_{im}$ des transactions $T_1,\ldots,T_n$ de H. Il représente un état d'avancement d'une exécution concurrente des transactions $T_{i1},\ldots,T_{im}$.

Un historique complet est un cas particulier d'historiques où toutes les transactions sont arrivées à leur terme.

On note **Op(H)** l'ensemble des opérations qui constituent l'historique H.

Dans un historique H, une transaction $T_i$ est dite ***validée*** si $C_i \in H$, ***annulée*** si $A_i \in H$ ou ***active*** si $C_i \notin H$ et $A_i \notin H$.

# Le modèle

Ce modèle simple peut être étendu :

- Répétions d'une opération : Dans une transaction $T_i$, on ne permet pas plusieurs occurrences d'une opération $R_i(x)$ ou $W_i(x)$, car l'opération est repérée par l'indice i de la transaction. Il est possible d'étendre le modèle en autorisant des indices composés i.1,…, i.n où le suffixe repère les occurrences.
- Plus de concurrence : On peut étendre le modèle en autorisant l'exécution en parallèle des opérations qui ne sont pas conflictuelles. Une transaction et un historique sont alors définis comme un ordre partiel sur l'ensemble des opérations qui les constituent.
- Plus d'opérations : Les seules opérations SQL représentées sont l'interrogation simple (Select) et la modification simple (Update). Il est possible d'étendre le modèle pour prendre en compte l'interrogation d'un ensemble de lignes à partir d'un prédicat, la création et la suppression de données (Insert et Delete).

Ces extensions ne remettent pas en cause les résultats obtenus.

Que le modèle soit simple ou étendu, une hypothèse est toujours formulée : Les opérations de lecture et d'écriture sont considérées atomiques.



# Dépendance de lecture

Dans un historique H, une transaction $T_j$ ***lit une donnée x à partir*** d'une transaction $T_i$ si :

- $T_j$ lit x après que $T_i$ l'a écrit : $W_i(x) < R_j(x)$
- $T_i$ n'a pas été annulée avant que $T_j$ lise x : $A_i \prec R_j(x)$
- Si une transaction a écrit x entre le moment où $T_i$ a écrit x et le moment où $T_j$ lit x alors elle a été annulée avant que $T_j$ lise x :

  Si $W_i(x) < W_k(x) < R_j(x)$ alors $A_k < R_j(x)$

On dit que $T_j$ ***lit à partir de*** $T_i$ si $T_j$ lit une donnée quelconque x à partir de $T_i$.

On note **RF(H)** la relation $\{(T_i,x,T_j) / T_j$ lit x à partir de $T_i\}$

Exemples :

| | |
|---|---|
| $W_1(x)\ W_2(x)\ R_3(x)$ | $T_3$ lit x à partir de $T_2$ |
| $W_1(x)\ W_2(x)\ A_2\ R_3(x)$ | $T_3$ lit x à partir de $T_1$ |
| $W_1(x)\ R_1(x)$ | $T_1$ lit x à partir de $T_1$ |

La dépendance de lecture entre deux transactions est une caractéristique importante qui est la cause de la plupart des problèmes de concurrence :

- Sérialisabilité
- Recouvrabilité
- Annulations en cascade
- …

# Recouvrabilité

Un historique H est ***recouvrable*** si pour toute transaction, sa validation est après les validations des autres transactions à partir desquelles elle a lu, i.e. pour toute transaction $T_j$

Si $T_j$ lit à partir de $T_i$ et $i \neq j$ et $C_j \in H$ alors $C_i < C_j$

Exemples :

| | |
|---|---|
| $W_1(x)\ R_2(x)\ C_1\ C_2$ | Est recouvrable |
| $W_1(x)\ R_2(x)\ A_1\ A_2$ | Est recouvrable |
| $W_1(x)\ R_2(x)\ C_2\ A_1$ | N'est pas recouvrable |
| $W_1(x)\ R_2(x)\ A_1\ C_2$ | N'est pas recouvrable |
| $W_1(x)\ R_2(x)\ C_2\ C_1$ | N'est pas recouvrable |

$W_1(x)\ R_2(x)\ W_2(y)\ R_1(y)\ C_1\ C_2$

$W_1(x)\ R_2(x)\ W_2(y)\ R_1(y)\ C_2\ C_1$

Les 2 derniers ne sont pas recouvrables

La recouvrabilité est une notion importante car elle garantit que l'opération d'annulation est possible à implémenter.

La recouvrabilité est une notion orthogonale à la sérialisabilité.



# Annulations en cascade

Un historique ***évite les annulations en cascade*** si toute transaction ne lit à partir des autres transactions que des données validées, i.e. pour toute transaction $T_j$,

Si $T_j$ lit x à partir de $T_i$ et $i \neq j$ alors $C_i < R_j(x)$

Exemples :

| | |
|---|---|
| $W_1(x)\ W_2(y)\ C_1\ R_2(x)\ C_2$ | Evite la cascade |
| $W_1(x)\ W_2(x)\ C_2\ A_1$ | Evite la cascade |
| $W_1(x)\ R_2(x)\ A_1$ | N'évite pas la cascade |
| $W_1(x)\ R_2(x)\ C_2\ C_1$ | N'évite pas la cascade |

Eviter les annulations en cascade simplifie grandement l'implémentation de l'opération d'annulation, et de ce fait améliore les performances du système (qui est l'objectif premier de la concurrence).

**Résultat** : Un historique qui évite les annulations en cascade est recouvrable.

L'inverse n'est pas vrai :

H : $W_1(x)\ R_2(x)\ W_2(y)\ A_1$

H est recouvrable mais n'évite pas les annulations en cascade.

# Exécution strict

Un historique est ***strict*** si toute transaction ne lit ni n'écrit une donnée qui a déjà été modifiée par une autre transaction qui est encore active, i.e. pour toute opération de lecture ou d'écriture $O_j(x)$ :

Si $W_i(x) < O_j(x)$ et $i \neq j$ alors $A_i < O_j(x)$ ou $C_i < O_j(x)$

Exemple :

| | |
|---|---|
| $W_1(x)\ W_2(y)\ C_1\ W_2(x)\ C_2$ | Est strict |
| $W_1(x)\ R_2(y)\ C_1\ R_2(x)\ C_2$ | Est strict |
| $W_1(x)\ R_2(y)\ A_1\ R_2(x)\ C_2$ | Est strict |
| $W_1(x)\ W_2(x)\ C_2\ A_1$ | N'est pas strict |

N'autoriser que des historiques stricts permet d'implémenter l'opération d'annulation avec la technique simple des images avant (voir le contre-exemple ci-dessous).

**Résultat** : Un historique strict évite les annulations en cascade, et il est donc recouvrable.

L'inverse n'est pas vrai :

H : $W_1(x)\ W_2(x)\ C_2\ A_1$

H évite les annulations en cascade mais n'est pas strict.



# Notion de Sérialisabilité

La théorie de la sérialisabilité ne s'intéresse qu'aux transactions validées. Les transactions annulées n'ont aucun effet, et les transactions actives peuvent toujours se finir par une annulation. Cela revient à considérer des historiques formés uniquement de transactions qui sont validées. C'est ***la projection sur les transactions validées*** d'un historique H, notée **C(H)**, qui est la notion pertinente pour l'étude de la sérialisabilité.

La projection validée C(H) est un historique complet défini sur les transactions validées de H.

Un historique complet est ***sériel*** si les transactions qui le composent sont en série, i.e. pour toutes transactions $T_i$ et $T_j$, toutes les opérations de $T_i$ précèdent ou suivent toutes les opérations de $T_j$.

Un historique H est ***sérialisable*** si C(H) est « équivalent » à un historique sériel.

Exemple :

H : $R_1(x)\ R_2(y)\ W_1(y)\ W_2(z)\ C_1\ C_2$

H est sérialisable car C(H)=H est « équivalent » à $T_2T_1$

Simplification d'écriture : Dans la suite, les exemples d'historiques seront présentés sans les opérations de fin de transaction, $A_i$ ou $C_i$, ils sont implicitement terminés par les $C_i$ adéquats.

# Vue-Sérialisabilité

Dans un historique H, une écriture $W_i(x)$ est ***finale*** si elle est la dernière écriture de x produite par une transaction non annulée de H.
On note **FW(H)** l'ensemble des écritures finales de H.

Deux historiques H et H' sont ***vue-équivalents*** si

- Ils sont définis sur les mêmes transactions
- Ils ont les mêmes opérations :
  Op(H) = Op(H')
- Ils ont les mêmes écritures finales :
  FW(H) = FW(H')
- Ils ont les mêmes lectures « à partir de » :
  RF(H) = RF(H')

Un historique H est ***vue-sérialisable*** si C(H) est vue-équivalente à un historique sériel.

**Résultat** : Déterminer si un historique est vue-sérialisable est un problème NP-complet.



# Vue-Sérialisabilité

Exemple :

$T_1$ : $W_1(x)$ $R_1(y)$ $W_1(z)$
$T_2$ : $W_2(x)$ $W_2(y)$
$T_3$ : $R_3(x)$ $W_3(x)$

$H_1$ : $W_1(x)$ $W_2(x)$ $R_1(y)$ $W_2(y)$ $R_3(x)$ $W_1(z)$ $W_3(x)$
$T_3$ lit x à partir de $T_2$
Ecritures finales : $W_3(x)$, $W_2(y)$, $W_1(z)$

$H_2$ : $W_1(x)$ $R_1(y)$ $W_1(z)$ $W_2(x)$ $W_2(y)$ $R_3(x)$ $W_3(x)$
$T_3$ lit x à partir de $T_2$
Ecritures finales : $W_3(x)$, $W_2(y)$, $W_1(z)$

$H_1$ est vue-équivalent à $H_2$
$H_2$ est sériel
Donc $H_1$ est vue-sérialisable

$H_3$ : $W_2(x)$ $W_1(x)$ $R_3(x)$ $R_1(y)$ $W_3(x)$ $W_2(y)$ $W_1(z)$
$T_3$ lit x à partir de $T_1$
Ecritures finales : $W_3(x)$, $W_2(y)$, $W_1(z)$

$H_3$ n'est ni vue-équivalent à $H_1$, ni à $H_2$.
$H_3$ n'est pas vue-sérialisable car

- $W_3(x)$ n'est pas finale dans $T_1T_3T_2$, $T_2T_3T_1$, $T_3T_1T_2$ et $T_3T_2T_1$
- $T_3$ lit x à partir de $T_2$ dans $T_1T_2T_3$
- $T_1$ lit y à partir de $T_2$ dans $T_2T_1T_3$

# Conflit-Sérialisabilité

Dans un historique H, deux opérations sont ***en conflit*** si elles appartiennent à deux transactions non annulées différentes et si elles portent sur une même donnée et si au moins l'une d'entre elle est une écriture et l'autre est une lecture ou une écriture.

