Contraintes d'intégrité

Contrainte d'intégrité

Contrainte d'intégrité statique

- respectée pour chacun des états de la BD
- mécanisme déclaratif
  - PRIMARY KEY, UNIQUE, NOT NULL, DOMAIN, FOREIGN KEY, CHECK, ASSERTION
- procédural
  - TRIGGER (SQL:1999)

Contrainte d'intégrité dynamique

- contrainte sur changements d'états (un changement -> un changement -> etc..)
- référence aux états successifs de la base
- TRIGGER

Contrainte colonne

- Types SQL
  - INTEGER
  - CHAR
  - ...
- NOT NULL
- CHECK
- CREATE DOMAIN (SQL, pas disponible pour Oracle)

NOT NULL

```
salaireEmp INTEGER NOT NULL
```

- Par défaut : NULL autorisé

CHECK

- Définition de domaine pour un attribut
- Par intension (conditions nécessaire et suffisante pour appartenir à l'ensemble)

```
salaireEmp INTEGER NOT NULL
CHECK(salaireEmp > 1200 AND salaireEmp < 10000)
```

- Par extension (liste exhaustive des valeurs de l'ensemble)

```
color CHAR(1)
CHECK (color IN ('R', 'G', 'B'))
```


Clé primaire (PRIMARY KEY)

Colonnes clés primaires: NOT NULL et Indexées

```
IdClient INTEGER PRIMARY KEY
```

```
PRIMARY KEY (IdClient, idProduit, dateCommande)
```

UNIQUE

- occurrence unique dans la table de chaque valeur de la colonne (ou des colonnes)
- colonne(s) indexée(s)
- clé candidate

```
idEmp           INTEGER,
job             VARCHAR(64),
idDept          INTEGER,
PRIMARY KEY     (idEmp),
UNIQUE          (job,idDept),
```


Contrainte d'intégrité référentielle (FOREIGN KEY ... REFERENCES ...)

```
CREATE TABLE emp (
 ...
 idDept INTEGER,
 FOREIGN KEY (idDept) REFERENCES
  Dept(idDept),
 ...
```

- `idDept` de `emp` fait référence à la clé primaire `idDept` de la table `dept`
- La table `dept` doit d'abord être créée

Gestion de la contrainte d'intégrité référentielle

- Tentative de mise à jour de la clé primaire

```
CREATE TABLE Commande
(noCommande     INTEGER         NOT NULL,
 dateCommande   DATE            NOT NULL,
 noClient       INTEGER         NOT NULL,
 PRIMARY KEY (noCommande),
 FOREIGN KEY (noClient) REFERENCES Client(noClient)
)
```

```
DELETE FROM Client WHERE noClient = 10
```

- Options
  - `NO ACTION`: pas de mot clé `Oracle`, comportement par défaut
  - `ON DELETE CASCADE`: `Oracle`, tout n-uplet référençant la ligne supprimée est aussi supprimée
  - `ON DELETE SET NULL`: `Oracle`, valeur de l'attribut mis à null
  - `SET DEFAULT`: pas disponible sur `Oracle`
  - `ON UPDATE`

CHECK intra-ligne

```
CHECK (age > 18    and  sal > 0)      ?

CHECK ( (sal + NVL(prime,0)) > 1500)      ?
```

- Plusieurs colonnes de la même ligne peuvent être utilisées simultanément dans une contrainte CHECK

Check inter-ligne d'une même table

- Compare le n-uplet inséré à plusieurs lignes de la table

```
CHECK (sal > SELECT MIN(prime)  FROM emp)) ?
```

- Vérifié uniquement pour la ligne touchée
  - La contrainte peut être violée (mise à jour)
- Pas supporté par Oracle

CHECK inter-tables

- Concerne plusieurs tables

```
CHECK (age >=
  (   SELECT ageMinimumEmployé FROM conventionEntreprise
      WHERE année=2008 ) )    ?
```

- Vérifié uniquement pour la ligne touchée
  - La contrainte peut être violée
- Pas supporté par Oracle

Nom de contrainte (clause CONSTRAINT)

```
CONSTRAINT contNoClient
CHECK(IdClient > 0 AND IdClient < 100000)
```

- Facilite l'identification de la contrainte non respectée à l 'exécution
- Facilite la supression, désactivation/activation de contrainte

Modification contrainte

```
ALTER TABLE table  <constraint_clause>,...
 [{ENABLE|DISABLE} ALL TRIGGERS];

<constraint_clause>:
   ADD CONSTRAINT  ...

   DROP [CONSTRAINT <constraint_name> [CASCADE]] | [PRIMARY KEY|UNIQUE]
   [ENABLE|DISABLE] [CONSTRAINT <constraint_name>

   MODIFY [CONSTRAINT <constraint_name>] |  [PRIMARY KEY|UNIQUE]
                  <constrnt_state>

   RENAME CONSTRAINT <constraint_name> TO <constraint_name>

<constrnt_state>:
    [[NOT] DEFERRABLE] [INITIALLY {IMMEDIATE|DEFERRED)]
       ENABLE|DISABLE]  [VALIDATE|NOVALIDATE]
```

- `VALIDATE | NOVALIDATE`: vérifié ou non sur la table existante
- `ENABLE NOVALIDATE`: vérifié sur les nouvelles insertions mais pas sur les données existantes
- `DEFERRABLE`: contrainte déférée, la contrainte n'est vérifiée qu'au moment de la validation des modification (`INITIALLY` indique le comportement par défaut déférée ou pas)

Déclencheur

Déclencheur (TRIGGER)

- Procédure exécutée au niveau serveur

- BD active

- Permet le maintien de contraintes d'intégrité
  - statiques: alternatives aux mécanismes déclaratifs type CHECK par exemple, préférer les mécanismes déclaratif s'il est possible de les utiliser
  - dynamiques

- Permet le maintien d'éléments dérivés
  - colonne dérivée (contenue d'une colonne calculée à partir d'une ou plusieurs autres colonnes)
  - tables dérivée (copie de tables pour base répartie, table d'archivage, ...)
  - sécurité (ex: refus de modification), audit des opérations,

- Généralement, un déclencheur est associé à un événement sur une table
- Sous ORACLE, un déclencheur (trigger) peut être associé à pratiquement tout événement de la base (DDL, startup, logon, ...)

TRIGGER: Procédure déclenchée

- Utilisé pour spécifier des contraintes plus complexes que les contraintes statiques telles que:
  - Contraintes portant sur plusieurs tables
  - Contraintes nécessitant l'utilisation de sous-requêtes

- Permet d'utiliser des traitement procéduraux:
  écriture de procédures PL/SQL, JAVA, ...

- Déclenché par un événement particulier:
  - Le trigger est généralement associé à une table précise,
  - Il est alors déclenché en réaction à un événement sur cette table: INSERT, UPDATE ou DELETE
  - Il est possible de spécifier des conditions supplémentaires: clause WHEN
  - Possible déclenchement en cascade: un trigger peut déclencher d'autres triggers en cascade
    Attention au cycle de déclencheur:
    trigger A déclenche trigger B qui déclencher trigger A qui ....

- Résultat du trigger
  Le trigger se termine correctement s'il ne soulève pas d'exception et si les modifications effectuées ne violent pas d'autres contraintes (autre triggers ou contraintes statiques)
  Si le trigger est interrompu, toutes les opérations effectuées sont annulées (rollback implicite)

TRIGGER: Rappel syntaxe

```
CREATE [OR REPLACE] TRIGGER nom-trigger
       BEFORE | AFTER
       INSERT OR UPDATE [OF column] OR DELETE ON nom-table
       [FOR EACH ROW [WHEN (condition)]]
       bloc d'instructions pl/sql
```

- INSERT OR UPDATE [OF column] OR DELETE ON nom-table
  - précise le ou les événements provoquant l'exécution du trigger

- BEFORE | AFTER
  - spécifie l'exécution de la procédure comme étant avant (BEFORE) ou après (AFTER) l'exécution de l'événement déclencheur
  - BEFORE : vérification de l'insertion
  - AFTER : copie ou archivage des opérations

- [FOR EACH ROW]
  - indique que le trigger est exécuté pour chaque ligne modifiée/insérée/supprimée
  - si cette clause est omis, le trigger est exécuté une seul fois pour chaque commande UPDATE, INSERT, DELETE, quelque soit le nombre de lignes modifiées, insérées ou supprimées.