Il y a donc trois ordres entre opérations conflictuelles, pour tout i,j, i≠j :

$W_i(x) < W_j(x)$  $W_i(x) < R_j(x)$  $R_i(x) < W_j(x)$

On note **OC(H)** la relation « ordre des conflits » :

$\{(P_i,Q_j) / P_i$ et $Q_j$ sont en conflit et $P_i < Q_j\}$

Deux historiques H et H' sont ***conflit-équivalents*** si

- Ils sont définis sur les mêmes transactions
- Ils ont les mêmes opérations :
  Op(H) = Op(H')
- L'ordre de chaque paire d'opérations en confit est le même dans H et H' :
  OC(H) = OC(H')

Un historique H est ***conflit-sérialisable*** si C(H) est conflit-équivalente à un historique sériel.



# Conflit-Sérialisabilité

Exemple :

$T_1 : R_1(x)\ R_1(y)\ W_1(y)\ W_1(x)$
$T_2 : W_2(x)\ R_2(z)\ W_2(y)$

$H_1 : R_1(x)\ W_2(x)\ R_1(y)\ W_1(y)\ W_1(x)\ R_2(z)\ W_2(y)$
Ordre des conflits :
$OC(H_1) = \{ R_1(x) < W_2(x) ; W_2(x) < W_1(x) ; R_1(y) < W_2(y) ; W_1(y) < W_2(y) \}$

$H_2 : R_1(x)\ R_1(y)\ W_2(x)\ W_1(y)\ R_2(z)\ W_1(x)\ W_2(y)$
Ordre des conflits :
$OC(H_2) = \{ R_1(x) < W_2(x) ; R_1(y) < W_2(y) ; W_1(y) < W_2(y) ; W_2(x) < W_1(x) \}$

$H_3 : W_2(x)\ R_2(z)\ R_1(x)\ W_2(y)\ R_1(y)\ W_1(y)\ W_1(x)$
Ordre des conflits :
$OC(H_3) = \{ W_2(x) < R_1(x) ; W_2(x) < W_1(x) ; W_2(y) < R_1(y) ; W_2(y) < W_1(y) \}$

$H_1$ est confit-équivalent à $H_2$ car $OC(H_1) = OC(H_2)$

$H_3$ n'est pas conflit-équivalent à $H_1$ car $OC(H_3) \neq OC(H_1)$

$H_3$ est conflit-sérialisable car conflit-équivalent à $T_2T_1$

# Conflit-Sérialisabilité

**Résultat** : Si un historique est conflit-sérialisable alors il est vue-sérialisable.

En revanche, il existe des historiques vue-sérialisables qui ne sont pas conflit-sérialisables.

Exemple :

$H_1 : W_1(x)\ W_2(x)\ W_2(y)\ W_1(y)\ W_3(y)\ W_1(z)$

Ecritures finales : $W_1(z)$, $W_2(x)$, $W_3(y)$
Ordre des conflits :
$\{W_1(x) < W_2(x)\ ;\ W_2(y) < W_1(y)\ ;\ W_2(y) < W_3(y)\ ;$
$W_1(y) < W_3(y)\}$

$H_1$ est vue-sérialisable car vue-équivalent à $T_1T_2T_3$

$H_1$ n'est pas conflit-sérialisable car :

Dans $T_1T_2T_3$ et $T_1T_3T_2$ et $T_3T_1T_2$
on a $W_1(y) < W_2(y)$

Dans $T_2T_1T_3$ et $T_2T_3T_1$ et $T_3T_2T_1$
on a $W_2(x) < W_1(x)$



# Graphe de précédence

Le ***graphe de précédence*** d'un historique H, noté GP(H), est un graphe orienté où :

- Les nœuds sont les transactions validées
- Il existe un arc $T_i \rightarrow T_j$ chaque fois qu'une opération de $T_i$ précède et est en conflit avec une opération de $T_j$

Remarque :

$T_i \rightarrow T_j$ et $T_j \rightarrow T_k$ n'implique pas $T_i \rightarrow T_k$

En effet : $R_i(x) < W_j(x)$ et $W_j(x) < R_k(x)$ n'implique pas $T_i \rightarrow T_k$ car $R_i(x)$ et $R_k(x)$ ne sont pas en conflit

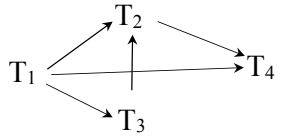
Exemple :

H : $W_1(x)\ R_2(x)\ W_1(y)\ R_2(y)\ R_3(x)\ R_4(y)\ W_4(y)\ W_2(x)$

Ordre des conflits :

$\{W_1(x) < R_2(x)\ ;\ W_1(x) < R_3(x)\ ;\ W_1(x) < W_2(x)$
$W_1(y) < R_2(y)\ ;\ W_1(y) < R_4(y)\ ;\ W_1(y) < W_4(y)\ ;$
$R_2(y) < W_4(y)\ ;\ R_3(x) < W_2(x)\}$

GP(H) :



# Graphe de précédence

**Résultat** : Un historique H est conflit-sérialisable ssi le graphe GP(H) est acyclique.

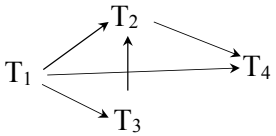
Un ***tri topologique*** d'un graphe acyclique orienté est une extension linéaire de l'ordre partiel sur les sommets déterminé par les arêtes, c'est-à-dire un ordre total compatible avec ce dernier.

Si GP(H) est acyclique, à partir d'un tri topologique, on peut obtenir un historique sériel conflit-équivalent à H.

Exemple :

H : $W_1(x)\ R_2(x)\ W_1(y)\ R_2(y)\ R_3(x)\ R_4(y)\ W_4(y)\ W_2(x)$

GP(H) :




$T_1T_3T_2T_4$ est sériel et conflit-équivalent à H
Donc H est conflit-sérialisable.



# Equivalence par permutation

Deux opérations qui ne sont pas en conflit sont indépendantes, elles peuvent permuter sans modifier le résultat de l'exécution concurrente.

Règles de permutation :

| | | | | |
|---|---|---|---|---|
| R1 : | $R_i(x)\ (R_j(y)$ | $\rightarrow$ | $R_j(y)\ R_i(x)$ | si $i \neq j$ |
| R2 : | $R_i(x)\ W_j(y)$ | $\rightarrow$ | $W_j(y)\ R_i(x)$ | si $i \neq j$ et $x \neq y$ |
| R3 : | $W_i(x)\ R_j(y)$ | $\rightarrow$ | $R_j(y)\ W_i(x)$ | si $i \neq j$ et $x \neq y$ |
| R4 : | $W_i(x)\ W_j(y)$ | $\rightarrow$ | $W_j(y)\ W_i(x)$ | si $i \neq j$ et $x \neq y$ |
| R5 : | $E_i\ O_j$ | $\rightarrow$ | $O_j\ E_i$ | si $i \neq j$ et $E_i \in \{C_i, A_i\}$ |
| R6 : | $O_i\ E_j$ | $\rightarrow$ | $E_j\ O_i$ | si $i \neq j$ et $E_j \in \{C_j, A_j\}$ |

Deux historiques H et H' sont ***équivalents par permutation*** si

- Ils sont définis sur les mêmes transactions
- Ils ont les mêmes opérations : Op(H) = Op(H')
- H peut être transformé en H' par l'application des règles R1, …, R6 un nombre fini de fois

**Résultat** : H est équivalent par permutation à H' ssi H est conflit-équivalent à H'

Exemple :

H : $W_1(x)\ R_2(x)\ W_1(y)\ W_1(z)\ R_3(z)\ W_2(y)\ W_3(y)\ W_3(z)$

$W_1(x)\ W_1(y)\ R_2(y)\ W_1(z)\ R_3(z)\ W_2(y)\ W_3(y)\ W_3(z)$

$W_1(x)\ W_1(y)\ W_1(z)\ R_2(y)\ R_3(z)\ W_2(y)\ W_3(y)\ W_3(z)$

$W_1(x)\ W_1(y)\ W_1(z)\ R_2(y)\ W_2(y)\ R_3(z)\ W_3(y)\ W_3(z)$

$T_1T_2T_3$

H est équivalent par permutation à $T_1T_2T_3$.

Donc H est conflit-sérialisable.

# Notion de verrou

Il existe 2 types de verrous simples :
- Les verrous exclusifs ou verrou d'écriture.
- Les verrous partagés ou verrou de lecture.

Deux verrous V et V' sont dit ***incompatibles*** si lorsqu'une transaction T a posé un verrou V sur une donnée x, aucune autre transaction ne peut poser un verrou V' sur x avant que T n'ait supprimé son verrou.

Incompatibilité des verrous :

| | | Pose possible par $T_2$ sur x | |
|---|---|---|---|
| | | Exclusif | Partagé |
| Posé par $T_1$ sur x | Exclusif | Non | Non |
| | Partagé | Non | Oui |

Seuls les verrous partagés sont compatibles entre eux.

Un verrou exclusif permet de protéger une écriture contre une autre écriture ou une lecture.
Un verrou partagé permet de protéger une lecture contre une écriture.

Primitives de manipulation des verrous :
- $XL_i(x)$ (resp. $SL_i(x)$) : c'est la pose du verrou exclusif (resp. partagé) sur la donnée x.
- $U_i(x)$ : c'est la suppression du verrou qui porte sur la donnée x.

Ces primitives doivent s'exécuter de façon atomique (i.e. le parallélisme est exclu) afin d'assurer l'incompatibilité des verrous.



# Verrouillage

Principe d'un ***protocole simple de verrouillage*** :
- Poser un verrou avant l'accès à une donnée : Avant d'écrire (resp. de lire) une donnée, une transaction $T_i$ doit poser un verrou exclusif (resp. partagé) sur celle-ci.
- Respecter l'incompatibilité des verrous : Si d'autres transactions ont déjà posé un verrou incompatible, la transaction $T_i$ ne peut poser son verrou qu'après la suppression de tous les verrous incompatibles.
- Supprimer le verrou après l'accès à la donnée : Il n'est pas nécessaire que la suppression du verrou ait lieu immédiatement après l'accès à la donnée, mais elle ne peut jamais avoir lieu avant.

Dans sa forme simple, le protocole de verrouillage n'exclut que le parallélisme des opérations de lecture et d'écriture conflictuelles, à condition que l'opération de verrouillage soit atomique.
La sérialisabilité n'est pas assurée : un historique conforme au protocole ci-dessus peut ne pas être sérialisable.

Exemple (mise à jour perdue) :

H : $SL_1(x)$ $SL_2(x)$ $R_1(x)$ $R_2(x)$ $U_1(x)$ $U_2(x)$
$XL_1(x)$ $W_1(x)$ $U_1(x)$ $XL_2(x)$ $W_2(x)$ $U_2(x)$ $C_1$ $C_2$

H respect le protocole simple de verrouillage

Projection de H en éliminant les verrous :

H' : $R_1(x)$ $R_2(x)$ $W_1(x)$ $W_2(x)$ $C_1$ $C_2$

H' n'est pas sérialisable, donc H n'est pas sérialisable.

# Verrouillage

Si la suppression d'un verrou a lieu immédiatement après l'accès (en écriture ou en lecture) de la donnée, on dit que le verrou est un ***verrou court***. Si le verrou est maintenu jusqu'à la fin de la transaction, on le qualifie de ***verrou long***.

La ***conversion de verrou*** :

- Permettre à une même transaction de poser plusieurs types de verrou sur une même donnée.
- Faire en sorte qu'à tout moment, une transaction ne détient qu'un seul verrou sur la donnée
- Lorsqu'une transaction détient un verrou sur une donnée, une nouvelle pose de verrou produit une conversion du verrou existant en un verrou qui agrège l'incompatibilité des 2 verrous.

Remarque : Les poses successives de verrous par une même transaction sur une même donnée vont du verrou le moins restrictif au verrou le plus restrictif, et jamais l'inverse.

Exemple : Une transaction peut commencer par poser un verrou partagé pour protéger la lecture de x, et en suite poser un verrou exclusif pour protéger l'écriture de x. Il y a alors conversion du verrou partagé en un verrou exclusif.

H : $SL_1(x)\ R_1(x)\ \ldots\ XL_1(x)\ W_1(x)\ \ldots\ U_1(x)\ \ldots$

Conversion du verrou partagé en verrou exclusif



# Verrouillage à 2 phases

Le ***protocole de verrouillage à 2 phases*** (2PL) est le protocole simple augmenté de la règle suivante :

> Pour chaque transaction, dès qu'elle supprime un verrou, elle ne pose plus de nouveaux verrous.

Pour chaque transaction, il y a donc 2 phases : une phase de pose de verrous suivie d'une phase de suppression de verrous.

**Résultat** : Un historique conforme à 2PL est conflit-sérialisable.

Exemple :

$T_1$ : $SL_1(x)\ R_1(x)\ XL_1(y)\ W_1(y)\ U_1(x)\ U_1(y)\ C_1$
$T_2$ : $SL_2(y)\ R_2(y)\ XL_2(z)\ W_2(z)\ U_2(y)\ U_2(z)\ C_2$
$T_1$ et $T_2$ sont à 2 phases, la $2^{ème}$ phase est sur fond gris.

$H_1$ : $SL_1(x)\ R_1(x)\ SL_2(y)\ R_2(y)\ XL_2(z)\ W_2(z)\ U_2(y)\ U_2(z)$
$XL_1(y)\ W_1(y)\ U_1(x)\ U_1(y)\ C_1\ C_2$

$H_1$ est conforme à 2PL.
Projection de $H_1$ en éliminant les verrous :
$R_1(x)\ R_2(y)\ W_2(z)\ W_1(y)\ C_1\ C_2$

$H_1$ est conflit-sérialisable car conflit-équivalent à $T_2T_1$.

# Verrouillage à 2 phases

Si l'exécution concurrente n'est pas sérialisable alors il est impossible de produire un historique avec verrous qui soit conforme à 2PL.

Exemple (mise à jour perdue) :
H : $R_1(x)$ $R_2(x)$ $W_1(x)$ $W_2(x)$ $C_1$ $C_2$
H n'est pas sérialisable.

$T_1$' : $SL_1(x)$ $R_1(x)$ $XL_1(x)$ $W_1(x)$ $U_1(x)$ $C_1$
$T_2$' : $SL_2(x)$ $R_2(x)$ $XL_2(x)$ $W_2(x)$ $U_2(x)$ $C_2$
$T_1$' et $T_2$' sont à 2 phases et verrouillent systématiquement les données accédées.

Il est impossible de construite un historique complet défini sur {$T_1$', $T_2$'} qui respecte l'ordre des opérations de H et aussi l'incompatibilité des verrous :

H' : $SL_1(x)$ $R_1(x)$ $SL_2(x)$ $R_2(x)$ ($XL_1(x)$ ou $XL_2(x)$) ?



# Verrouillage à 2 phases

Il existe des historiques conflit-sérialisable qui ne sont pas conforme à 2PL.

Exemple :
H : $R_1(x)$ $W_2(x)$ $W_1(y)$ $W_2(y)$ C1 C2
H est conflit-sérialisable car conflit-équivalent à $T_1T_2$

$T_1$' : $SL_1(x)$ $R_1(x)$ $XL_1(y)$ $W_1(y)$ $U_1(x)$ $U_1(y)$ $C_1$
$T_2$' : $XL_2(x)$ $W_2(x)$ $XL_2(y)$ $W_2(y)$ $U_2(x)$ $U_2(y)$ $C_2$
$T_1$' et $T_2$' sont à 2 phases et verrouillent systématiquement les données accédées.

Il est impossible de construite un historique complet défini sur {$T_1$', $T_2$'} qui respecte l'ordre des opérations de H et aussi l'incompatibilité des verrous :
H' : $SL_1(x)$ $R_1(x)$ $XL_1(y)$ $XL_2(x)$ ?

# 2PL strict

2PL n'assure pas la recouvrabilité : Il existe des historiques conformes à 2PL qui ne sont pas recouvrables.
2PL n'évite pas les annulations en cascade.

Exemple :

$H_1$ : $XL_1(x)\ W_1(x)\ U_1(x)\ SL_2(x)\ R_2(x)\ U_2(x)\ C_2\ A_1$
$H_2$ : $XL_1(x)\ W_1(x)\ U_1(x)\ SL_2(x)\ R_2(x)\ A_1\ U_2(x)\ A_2$
$H_1$ et $H_2$ sont conformes à 2PL.
$H_1$ n'est pas recouvrable.
$H_2$ n'évite pas les annulations en cascade.
Dans les 2 historiques, $T_1$ supprime trop tôt le verrou exclusif qu'elle détient sur x.
Solution : Supprimer les verrous après que l'opération de fin de transaction soit complètement exécutée.

Le ***protocole 2PL strict*** (S2PL) :

- Suivre le protocole 2PL
- Pour chaque transaction $T_i$, imposer que la suppression des verrous d'écriture se fasse après l'exécution complète de l'opération de fin de transaction $C_i$ ou $A_i$.

**Résultat** : Un historique conforme à S2PL est un historique strict.

Un historique conforme à S2PL est donc recouvrable et évite les annulations en cascade.



# S2PL en pratique

Deux gestions possibles de la concurrence :

1) Le SGBD prend en charge le respect de S2PL. Implicitement, avant chaque opération de lecture ou d'écriture, il pose automatiquement le verrou adéquat.
2) L'utilisateur prend en charge le verrouillage au niveau applicatif. Il doit alors se conformer au protocole S2PL.

Mais dans les 2 cas, c'est le SGBD qui prend en charge la suppression des verrous, afin de garantir la recouvrabilité de l'exécution concurrente car, en cas de panne, il doit pouvoir annuler les transactions non terminées afin de rétablir un état cohérent de la base de données.

De plus, c'est le SGBD qui assure le respect de l'incompatibilité des verrous. Il peut immédiatement annuler la transaction qui viole l'incompatibilité des verrous. Mais la méthode la plus utilisée est la mise en attente :

- Une pose de verrou est possible si aucune autre transaction n'a posé un verrou incompatible.
- Sinon, la pose du verrou est retardée et la transaction est mise en attente jusqu'à ce que le dernier verrou incompatible soit supprimé.

Ainsi les opérations de l'exécution concurrente entrante sont réordonnées par la mise en attente de la transaction. Ce réagencement des opérations concurrentes assure dans la mesure du possible la sérialisabilité.
Mais un nouveau problème apparait à cause de la mise en attente croisée de plusieurs transactions : c'est l'***interblocage***.

# Interblocage

La gestion de l'incompatibilité des verrous par la mise en attente des transactions peut produire un interblocage, il faut donc pouvoir détecter/guérir ou éviter cette situation.

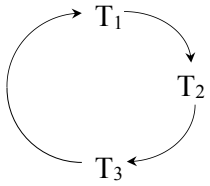
◆ **Détection :**
Le ***graphe d'attente*** (Wait-For-Graph) d'un historique H, noté WFG(H), est un graphe orienté où :

- Les nœuds sont les transactions actives
- Un arc $T_i \rightarrow T_j$ est créé chaque fois qu'une transaction $T_i$ demande un verrou incompatible avec le verrou détenu par une autre transaction $T_j$
- L'arc $T_i \rightarrow T_j$ est supprimé lorsque $T_j$ supprime son verrou.

**Résultat** : Il y a un interblocage ssi le graphe WFG possède un cycle.

Exemple :

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| $SL_1(x)$ | | |
| $R_1(x)$ | $XL_2(y)$ | |
| | $W_2(y)$ | $SL_3(z)$ |
| $SL_1(y)$ | | $R_3(z)$ |
| bloquée | $XL_2(z)$ | |
| | bloquée | $XL_3(x)$ |
| | | bloquée |






# Interblocage

◆ **Guérison :**
Lorsqu'un cycle est détecté dans le graphe WFG, pour corriger la situation, on annule une des transactions qui participe au cycle en utilisant les critères suivants :

- Minimiser le coût : celle qui a le moins de verrous
- Maximiser les ressources libres : celle qui a le plus de verrous
- Eviter la famine : celle qui a été annulée le moins souvent
- Minimiser les cycles : celle qui participe au plus grand nombre de cycles

◆ **Prévention :**
Chaque transaction $T_i$ obtient une estampille $ts(T_i)$ au démarrage.
Lorsque $T_i$ demande un verrou incompatible avec celui détenu par $T_j$, pour éviter l'interblocage, on autorise la mise en attente de $T_i$ en utilisant l'une des deux règles :

- Wait-Die : $T_j$ est plus jeune que $T_i$, $ts(T_j) > ts(T_i)$
- Wound-Wait : $T_j$ est plus veille que $T_i$, $ts(T_j) < ts(T_i)$

Si la mise en attente n'est pas autorisée, c'est toujours la transaction la plus jeune qui est annulée :

- Wait-Die : $T_i$ est annulée
- Wound-Wait : $T_j$ est annulée et $T_i$ pose son verrou

La transaction annulée est relancée avec la même estampille pour éviter la famine (i.e. la même transaction est perpétuellement relancée).