- [WHEN (condition)]
  conditions optionnelles permettant de restreindre le déclenchement du trigger
  !!! Ne peut contenir de requêtes
  (exemple: WHEN emp.sal > 5000)

TRIGGER: référence aux attributs des n-uplets modifiés

Dans le code PL/SQL associé à un trigger de niveau ligne, on peut accéder aux valeurs des attributs de la ligne modifiée, avant et après la modification:
=> utilisation des variables :

- :old (sauf dans WHEN => old)
- :new (sauf dans WHEN => new)
- Possibilité d'utiliser d'autres noms
  (clause REFERENCING NEW AS nouveauNom
  OLD AS nouveauNom)

• Pour un trigger sur INSERT
  les valeurs des attributs du n-uplet inséré sont dans
  :new.<nom attribut>
• Pour un trigger sur UPDATE
  les valeurs de l'ancien n-uplet sont dans :old.<nom attribut>
  les valeurs du n-uplet après modification sont dans :new.<nom attribut>
• Pour un trigger sur DELETE
  les valeurs des attributs du n-uplet supprimé sont dans
  :old.<nom attribut>

TRIGGER: ligne/instruction

Déclencheur de ligne (spécifié par FOR EACH ROW),
- **Exécutés une fois pour chaque ligne affectée** par l'événement spécifié.
- Permet **d'accéder aux valeurs des lignes affectées** avant (:old) et après (:new) l'opération.
- Si le triggers est précisé comme étant déclenché avant l'événement (before) les valeurs du n-uplet peuvent être modifiées avant leur insertion dans la base de données (par ex :new.attribut := :new.attribut + 1).
- Il est possible de spécifier des **conditions** de déclenchement (WHEN) pouvant porter sur les valeurs des anciens/nouveaux n-uplet

Déclencheur d'instruction (par défaut)
- **Exécuté une seule fois** avant ou après la totalité de l'événement, indépendamment du nombre de lignes affectés
- Restrictions: ne permet pas d'accéder ou de modifier les valeurs des lignes affectées avant ou après l'événement (:old.nomcol et :new.nomcol) ni de spécifier des conditions WHEN

TRIGGER: remarques

- Mutation : tant que les modifications (INSERT, UPDATE ou DELETE) d'une table ne sont pas terminées (modifications totalement terminées et validées), celle-ci est dite en cours de mutation.

  Un trigger ne peut lire ou modifier une table en cours de mutation.

  => un trigger AFTER un bloc d'instruction (pas de clause FOR EACH ROW), peut modifier la table associée à l'événement déclencheur.

- Validation : un déclencheur ne peut ni exécuter d'instruction COMMIT ou ROLLBACK, ni appeler de fonction, procédure ou sous-programme de package invoquant ces instructions.

TRIGGER: bloc d'instruction

Bloc PL/SQL standard

```
DECLARE
  Déclaration de variables et constantes avec leur type
BEGIN
  Bloc d'instructions PL/SQL
END
```

Le bloc d'instructions PL/SQL peut contenir:

– des blocs spécifiant des actions différentes fonction de l'événement déclencheur
- IF INSERTING THEN bloc d'instructions pl/sql END IF
- IF UPDATING THEN bloc d'instructions pl/sql END IF
- IF DELETING THEN bloc d'instructions pl/sql END IF

– des Instructions SQL
  – SELECT, INSERT, UPDATE, DELETE, … mais pas de COMMIT et ROLLBACK

– Instructions de contrôle de flux (IF, LOOP, WHILE, FOR)

– Générer des exceptions
  raise_application_error(code_erreur,message)
    code_erreur compris entre -20000 et -20999 (sinon code d'erreur oracle)