# Interblocage

Exemple (Wait-Die) :

| $T_1$ (ts=1) | $T_2$ (ts=2) | $T_3$ (ts=3) |
|---|---|---|
| $SL_1(x)$ | | |
| $R_1(x)$ | $XL_2(y)$ | |
| | $W_2(y)$ | $SL_3(z)$ |
| | | $R_3(z)$ |
| $SL_1(y)$ | | |
| bloquée | $XL_2(z)$ | |
| | bloquée | $XL_3(x)$ |
| | débloquée ← | annulée |
| | | relancée |
| | | $SL_3(z)$ |
| | | annulée |

Exemple (Wound-Wait) :

| $T_1$ (ts=1) | $T_2$ (ts=2) | $T_3$ (ts=3) |
|---|---|---|
| $SL_1(x)$ | | |
| $R_1(x)$ | $XL_2(y)$ | |
| | $W_2(y)$ | $SL_3(z)$ |
| | | $R_3(z)$ |
| $SL_1(y)$ → | annulée | |
| $R_1(y)$ | | |
| | relancée | $XL_3(x)$ |
| | $XL_2(y)$ | bloquée |
| | bloquée | |

**Caractéristique** : La méthode Wait-Die produit plus d'annulations que Wound-Wait, mais le travail annulé est moins important.



# Verrouillage Multi-niveau

**Motivation :** Pour éviter le problème des fantômes, verrouiller les lignes (i.e. les données) est insuffisant. Il faut verrouiller la table pour interdire les insertions par les autres transactions.
Cela introduit une nouvelle problématique : le verrouillage à plusieurs niveaux car les transactions vont être autorisées à verrouiller à différents niveaux, en fonction de leurs besoins, afin de permettre la plus forte concurrence.

**Granularité du verrouillage :**
Le ***granule*** est la structure de données sur laquelle s'applique la pose de verrous.

Base
↓
Table
↓
Bloc
↓
Ligne

En SQL, seuls deux niveaux sont disponibles pour poser des verrous : La table et la ligne.

Remarque : Plus la taille du granule est petite, meilleure est la concurrence.

**Idée de base** : La pose d'un verrou V sur un granule verrouille implicitement tous ses descendants. Ainsi toute autre transaction ne pourra pas poser un verrou V' incompatible sur un des descendants.
Par exemple : la pose d'un verrou exclusif sur une table interdit, aux autres transactions, la pose d'un verrou exclusif ou partagé sur une ligne de la table.

# Verrouillage Multi-niveau

**Nouveaux modes de verrouillage :**
Lorsqu'une transaction veut poser un verrou sur un nœud de la hiérarchie, il faut contrôler que ce nœud ne soit pas déjà verrouillé de façon implicite par la pose d'un verrou incompatible sur un de ses ancêtres. Pour accomplir simplement cela, il suffit que la transaction pose un verrou intentionnel sur tous les ancêtres, et le contrôle est alors assuré grâce à la compatibilité des verrous.

Deux nouveaux verrous :
- Verrou d'intention de lecture (IS)
- Verrou d'intention d'écriture (IX)

Quand un nœud est verrouillé avec un verrou IS (resp. IX), cela signifie qu'un descendant du nœud est verrouillé avec un verrou S (resp. un verrou S ou X).

**Compatibilité des verrous :**

| | IS | IX | S | X | SIX |
|---|---|---|---|---|---|
| IS | Oui | Oui | Oui | Non | Oui |
| IX | Oui | Oui | Non | Non | Non |
| S | Oui | Non | Oui | Non | Non |
| X | Non | Non | Non | Non | Non |
| SIX | Oui | Non | Non | Non | Non |

Le verrou SIX est l'agrégat des verrous S et IX. Il est introduit pour permettre la conversion des verrous S et IX en verrou SIX. L'opération de conversion est alors une opération fermée sur les verrous.



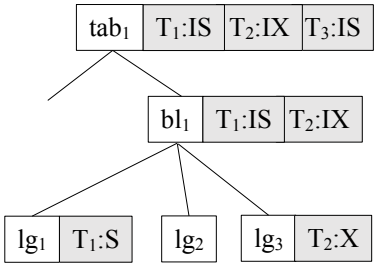
# Verrouillage Multi-niveau

**Protocole de verrouillage multi-niveau** (MGL) :
- Verrouillage explicite de la racine jusqu'au granule :
  Si x n'est pas la racine, pour poser sur x un verrou S ou IS (resp. un verrou X, IX ou SIX), $T_i$ doit détenir un verrou IS ou IX (resp. un verrou IX ou SIX) sur le père de x et aucune autre transaction ne doit détenir un verrou incompatible sur x.
- Verrouillage implicite des descendants du granule :
  Pour lire (resp. écrire) x, $T_i$ doit détenir un verrou S ou X (resp. un verrou X) sur x ou sur un de ses ancêtres.
- Déverrouillage du granule jusqu'à la racine :
  $T_i$ ne peut pas supprimer un verrou qu'elle détient sur x, si elle détient un verrou sur un des fils de x.

**Résultat** : Si les transactions suivent le protocole MGL alors elles ne peuvent pas détenir des verrous incompatibles (explicites ou implicites) sur une même donnée.

Exemple :

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| $ISL_1(tab_1)$ | | |
| $ISL_1(bl_1)$ | | |
| $SL_1(lg_1)$ | $IXL_2(tab_1)$ | |
| $R_1(lg_1)$ | $IXL_2(bl_1)$ | |
| | $XL_2(lg_3)$ | $ISL_3(tab_1)$ |
| | $W_2(lg_3)$ | $SL_3(bl_1)$ |
| | | bloquée |



# Estampillage

Principe du contrôleur de concurrence TO (Timestamp Ordering) :

- Ordonner par le temps de début : A chaque transaction $T_i$ est associée une unique estampille $ts(T_i)$ lorsqu'elle démarre.
- Rejeter une opération si elle arrive trop tard, sinon la traiter : Quand une opération $p_i$ de lecture ou d'écriture arrive après une opération $q_j$ qui est en conflit avec $p_i$ (i.e. $q_j < p_i$, $p_i$ et $q_j$ sont en conflit), si $ts(T_i) < ts(T_j)$, alors $p_i$ est rejetée, sinon elle est traitée.

**Propriété** : Dans un historique H conforme à TO, l'ordre des conflits suit l'ordre de début des transactions, i.e. pour toutes opérations $p_i$ et $q_j$ de H, on a :

Si $(p_i, q_j) \in OC(H)$ alors $ts(T_i) < ts(T_j)$

**Résultat** : Si un historique est conforme à TO alors il est conflit-sérialisable.

**Caractéristiques :**

Une exécution conforme à TO est équivalente à une exécution sérielle dans laquelle les transactions apparaissent dans l'ordre de leurs estampilles.

TO n'assure pas la recouvrabilité.

Exemple :

H : $W_1(x)\ R_2(x)\ W_2(y)\ C_2\ A_1$

H est conforme à TO car $W_1(x) < R_2(x)$ et $ts(T_1) < ts(T_2)$

H n'est pas recouvrable



# Estampillage

**Mise en œuvre** :

A chaque donnée x, on associe 2 estampilles :

- tsR(x) qui est l'estampille de la plus jeune transaction qui a lu x
- tsW(x) qui est l'estampille de la plus jeune transaction qui a écrit x

Traitement d'une lecture $R_i(x)$ :

- Si $ts(T_i) \geq tsW(x)$ alors la lecture est acceptée et $tsR(x) \leftarrow \max(ts(T_i), tsR(x))$
- Sinon $T_i$ est annulée et relancée avec une nouvelle estampille.

Traitement d'une écriture $W_i(x)$ :

- Si $ts(T_i) \geq tsR(x)$ et $ts(T_i) \geq tsW(x)$ alors l'écriture est acceptée et $tsW(x) \leftarrow ts(T_i)$
- Sinon $T_i$ est annulée et relancée avec une nouvelle estampille.

# Estampillage

Exemple :

H : $W_1(z)\ W_3(x)\ C_1\ R_2(x)\ W_3(y)\ C_3\ R_2(y)\ C_2$

| | | | x | y | z |
|---|---|---|---|---|---|
| $T_1$ ts=1 | $T_2$ ts=2 | $T_3$ ts=3 | tsR=0 tsW=0 | tsR=0 tsW=0 | tsR=0 tsW=0 |
| $W_1(z)$ | | | | | tsW=1 |
| | | $W_3(x)$ | tsW=3 | | |
| $C_1$ | | | | | |
| | $R_2(x)$ annulée | | | | |
| | relancée ts=4 $R_2(x)$ | | tsR=4 | | |
| | | $W_3(y)$ | | tsW=3 | |
| | | $C_3$ | | | |
| | $R_2(y)$ | | | tsR=4 | |
| | $C_2$ | | | | |



# Multi-version

◆ **Motivation :**

Faire en sorte que les lectures ne soient pas bloquées (ou annulées) par les écritures, en gérant plusieurs versions d'une même donnée.

Exemple :

$T_0 : W_0(x)\ W_0(y)\ C_0$

$T_1 : W_1(x)\ R_1(y)\ W_1(u)\ C_1$

$T_2 : R_2(x)\ W_2(y)\ W_2(v)\ C_2$

H : $W_0(x)\ W_0(y)\ C_0\ W_1(x)\ R_2(x)\ W_2(y)\ R_1(y)\ W_1(u)\ C_1\ W_2(v)\ C_2$

H n'est pas vue-sérialisable car

- Dans $T_0T_1T_2$, $T_1$ lit y à partir de $T_0$. Dans H, $T_1$ lit y à partir de $T_2$
- Dans $T_0T_2T_1$, $T_2$ lit x à partir de $T_0$. Dans H, $T_2$ lit x à partir de $T_1$

En multi-version, $T_1$ peut lire la version de y créée par $T_0$.
Alors H sera équivalent à $T_0T_1T_2$, donc sérialisable.

◆ **Principe :**

Le contrôleur de concurrence produit un historique multi-version (MV) à partir d'un historique à une version (1V).

L'historique MV représente les opérations sur les versions des données que doit traiter le gestionnaire de données.

L'historique 1V représente l'interprétation de l'historique MV que perçoit l'utilisateur.

# Historique MV

◆ **Historique multi-version (MV) sur $\{T_1,\ldots,T_n\}$ :**
Les opérations des transactions $T_1,\ldots, T_n$, qui sont des opérations à une version, sont traduites en opérations multi-version suivant les règles ci-dessous.

- Une écriture sur une donnée x produit une nouvelle copie (une version) de x. On note les versions de x par $x_i$, $x_j$, … où l'indice est le numéro de la transaction qui a produit la version de x. L'écriture $W_i(x)$ de $T_i$ sera traduite par $W_i(x_i)$.
- Une lecture indique la version de x à lire. La lecture $R_j(x)$ de $T_j$ sera traduite par $R_j(x_i)$ où $x_i$ est la version de x qui est lue par $T_j$. Cela suppose que la version $x_i$ existe au moment de la lecture, i.e. que $T_i$ ait créé $x_i$ et ne soit pas annulée avant la lecture de $T_j$ (i.e. $W_i(x_i) < R_j(x_i)$ et $A_i \not< R_j(x_i)$). De plus, si $T_j$ a créé une version de x avant qu'elle lise x ($W_j(x) < R_j(x)$) alors elle doit lire cette version (i=j).
- Les opérations de fin de transaction ($C_i$ et $A_i$) sont traduites à l'identique.
- L'historique MV préserve l'ordre des opérations spécifié par chaque transaction (i.e. si $P_i$ précède $Q_i$ dans $T_i$ alors la traduction de $P_i$ précède celle de $Q_i$ dans l'historique).
- L'historique MV est recouvrable (i.e. le commit de $T_j$ est après les commits des transactions à partir desquelles $T_j$ a lu).



# Historique MV

Exemple (En reprenant l'exemple de l'introduction) :
$T_0 : W_0(x)\ W_0(y)\ C_0$ $T_1 : W_1(x)\ R_1(y)\ W_1(u)\ C_1$
$T_2 : R_2(x)\ W_2(y)\ W_2(v)\ C_2$

$H : W_0(x)\ W_0(y)\ C_0\ W_1(x)\ R_2(x)\ W_2(y)\ R_1(y)\ W_1(u)\ C_1\ W_2(v)\ C_2$

$H_1 : W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_2(x_1)\ W_2(y_2)\ R_1(y_2)\ W_1(u_1)\ C_1\ W_2(v_2)\ C_2$

$H_2 : W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_2(x_1)\ W_2(y_2)\ R_1(y_0)\ W_1(u_1)\ C_1\ W_2(v_2)\ C_2$

$H_1$ et $H_2$ sont des historiques MV sur $\{T_0, T_1, T_2\}$

Remarque :
- Comme le montre l'exemple, pour représenter une exécution concurrente, il existe plusieurs historiques MV là où un seul historique 1V est possible.
- $H_2$ est l'historique MV qui représente l'exécution concurrente décrite par H et qui est sérialisable.

# Historiques MV et 1V

Un historique MV et un historique 1V n'ont pas les mêmes opérations, et n'ont pas les mêmes conflits entre opérations.

Dans un historique MV, il n'y a qu'un seul type de conflit entre une écriture qui précède une lecture. Deux opérations d'écriture ne sont pas en conflit car elles écrivent 2 versions différentes. Une lecture qui précède une écriture n'est pas en conflit avec celle-ci car, par définition, une lecture ne peut pas lire une version qui n'a pas encore été créée.

Il n'est donc pas acceptable de baser l'équivalence de deux historiques MV et 1V sur la seule notion de conflit.

Exemple :

$H_{MV}$ : $W_0(x_0)$ $C_0$ $W_1(x_1)$ $C_1$ $R_2(x_0)$ $W_2(y_2)$ $C_2$
$H_{1V}$ : $W_0(x)$ $C_0$ $W_1(x)$ $C_1$ $R_2(x)$ $W_2(y)$ $C_2$

L'ordre du seul conflit de $H_{MV}$ est $W_0(x_0) < R_2(x_0)$.
Les opérations correspondantes dans $H_{1V}$ respectent cet ordre : $W_0(x) < R_2(x)$.
Or $H_{MV}$ et $H_{1V}$ ne peuvent pas être considérés comme équivalents car dans $H_{MV}$, $T_2$ lit x à partir de $T_0$ alors que dans $H_{1V}$, $T_2$ lit x à partir de $T_1$. $T_2$ peut lire 2 valeurs différentes, et le résultat de $H_{MV}$ peut être différent de celui de $H_{1V}$.



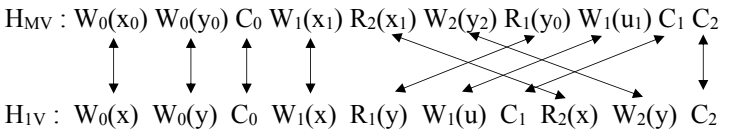
# Historiques MV et 1V

◆ **Equivalence entre un historique MV et un historique 1V :**

Un historique MV $H_{MV}$ et un historique 1V $H_{1V}$ sont ***équivalents*** si :

- $H_{MV}$ et $H_{1V}$ sont sur le même ensemble de transactions
- Il y a une correspondance un-à-un (i.e. une bijection) entre les opérations de $H_{MV}$ et celles de $H_{1V}$
- $H_{MV}$ et $H_{1V}$ ont les mêmes lectures « à partir de ».

Exemple :




$H_{MV}$ et $H_{1V}$ sont équivalents car il y a bien une bijection entre les opérations, représentée par les flèches ci-dessus, et dans les 2 historiques on a :

- $T_1$ lit y à partir de $T_0$
- $T_2$ lit x à partir de $T_1$

# Equivalence MV

Dans un historique MV, on a les propriétés suivantes :

- $T_j$ lit x à partir de $T_i$ ssi $R_j(x_i)$ appartient à l'historique
- Toute écriture est finale
- Les seules opérations en conflit correspondent aux lectures « à partir de »

Deux historiques MV H et H' sont ***équivalents*** si

- H et H' sont définis sur les mêmes transactions et ont les mêmes opérations
- H et H' ont les mêmes lectures « à partir de »

Sur les historiques MV, l'équivalence de vue coïncide avec l'équivalence de conflit.

**Propriété** : Deux historiques définis sur les mêmes transactions sont équivalents s'ils ont exactement les mêmes opérations.

L'équivalence entre deux historiques MV est une condition triviale : Ils sont différents uniquement sur l'ordre des opérations, mais les lectures « à partir de », les conflits et les écritures sont les mêmes.



# Sérialisabilité MV

Un historique MV est ***sériel*** si les transactions qui le composent sont en série.

Pour définir la sérialisabilité d'un historique MV, l'équivalence avec un historique MV sériel est une condition insuffisante.

En effet :

$H_1$ : $W_0(x_0)$ $C_0$ $R_1(x_0)$ $R_2(x_0)$ $W_1(x_1)$ $C_1$ $W_2(x_2)$ $C_2$
$H_2$ : $W_0(x_0)$ $C_0$ $R_1(x_0)$ $W_1(x_1)$ $C_1$ $R_2(x_0)$ $W_2(x_2)$ $C_2$

$H_1$ est équivalent à $H_2$ qui est sériel. Or $H_2$ ne peut pas être interprété comme un historique 1V, car dans celui-ci, $T_2$ ne peut pas lire x à partir de $T_0$.

Un historique MV est ***1V-sériel*** s'il est sériel et si pour toute lecture $R_j(x_i)$ (i.e. $T_j$ lit x à partir de $T_i$ ) on a :

- soit i=j
- soit $T_i$ est la dernière transaction précédant $T_j$ qui a produit la version de x.

Exemple :

$H_1$ : $W_0(x_0)$ $C_0$ $R_1(x_0)$ $W_1(x_1)$ $C_1$ $R_2(x_0)$ $C_2$
$H_2$ : $W_0(x_0)$ $C_0$ $R_2(x_0)$ $C_2$ $R_1(x_0)$ $W_1(x_1)$ $C_1$

$H_1$ est sériel mais pas 1V-sériel.
$H_2$ est 1V-sériel.

# Sérialisabilité MV

Un historique MV H est ***1V-sérialisable*** (OneCopy-Serializable) si sa projection validée C(H) est équivalente à un historique MV 1V-sériel.

Exemple (En reprenant l'exemple de l'introduction) :

$T_0 : W_0(x)\ W_0(y)\ C_0$ $T_1 : W_1(x)\ R_1(y)\ W_1(u)\ C_1$
$T_2 : R_2(x)\ W_2(y)\ W_2(v)\ C_2$

$H : W_0(x)\ W_0(y)\ C_0\ W_1(x)\ R_2(x)\ W_2(y)\ R_1(y)\ W_1(u)\ C_1\ W_2(v)\ C_2$

$H_1 : W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_2(x_1)\ W_2(y_2)\ R_1(y_2)\ W_1(u_1)\ C_1\ W_2(v_2)\ C_2$

$H_2 : W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_2(x_1)\ W_2(y_2)\ R_1(y_0)\ W_1(u_1)\ C_1\ W_2(v_2)\ C_2$

$H_3 : W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_1(y_0)\ W_1(u_1)\ C_1\ R_2(x_1)\ W_2(y_2)\ W_2(v_2)\ C_2$

$H_1$ n'est pas 1V-sérialisable car il n'existe aucun historique MV sériel contenant les mêmes opérations que $H_1$.

$H_3$ est 1V-sériel et est équivalent à $H_2$. $H_2$ est 1V-sérialisable.

Remarque : Un historique MV sériel peut être 1V-sérilisable sans être 1V-sériel. Exemple :

$H_1 : W_0(x_0)\ C_0\ R_1(x_0)\ W_1(x_1)\ C_1\ R_2(x_0)\ C_2$
$H_2 : W_0(x_0)\ C_0\ R_2(x_0)\ C_2\ R_1(x_0)\ W_1(x_1)\ C_1$

$H_1$ est sériel mais pas 1V-sériel. $H_1$ est équivalent à $H_2$ qui est 1V-sériel. Donc $H_1$ est 1V-sérialisable.



# Sérialisabilité MV

**Résultat** : Un historique MV H sur un ensemble de transactions T est 1V-sérialisable ssi sa projection validée C(H) est équivalente à un historique 1V sur T qui est sériel.

Ce résultat montre que la 1V-sérialisabilité est un critère correct de sérialisabilité.

Exemple (En reprenant l'exemple de l'introduction) :

$T_0 : W_0(x)\ W_0(y)\ C_0$ $T_1 : W_1(x)\ R_1(y)\ W_1(u)\ C_1$
$T_2 : R_2(x)\ W_2(y)\ W_2(v)\ C_2$

$H : W_0(x)\ W_0(y)\ C_0\ W_1(x)\ R_2(x)\ W_2(y)\ R_1(y)\ W_1(u)\ C_1\ W_2(v)\ C_2$

$H_2 : W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_2(x_1)\ W_2(y_2)\ R_1(y_0)\ W_1(u_1)\ C_1\ W_2(v_2)\ C_2$

$H_3 : W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_1(y_0)\ W_1(u_1)\ C_1\ R_2(x_1)\ W_2(y_2)\ W_2(v_2)\ C_2$

$H_4 : W_0(x)\ W_0(y)\ C_0\ W_1(x)\ R_1(y)\ W_1(u)\ C_1\ R_2(x)\ W_2(y)\ W_2(v)\ C_2$

Précédemment nous avons vu que $H_2$ est 1V-sérialisable car il est équivalent à $H_3$ qui est 1V-sériel.
$H_3$ est équivalent à $H_4$ qui est un historique 1V et qui est sériel ($H_4=T_0T_1T_2$).
Par transitivité de l'équivalence, $H_2$ est bien équivalent à $H_4$ qui est un historique 1V et qui est sériel.