– Faire appel à des procédures et fonctions PL/SQL

TRIGGER: Activation / désactivation / suppression

Désactivation du trigger :
```
ALTER TRIGGER <nomTrigger> DISABLE ;
```

Activation de déclencheur :
```
ALTER TRIGGER <nomTrigger> ENABLE ;
```

Suppression de déclencheur :
```
DROP TRIGGER <nomTrigger> ;
```

Exemple TRIGGER : maintien d'une contrainte d'intégrité dynamique

Empêcher une augmentation du prix d'un circuit de plus de 10% du prix actuel

```
CREATE TRIGGER circuitRestreindreAugmentationPrix
BEFORE UPDATE OF prixInscription ON Circuit
REFERENCING
      OLD AS rowAvant
      NEW AS rowApres
FOR EACH ROW
WHEN(   rowApres.prixInscription >
        rowAvant.prixInscription *1.1)
BEGIN
      -- souleverUneException;
      RAISE_APPLICATION_ERROR(-20200,
          'Augmentation prix circuit trop importante') ;
END ;
```

Exception Oracle:
RAISE_APPLICATION_ERROR(code, message)
-> ROLLBACK implicite

Borner l'augmentation de prix

- Modifier la valeur de l'attribut prix d'un n-uplet avant son insertion dans la table

```
CREATE TRIGGER circuitBornerAugmentationPrix
BEFORE UPDATE OF prixInscription ON Circuit
REFERENCING
     OLD AS rowAvant
     NEW AS rowApres
FOR EACH ROW
WHEN( rowApres.prixInscription >
      rowAvant.Inscription*1.1)
BEGIN
     :rowApres.prixInscription :=
       :rowAvant.prixInscription * 1.1;
END;
```

Exemple TRIGGER: contrainte d'intégrité statique

NbPlaceDisponible d'un circuit toujours compris entre 0 et 100

```
CREATE TRIGGER PlaceCircuitVerifier
BEFORE INSERT OR UPDATE OF nbPlaceDisponible
    ON Circuit
FOR EACH ROW
WHEN ( new.nbPlaceDisponible < 0) OR
     ( new.nbPlaceDisponible > 100)
BEGIN
     -- souleverUneException;
END
```

CHECK préférable

Exemple TRIGGER: vérification multi-tables

Lors d'une nouvelle livraison, la quantité à livrer ne peut dépasser la quantité en stock disponible

```
CREATE TRIGGER verifierQuantiteEnStock
BEFORE INSERT ON Livraison
FOR EACH ROW
DECLARE
 QuantStock v INTEGER;
BEGIN
 SELECT   quantitéEnStock INTO QuantStock_v
 FROM     Article
 WHERE    IdArticle = :new.IdArticle;

 IF (:new.quantiteLivree > QuantStock_v )
     -- souleverUneException;
 END IF ;
END
```

Exemple TRIGGER: limitation de modification

La relation Livraison contient les attributs idLivraison, idCommande, idArticle et quantiteLivraison

Sécurité: on n'autorise les modifications uniquement sur l'attribut quantitelivraison

Toutes les autres modifications provoquent une exception

```
CREATE TRIGGER EmpecherModifLivraison
BEFORE UPDATE OF IdLivraison, IdCommande, IdArticle
       ON Livraison
BEGIN
     -- souleverUneException;
END
```

Exemple trigger: colonnes liées

- Le nombre d'article en stock est ajusté en fonction des livraisons reçues/envoyées

```
CREATE TRIGGER AI_AjusterQuantitéEnStock
AFTER INSERT ON Livraison
FOR EACH ROW
BEGIN
     UPDATE Article
     SET quantiteEnStock = quantiteEnStock -
      :new.quantiteLivree
     WHERE IdArticle = :new.IdArticle;
END
```

```
CREATE TRIGGER AU AjusterQuantiteStock
AFTER UPDATE OF quantiteLivree ON Livraison
FOR EACH ROW
BEGIN
     UPDATE Article
     SET quantiteEnStock = quantiteEnStock -
      ( :new.quantiteLivree- :old.quantiteLivree)
     WHERE IdArticle = :old.IdArticle;
END
```

TRIGGER: Ordre d'exécution

Ordre d'exécution des TRIGGER ?

- BEFORE avant AFTER,...
- entre TRIGGER de même type ?
  - forcer un ordre en les combinant

Combiner plusieurs TRIGGER:

- vérifier quel est l 'événement déclencheur
- pour ORACLE, tester IF INSERTING, DELETING, UPDATING

Attention à l'ordre d'exécution des déclencheurs !!!