# Graphe de sérialisation

Les versions d'une même donnée sont totalement ordonnées. Mais cet ordre dépend de la stratégie adoptée par le contrôleur de concurrence. Il peut suivre l'ordre de début ou de fin des transactions. On note << l'ordre obtenu en faisant l'union des ordres des versions pour chaque donnée.

Le ***graphe de sérialisation*** d'un historique MV H pour un ordre des versions <<, noté MVSG(H,<<), est un graphe orienté où :

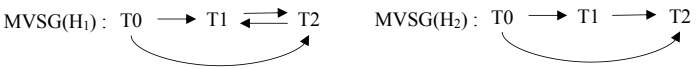
- Les nœuds sont les transactions validées
- Un arc $T_i \rightarrow T_j$ est présent pour chaque opération de lecture $R_j(x_i)$ de C(H) avec $i \neq j$
- Pour chaque $W_i(x_i)$ et $R_k(x_j)$ de C(H) où i,j et k sont distincts :
  - Si $x_i << x_j$ alors l'arc $T_i \rightarrow T_j$ est présent
  - Sinon l'arc $T_k \rightarrow T_i$ est présent

**Résultat** : Un historique MV H est 1V-sérialisable ssi il existe un ordre sur les versions << tel que le graphe de sérialisation MVSG(H,<<) soit sans cycle.

Exemple :

$H_1$ : $W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_2(x_1)\ W_2(y_2)\ R_1(y_2)\ W_1(u_1)\ C_1\ W_2(v_2)\ C_2$ Avec $x_0<<x_1$ et $y_0<<y_2$

$H_2$ : $W_0(x_0)\ W_0(y_0)\ C_0\ W_1(x_1)\ R_2(x_1)\ W_2(y_2)\ R_1(y_0)\ W_1(u_1)\ C_1\ W_2(v_2)\ C_2$ Avec $x_0<<x_1$ et $y_0<<y_2$






# MV ordonné par estampille

Principe du contrôleur de concurrence MVTO (MultiVersion Timestamp Ordering) :

- A chaque transaction $T_i$ est associée une unique estampille $ts(T_i)$ lorsqu'elle démarre.
- Une lecture $R_i(x)$ est convertie en $R_i(x_k)$ où $x_k$ est la version créée par la transaction $T_k$ telle que $ts(T_k)$ est la plus grande estampille inférieure ou égale à $ts(T_i)$.
- Une écriture $W_i(x)$ est rejetée et produit l'annulation de la transaction $T_i$ si une lecture $R_j(x_k)$ a déjà eu lieu telle que $ts(T_k) < ts(T_i) < ts(T_j)$. Sinon, l'écriture $W_i(x)$ est convertie en $W_i(x_i)$.
- De plus, pour que l'exécution soit recouvrable, il faut retarder le commit de $T_i$ de façon à ce qu'il ait lieu après le commit des transactions à partir desquelles $T_i$ a lu.

L'ordre des versions suit l'ordre des estampilles :
$x_i << x_j$ ssi $ts(T_i) < ts(T_j)$

**Résultat** :
Tout historique MV produit par MVTO est 1V-sérialisable

# MV ordonné par estampille

**Caractéristiques :**

MVTO assure la sérialisabilité par l'ordre d'entrée des transactions dans le système :

Si $T_i$ doit précéder $T_j$ alors $ts(T_i) < ts(T_j)$

MVTO ne rejette jamais une lecture, mais une lecture peut provoquer une mise en attente du commit.

MVTO n'évite pas les annulations en cascade. Seule la recouvrabilité est assurée.



# MV ordonné par estampille

**Mise en œuvre :**

Chaque version de donnée est étiquetée par un intervalle [wts, rts] où wts est l'estampille de la transaction qui a créé la version et rts est l'estampille de la plus récente transaction qui a lu la version. A la création de la version rts = wts.

Pour traiter une lecture $R_i(x)$, il faut parcourir l'ensemble des versions de x pour trouver celle dont l'intervalle [wts, rts] est tel que wts est la plus grande estampille inférieure ou égale à $ts(T_i)$. Si $rts < ts(T_i)$ alors $rts = ts(T_i)$.

Pour traiter une écriture $W_i(x)$, il faut parcourir l'ensemble des versions de x pour trouver celle dont l'intervalle [wts, rts] est tel que wts est la plus grande estampille inférieure à $ts(T_i)$. Si $ts(T_i) < rts$ alors l'écriture est rejetée et produit l'annulation de la transaction Ti. Sinon la version $x_i$ est créée avec l'intervalle $[ts(T_i), ts(T_i)]$.

Il faut mettre en attente le commit de $T_i$ jusqu'au commit de toutes les transactions qui ont créé des versions lues par $T_i$.

Les versions $x_k$ antérieures à une version $x_i$ (i.e. $ts(T_k) < ts(T_i)$) sont inutiles et peuvent être supprimées si aucune transaction active $T_j$ n'a son estampille $ts(T_j) < ts(T_i)$.

# MV ordonné par estampille

**Exemple :**

$H_{1V}$ : $W_1(x)$ $W_2(x)$ $W_2(y)$ $W_1(y)$ $C_1$ $C_2$

| $T_1$ ts=1 | | $T_2$ ts=2 | | $x_1$ | $x_2$ | $y_1$ | $y_2$ |
|---|---|---|---|---|---|---|---|
| $W_1(x)$ | $W_1(x_1)$ | | | wts=1 rts=1 | | | |
| | | $W_2(x)$ | $W_2(x_2)$ | | wts=2 rts=2 | | |
| | | $W_2(y)$ | $W_2(y_2)$ | | | | wts=2 rts=2 |
| $W_1(y)$ $C_1$ | $W_1(y_1)$ $C_1$ | | | | | wts=1 rts=1 | |
| | | $C_2$ | $C_2$ | | | | |

$H_{MV}$ : $W_1(x_1)$ $W_2(x_2)$ $W_2(y_2)$ $W_1(y_1)$ $C_1$ $C_2$

$H_{1V}$ : $R_1(x)$ $W_1(x)$ $R_2(x)$ $R_2(y)$ $C_2$ $R_1(y)$ $W_1(y)$ $C_1$

| $T_1$ ts=1 | | $T_2$ ts=2 | | $x_0$ | $x_1$ | $y_0$ | $y_1$ |
|---|---|---|---|---|---|---|---|
| | | | | wts=0 rts=0 | | wts=0 rts=0 | |
| $R_1(x)$ | $R_1(x_0)$ | | | wts=0 rts=1 | | | |
| $W_1(x)$ | $W_1(x_1)$ | | | | wts=1 rts=1 | | |
| | | $R_2(x)$ | $R_2(x_1)$ | | wts=1 rts=2 | | |
| | | $R_2(y)$ | $R_2(y_0)$ | | | wts=0 rts=2 | |
| | | $C_2$ | attente | | | | |
| $R_1(y)$ | $R_1(y_0)$ | | | | | | |
| $W_1(y)$ | rejetée $A_1$ | | cascade $A_2$ | | | | |

$H_{MV}$ : $R_1(x_0)$ $W_1(x_1)$ $R_2(x_1)$ $R_2(y_0)$ $R_1(y_0)$ $A_1$ $A_2$



# Niveau d'Isolation

Ensemble, la sérialisabilité et la recouvrabilité définissent le niveau d'isolation des transactions le plus exigeant, qui évite tous les problèmes liés à l'exécution concurrente. Mais le prix à payer est au niveau des performances :

- Mécanismes couteux (au pire NP-complet) pour assurer la sérialisabilité la plus élevée.
- Mécanismes plus simples (verrous) mais la concurrence est réduite.

La recouvrabilité est incontournable à partir du moment où l'annulation est le mécanisme central sur lequel s'appuie le SGBD pour assurer l'intégrité des données en toute situation, et surtout en cas de panne.

En partant du constat que certaines applications n'exigent pas un niveau d'isolation aussi fort, d'autres niveaux d'isolation peuvent être définis permettant d'atteindre des performances plus élevées, au détriment de la sérialisabilité, qui n'est plus assurée.

De plus, au lieu d'utiliser un niveau global d'isolation pour toute une application, individualiser l'isolation au niveau des différents traitements effectués par l'application, permet des performances encore plus élevées.

Bien sûr, le critère de correction est celui qui assure que l'application et ses traitements évitent les problèmes de concurrence malgré le manque d'isolation. Cela aboutit à une gestion non-automatique de concurrence dont les développeurs sont les garants de la validité.

# Isolation SQL

La norme SQL définit plusieurs niveaux d'isolation pour une transaction en les caractérisant par les anomalies qu'ils permettent d'éviter.

***A1 (Lecture sale)*** : $T_1$ modifie une ligne. Puis $T_2$ lit la ligne avant que $T_1$ exécute un **`COMMIT`**. Si $T_1$ exécute un **`ROLLBACK`**, alors $T_2$ aura lu une ligne qui n'a jamais été validée et qui peut être considérée comme n'avoir jamais existé.
***A2 (Lecture non-reproductible)*** : $T_1$ lit une ligne. Puis $T_2$ modifie ou supprime cette ligne et exécute un **`COMMIT`**. Si $T_1$ essaie de relire la ligne, elle obtient la ligne modifiée ou elle découvre que la ligne a été supprimée.
***A3 (Fantôme)*** : $T_1$ lit un ensemble de lignes qui satisfait une condition de clause **`WHERE`**. Puis $T_2$ exécute une insertion d'une ou plusieurs lignes qui satisfont la condition utilisée par $T_1$ et exécute un **`COMMIT`**. Si $T_1$ répète la lecture avec la même condition, elle obtient un ensemble de lignes différent.

| Niveau d'isolation | Anomalie autorisée | Anomalie interdite |
|---|---|---|
| **`READ UNCOMMITTED`** | A1, A2, A3 | Aucune ? |
| **`READ COMMITED`** | A2, A3 | A1 |
| **`REPEATABLE READ`** | A3 | A1, A2 |
| **`SERIALIZABLE`** | Aucune ? | A1, A2, A3 |

Problèmes :

- **`SERIALIZABLE`** SQL =?= sérialisable (vue, conflit, 1V ?)
- Double interprétation : le paragraphe en entier correspond à une interprétation stricte, la partie sur fond gris est l'interprétation relâchée.



# Comparaison des niveaux d'isolation

Pour comparer les niveaux d'isolation, il y a 2 axes :

- Les différents niveaux de sérialisabilité (sérialisabilité générale, vue-sérialisabilité, conflit-sérialisabilité, 1V-sérialisabilité, verrouillage à 2 phases – 2PL, 2PL strict, estampillage – TO, estampillage multiversions – MVTO) sont comparés en fonction des historiques sérialisables qu'ils autorisent, car tous n'autorisent aucun historique non sérialisable, c'est le critère de sérialisabilité. Cette façon de faire n'est pas adaptée à la comparaison de niveaux d'isolation qui autorisent des historiques non sérialisables.

- Les niveaux d'isolation qui n'assurent pas la sérialisabilité autorisent des anomalies, c'est-à-dire des historiques non sérialisables. Il est donc pertinent de les comparer en comparant les ensembles d'historiques non sérialisables (i.e. les anomalies) qu'ils autorisent (ou interdisent).

  Bien sûr, lorsque 2 niveaux d'isolation autorisent le même ensemble d'historiques non sérialisables, il est possible qu'ils se différencient fortement par les historiques sérialisables qu'ils autorisent. Et donc, si l'un autorise beaucoup moins d'historiques sérialisables que l'autre, alors la concurrence correcte autorisée par le premier est beaucoup plus faible que celle autorisée le second.

# Comparaison des niveaux d'isolation

Un niveau d'isolation $N_1$ ***est équivalent*** à un autre niveau d'isolation $N_2$, noté $N_1 \approx N_2$, si les ensembles des historiques non sérialisables autorisés par N1 et par N2 sont identiques, c'est-à-dire : $\bar{S}(N_1) = \bar{S}(N_2)$.
Où $\bar{S}(N)$ est l'ensemble des historiques non sérialisables autorisés par le niveau d'isolation N.

Un niveau d'isolation $N_1$ ***est plus faible*** qu'un autre niveau d'isolation $N_2$ (ou $N_2$ ***est plus fort*** que $N_1$), noté $N_1 \prec N_2$, si tous les historiques non sérialisable autorisés par $N_2$ sont aussi autorisés par $N_1$, et il existe un historique non sérialisable autorisés par $N_1$ qui est interdit par $N_2$, c'est-à-dire : $\bar{S}(N_2) \subsetneq \bar{S}(N_1)$.

Remarques :

- On a : $A(N) = \bar{S}(N)$ et $\bar{A}(N) = \bar{S} - \bar{S}(N)$.
  Où $A(N)$ (resp. $\bar{A}(N)$) est l'ensemble des anomalies autorisées (resp. interdites) par le niveau d'isolation N et $\bar{S}$ est l'ensemble des historiques non sérialisables.
- Si $N_1 \prec N_2$ alors $N_1$ autorise des anomalies qui sont interdites par $N_2$, c'est-à-dire : $\bar{A}(N_1) \subsetneq \bar{A}(N_2)$.

Deux niveaux d'isolation $N_1$ et $N_2$ ***sont incomparables***, noté $N_1 \succ\prec N_2$ si pour chaque niveau d'isolation, il existe un historique non sérialisable autorisé par l'un et interdit par l'autre.



# Isolation par Verrou

***P0 (Ecriture sale)*** : $T_1$ modifie une donnée. $T_2$ modifie cette même donnée avant que $T_1$ soit terminée.

$W_1(x) \ldots W_2(x) \ldots (C_1 \text{ ou } A_1)\ldots$

***P1 (Lecture sale)*** : Une transaction $T_1$ modifie une donnée. Une autre transaction $T_2$ lit cette même donnée avant que $T_1$ soit terminée.

$W_1(x) \ldots R_2(x) \ldots (C_1 \text{ ou } A_1)\ldots$

***P2 (Lecture non-reproductible)*** : Une transaction $T_1$ lit une donnée. Une autre transaction $T_2$ modifie cette même donnée avant que $T_1$ soit terminée.

$R_1(x) \ldots W_2(x) \ldots (C_1 \text{ ou } A_1)\ldots$

***P3 (Fantôme)*** : Une transaction $T_1$ lit un ensemble de données satisfaisant une condition (i.e. un prédicat P). Les données lues par $T_1$ peuvent servir à calculer un agrégat. Une autre transaction $T_2$ écrit (i.e. insère, modifie ou supprime) une donnée qui satisfait la condition, avant que $T_1$ soit terminée.

$R_1(P) \ldots W_2(y \text{ in } P) \ldots (C_1 \text{ ou } A_1)\ldots$

***P4 (Mise à jour perdue)*** : Une transaction $T_1$ lit une donnée. Une autre transaction $T_2$ modifie cette même donnée (éventuellement à partir d'une lecture de cette donnée). Puis $T_1$ modifie aussi cette donnée en utilisant sa lecture précédente. Ensuite $T_1$ se termine en validant.

$R_1(x) \ldots W_2(x) \ldots W_1(x) \ldots C_1 \ldots$

# Isolation par Verrou

| Niveau Isolation | Phénomène Interdit | Verrou Lecture sur Donnée / Collection | Verrou Ecriture |
|---|---|---|---|
| Degré 0 | Ecriture Non Atomique | Aucun | Court |
| Degré 1 | P0 | Aucun | Long |
| Degré 2 | P0, P1 | Court sur les 2 | Long |
| Curseur Stable | P0, P1, P4 | Jusqu'à la fin du curseur sur donnée<br>Court sur collection | Long |
| Degré 2.99 | P0, P1, P4, P2 | Long sur donnée<br>Court sur collection | Long |
| Degré 3 | P0, P1, P4, P2, P3 | Long sur les 2 | Long |

Degré 0 assure uniquement l'atomicité des écritures
Degré 1 assure que l'opération d'annulation peut être implémentée avec la technique simple des images avant.

En supposant que les niveaux d'isolation définis dans la norme SQL interdisent le phénomène P0, on obtient les comparaisons suivantes :

Degré 1 ≈ **`READ UNCOMMITTED`**
Degré 2 ≈ **`READ COMMITTED`**
Degré 2.99 ≈ **`REPEATABLE READ`**
**`READ COMMITTED`** ≺ Curseur Stable ≺ **`REPEATABLE READ`**

Degré 3 est conforme à 2PL + Verrouillage Multi-Niveaux. Il assure donc la conflit-sérialisabilité.

**`SERIALIZABLE`** SQL ≺ Degré 3



# Isolation par Snapshot

L'isolation par Snapshot (photo instantanée) est basée sur les multi-versions : les écritures sur une même donnée x créent des versions de x.
Une transaction $T_i$ lit toujours à partir d'un cliché (snapshot), photo instantanée de l'état persistant (validé) de la base de données, pris avant le début de la transaction. Pour marquer ce début, moment où le cliché est pris, on utilise l'opération $S_i$ (Start $T_i$), l'implémentation peut utiliser une estampille.

Un historique MV H est conforme à l'***Isolation par Snapshot*** **(SI)** s'il suit les 2 règles suivantes :

**SI-Version :** Quand une transaction $T_i$ lit une donnée x à partir d'une autre transaction, elle lit la plus récente version de x validée avant le début de $T_i$, c'est-à-dire :
$R_i(x_j) \in H$ avec $i \neq j$ si $W_j(x_j) < C_j < S_i$ et s'il n'existe aucune opération $W_k(x_k)$ et $C_k$ ($k \neq j$) tel que $W_k(x_k) < S_i$ et $C_j < C_k < S_i$

**SI-Ecriture :** Deux transactions concurrentes (i.e. actives en même temps) ne peuvent pas écrire sur une même donnée, c'est-à-dire :
Pour toute transaction $T_i$ et $T_j$ si $S_i < S_j < C_i$ alors $W_i(x_i) \notin H$ ou $W_j(x_j) \notin H$ pour toute donnée x.

**Avantage SI** : Les transactions en lecture seule ne sont ni bloquées, ni annulées, et elles ne bloquent pas les opérations de mise à jour concurrentes.
**Résultat** : Un historique conforme à SI est un historique strict. Il est donc recouvrable et évite les annulations en cascade. Et l'annulation peut être implémentée avec les images avant.

# Isolation par Snapshot

Il existe des historiques 1V-séralisables qui ne sont pas conforme à SI.

Exemple :
$H_{MV}$ : $S_0\ W_0(x_0)\ W_0(y_0)\ C_0\ S_1\ R_1(x_0)\ S_2\ R_2(y_0)\ W_2(y_2)\ C_2\ R_1(y_2)\ W_1(y_1)\ C_1$
$H_{MV}$ n'est pas conforme à SI car les 2 règles sont violées :

- $R_1(y)$ est associé à $W_2(y)$ alors qu'il aurait dû être associé à $W_0(y)$ ($W_2(y)$ est une écriture qui se produit et qui est validée après le début de $T_1$)
- $T_1$ et $T_2$ sont concurrentes et écrivent sur la donnée y

$H_{T0T2T1}$ : $T_0\ S_2\ R_2(y_0)\ W_2(y_2)\ C_2\ S_1\ R_1(x_0)\ R_1(y_2)\ W_1(y_1)\ C_1$
$H_{T0T2T1}$ est 1V-sériel et est équivalent à $H_{MV}$.
Donc $H_{MV}$ est 1V-sérialisable.

La règle SI-Ecriture a été introduite pour éviter l'anomalie de mise à jour.

Exemple (mise à jour perdue) :
$T_0$ : $S_0\ W_0(x)\ C_0$ $T_1$ : $S_1\ R_1(x)\ W_1(x)\ C_1$
$T_2$ : $S_2\ R_2(x)\ W_2(x)\ C_2$

$H_{MV}$ : $T_0\ S_1\ R_1(x_0)\ S_2\ R_2(x_0)\ W_1(x_1)\ W_2(x_2)\ C_1\ C_2$
$H_{T0T1T2}$ : $T_0\ S_1\ R_1(x_0)\ W_1(x_1)\ C_1\ S_2\ R_2(x_0)\ W_2(x_2)\ C_2$
$H_{T0T2T1}$ : $T_0\ S_2\ R_2(x_0)\ W_2(x_2)\ C_2\ S_1\ R_1(x_0)\ W_1(x_1)\ C_1$

$H_{MV}$ est équivalent à $H_{T0T1T2}$ et à $H_{T0T2T1}$. Mais ni $H_{T0T1T2}$, ni $H_{T0T2T1}$ ne sont 1V-sériels. $H_{MV}$ n'est pas 1V-sérialisable.
$H_{MV}$ ne respecte pas la règle SI-Ecriture car $T_1$et $T_2$ écrivent sur x.



# Implémentation de SI-Ecriture

Deux façons d'implémenter la règle SI-Ecriture :

**Le premier qui valide gagne (First commited wins) :**
Quand une transaction $T_i$ démarre elle obtient une estampille de début $tsS_i$. Lorsqu'elle est prête à valider, elle obtient une estampille de commit $tsC_i$, qui doit être plus grande que toute estampille existante $tsS_k$ et $tsC_k$.

$T_i$ valide avec succès s'il n'existe aucune transaction $T_k$ telle que $tsS_i < tsC_k < tsC_i$ et qui a écrit une donnée aussi écrite par $T_i$. Sinon $T_i$ est annulée.

**Le premier qui modifie gagne (First updater wins) :**
Supposons deux transactions $T_1$ et $T_2$ concurrentes telles que $T_1$ a écrit x, et $T_2$ veut aussi écrire x. Les écritures sont protégées par des verrous exclusifs. Alors $T_2$ ne peut pas écrire x avant que $T_1$ ait relâché son verrou sur x. Il y a donc trois possibilités :

1) Si $T_2$ est en attente d'écrire x lorsque $T_1$ valide, $T_2$ est immédiatement annulée.
2) Si $T_1$ valide avant que $T_2$ essaie d'écrire x, $T_2$ est annulée quand elle tente d'effectuer l'écriture.
3) Si $T_1$ relâche son verrou parce qu'elle est annulée, $T_2$ est autorisé à écrire x.

Remarque : Au lieu d'annuler complétement la transaction $T_2$, il est possible de rejeter uniquement l'opération d'écriture (implémentation Oracle).

# Anomalies sous SI

***A5b (Ecriture croisée)*** : Une transaction écrit sur une donnée y, un calcul effectué à partir d'une donnée x, et une autre transaction effectue le travail inverse (i.e. écriture de x à partir de y).

$R_1(x) \dots R_2(y) \dots W_1(y) \dots W_2(x) \dots C_1\ C_2$

Exemple (violation de contrainte) :
$T_0 : S_0\ W_0(x_0)\ W_0(y_0)\ C_0$ $T_1 : S_1\ R_1(x)\ R_1(y)\ W_1(x)\ C_1$
$T_2 : S_2\ R_2(x)\ R_2(y)\ W_2(y)\ C_2$

$H_{MV} : T_0\ S_1\ R_1(x_0)\ S_2\ R_2(x_0)\ R_1(y_0)\ R_2(y_0)\ W_1(x_1)\ C_1\ W_2(y_2)\ C_2$
$H_{MV}$ est conforme au protocole SI
$H_{MV}$ n'est pas 1V-sérilisable car ni $H_{T0T1T2}$ ni $H_{T0T2T1}$ ne sont 1V-sériels

$H_{T0T1T2} : T_0\ S_1\ R_1(x_0)\ R_1(y_0)\ W_1(x_1)\ C_1\ S_2\ R_2(x_0)\ R_2(y_0)\ W_2(y_2)\ C_2$
N'est pas 1V-sériel car $T_2$ ne lit pas la version la plus récente de x.

$H_{T0T2T1} : T_0\ S_2\ R_2(x_0)\ R_2(y_0)\ W_2(y_2)\ C_2\ S_1\ R_1(x_0)\ R_1(y_0)\ W_1(x_1)\ C_1$
N'est pas 1V-sériel car $T_1$ ne lit pas la version la plus récente de y.

$H_{1V} : T_0\ S_1\ R_1(x)\ S_2\ R_2(x)\ R_1(y)\ R_2(y)\ W_1(x)\ C_1\ W_2(y)\ C_2$
Est équivalent à $H_{MV}$ et n'est pas conflit-sérialisable

Illustration : x + y > 0 est un invariant (contrainte à satisfaire)
$T_1 : S_1\ R_1(x,50)\ R_1(y,50)\ W_1(x,-40)\ C_1$
Au début : 50 + 50 > 0 A la fin : - 40 + 50 > 0
$T_2 : S_2\ R_2(x,50)\ R_2(y,50)\ W_2(y,-40)\ C_2$
Au début : 50 + 50 > 0 A la fin : 50 – 40 > 0

$H_{MV} : T_0\ S_1\ R_1(x_0,50)\ S_2\ R_2(x_0,50)\ R_1(y_0,50)\ R_2(y_0,50)\ W_1(x_1,-40)\ C_1\ W_2(y_2\text{-}40)\ C_2$
Au début : 50 + 50 > 0 A la fin : - 40 - 40 $\not>$ 0



# Anomalies sous SI

Au début, on pensait que le fait de lire à partir d'un état persistant de la base de données était suffisant pour garantir qu'aucune anomalie ne se produise lorsque la transaction est en lecture seule, mais ce n'est pas le cas.

***A6 (Anomalie en lecture seule)*** : Il faut faire intervenir 3 transactions. Une transaction $T_3$ en lecture seule est enchâssée dans deux autres transactions $T_1$ et $T_2$ :

$R_2(x)\ W_1(x)\ C_1\ R_3(x)\ R_3(y)\ C_3\ W_2(y)\ C_2$

Exemple :
$H_{MV} : T_0\ S_2\ R_2(x_0)\ R_2(y_0)\ S_1\ R_1(x_0)\ W_1(x_1)\ C_1\ S_3\ R_3(x_1)\ R_3(y_0)\ C_3\ W_2(y_2)\ C_2$
$H_{MV}$ est conforme à SI.
$H_{MV}$ n'est pas 1V-sérialisable car :

- $H_{T0T1T2T3}$, $H_{T0T2T1T3}$ et $H_{T0T2T3T1}$ ne sont pas 1V-sériel car $W_2(y_2) < R_3(y_0)$ et $T_3$ ne lit pas y à partir de $T_2$.
- $H_{T0T3T2T1}$ et $H_{T0T3T1T2}$ ne sont pas des historiques MV car $R_3(x_1) < W_1(x_1)$.
- $H_{T0T1T3T2}$ n'est pas 1V-sériel car $W_1(x_1) < R_2(x_0)$ et $T_2$ ne lit pas x à partir de $T_1$.

Illustration : x et y représentent des comptes joints
$T_1$ : Ajoute 20 à x
$T_2$ : Soustrait 10 à y. Puis soustrait 1 si x + y < 0
$T_3$ : Affiche x et y
$H : T_0\ S_2\ R_2(x_0,0)\ R_2(y_0,0)\ S_1\ R_1(x_0,0)\ W_1(x_1,20)\ C_1\ S_3\ R_3(x_1,20)\ R_3(y_0,0)\ C_3\ W_2(y_2,-11)\ C_2$
$T_3$ affiche x=20 et y=0. Impossible avec une exécution sérielle. H et $T_2T_1$ produisent x=20 et y=-11.

# Anomalies sous SI

L'anomalie P3(Fantôme) est possible sous SI (A3 est impossible).
Exemple :
Soit la table Affectations (IdEmp, Tache, DateT, NbHeures)

$T_1$ :
```
select sum(NbHeures) into NbH from Affectations
where IdEmp = 1 and DateT = d1;
if NbH + 4 <= 8 then
    insert Affectations values(1,'T3',d1,4);
end if;
commit;
```

$T_2$ :
```
select sum(NbHeures) into NbH from Affectations
where IdEmp = 1 and DateT = d1;
if NbH + 5 <= 8 then
    insert Affectations values(1,'T4',d1,5);
end if;
commit;
```

On suppose qu'aucune tâche n'est affectée à l'employé 1.
Soit l'exécution suivante :

| $T_1$ | $T_2$ |
|---|---|
| select sum(NbHeures) | |
| from Affectations | select sum(NbHeures) |
| where IdEmp=1 | from Affectations |
| and DateT=d1 ; | where IdEmp=1 |
| | and DateT=d1 ; |
| insert into Affectations | |
| values(1,'T3',d1,4) ; | insert into Affectations |
| | values(1,'T4',d1,5) ; |
| commit ; | commit; |

Les 2 transactions réussissent et NbHeures = 9



# Caractéristiques de SI

**Résultat** : Les seules anomalies autorisées sous SI sont :
- A5b – Ecriture croisée
- A6 – Anomalie de lecture seule
- P3 – Fantôme

**Comparaison avec les autres niveaux d'isolation** :

Degré 2 (≈ **`READ COMMITTED`**) ≺ SI ≺ Degré 3
A3(Fantôme SQL) est possible sous Degré 2, mais impossible sous SI.
A5b(Ecriture croisée) est possible sous SI, mais impossible sous Degré 3

SI ≻≺ Degré 2.99 (≈ **`REPEATABLE READ`**)
A3(Fantôme SQL) est possible sous Degré 2.99, mais impossible sous SI.
A5b(Ecriture croisée) est possible sous SI, mais impossible sous Degré 2.99

SI ≈ **`SERIALIZABLE`** SQL
A1(Lecture sale) impossible sous les 2.
A2(Lecture non-reproductible) impossible sous les 2.
A3(Fantôme) impossible sous les 2.

# Oracle

◆ **Verrouillage automatique ligne et table :**

- Verrouillage automatique en mode exclusif de chaque ligne concernée par un ordre de mise à jour (`INSERT, DELETE, UPDATE`) et en mode `ROW EXCLUSIVE` de la table.
- Transactions à deux phases strictes.
- Aucun verrou n'est posé automatiquement par un ordre de lecture.

◆ **Isolation Snapshot au niveau d'une instruction :**

Oracle assure que les lectures faites par une instruction DML (`SELECT, INSERT, DELETE, UPDATE`) sont réalisées sur l'état de la base avant l'exécution de l'instruction. Les lectures faites par une instruction DML ignorent :

- toute modification faite par une autre transaction et non confirmée,
- toute modification faite par une autre transaction et confirmée pendant l'exécution de cette instruction.

Implémentation de la règle SI-Ecriture par « le premier qui modifie gagne ». Si une instruction viole cette règle, elle est annulée et relancée automatiquement.

◆ **Conclusion :**

- Au niveau d'une transaction, le mode par défaut est **`READ COMMITTED`** (les écritures sales (P0) et les lectures sales (P1 ou A1) ne sont pas possibles).
- En plus, au niveau d'une instruction DML, les lectures non-reproductibles (A2) et les fantômes (A3) ne sont pas possibles.



# Oracle

◆ **Mécanismes proposés pour gérer la concurrence :**

- Isolation Snapshot au niveau de la transaction avec :
  ```
  SET TRANSACTION READ ONLY
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
  ```
- Verrouillage explicite de lignes en mode exclusif :
  ```
  SELECT ... FOR UPDATE [OF col,...,col]
  ```
- Verrouillage explicite d'une table :
  ```
  LOCK TABLE table IN mode MODE [NOWAIT]
  ```

  Mode de verrouillage :
  ```
  EXCLUSIVE, SHARE, ROW SHARE,
  ROW EXCLUSIVE, SHARE ROW EXCLUSIVE
  ```

  Compatibilité des modes de verrouillage :

| compatible / acquis | Exclusive | Share | Row Share Select for update | Row Ex. Insert, Delete, Update | Sh. Row Ex. |
|---|---|---|---|---|---|
| Exclusive | Non | Non | Non | Non | Non |
| Share | Non | Oui | Oui | Non | Non |
| Row Share Select for update | Non | Oui | Oui | Oui | Oui |
| Row Ex. Insert, Delete, Update | Non | Non | Oui | Oui | Non |
| Sh. Row Ex. | Non | Non | Oui | Non | Non |