TRIGGER: Ordre d'exécution global

```
Exécuter les TRIGGER BEFORE STATEMENT
Pour chaque ligne touchée par l'opération
      Exécuter les TRIGGER BEFORE ROW
      Exécuter l'opération
      Exécuter les TRIGGER AFTER ROW
Fin pour
Exécuter les TRIGGER AFTER STATEMENT
```

TRIGGER: limites

- peuvent être complexes à coder
- pas de respect de standard:
  contraintes particulières aux langages de codage
- pas de mise à jours sur la table affectée

TRIGGER: exemple de déclenchement

```
SQL> CREATE OR REPLACE TRIGGER BUArticleBornerAugPrix
  2  BEFORE UPDATE OF prixUnitaire ON Article
  3  REFERENCING
  4             OLD AS ligneAvant
  5             NEW AS ligneAprès
  6  FOR EACH ROW
  7  WHEN (ligneAprès.prixUnitaire > ligneAvant.prixUnitaire*1.1)
  8  BEGIN
  9             :ligneAprès.prixUnitaire := :ligneAvant.prixUnitaire*1.1;
 10  END;
 11  /

Déclencheur créé.
```

```
SQL> -- Test du TRIGGER BUArticleBornerAugPrix
SQL> SELECT * FROM Article WHERE IdArticle = 10
  2  /

IDARTICLE DESCRIPTION          PRIXUNITAIRE QUANTITÉENSTOCK
--------- -------------------- ------------ ---------------
       10 Boule de cèdre              10,99              10

SQL> UPDATE    Article
  2  SET       prixUnitaire = 15.99
  3  WHERE     IdArticle = 10
  4  /

1 ligne mise à jour.

SQL> SELECT * FROM Article WHERE IdArticle = 10
  2  /

IDARTICLE DESCRIPTION          PRIXUNITAIRE QUANTITÉENSTOCK
--------- -------------------- ------------ ---------------
       10 Boule de cèdre              12,09              10
```

Trigger INSTEAD OF pour Vue non modifiable

```
SQL> SELECT * FROM Article WHERE IdArticle = 10
  2  /

 IDARTICLE DESCRIPTION          PRIXUNITAIRE QUANTITÉENSTOCK
---------- -------------------- ------------ ---------------
        10 Boule de cèdre              10,99              20

SQL> CREATE VIEW ArticlePrixPlusTaxe AS
  2   SELECT  IdArticle, description, prixUnitaire * 1.15 AS prixPlusTaxe
  3   FROM  Article
  4  /

View created.

SQL> UPDATE ArticlePrixPlusTaxe
  2  SET prixPlusTaxe = 23
  3  WHERE IdArticle = 10
  4  /
SET prixPlusTaxe = 23
    *
ERROR at line 2:
ORA-01733: virtual column not allowed here
```

```
SQL> CREATE OR REPLACE TRIGGER InsteadUpdate
  2  INSTEAD OF UPDATE ON ArticlePrixPlusTaxe
  3  REFERENCING
  4   OLD AS ligneAvant
  5   NEW AS ligneAprès
  6  FOR EACH ROW
  7  BEGIN
  8   UPDATE Article
  9   SET
 10         IdArticle = :ligneAprès.IdArticle,
 11         description = :ligneAprès.description,
 12         prixUnitaire = :ligneAprès.prixPlusTaxe / 1.15
 13   WHERE IdArticle = :ligneAvant.noArticle;
 14  END;
 15  /

Trigger created.

SQL> UPDATE ArticlePrixPlusTaxe
  2  SET prixPlusTaxe = 23
  3  WHERE IdArticle = 10
  4  /

1 row updated.

SQL> SELECT * FROM Article WHERE IdArticle = 10
  2  /

 IDARTICLE DESCRIPTION          PRIXUNITAIRE QUANTITÉENSTOCK
---------- -------------------- ------------ ---------------
        10 Boule de cèdre               20              20
```

Procédures stokées en Java

```
import java.sql.*;
import java.io.*;
public class RoutineServeur extends Object {

  public static int getQuantiteEnStock (int noArticle) throws SQLException {
      // Retourne -1 si l'article n'existe pas
      PreparedStatement unEnonceSQL = null;
      int quantiteEnStock = -1;
      try {
        Connection uneConnection =
          DriverManager.getConnection("jdbc:default:connection:");
        unEnonceSQL = uneConnection.prepareStatement
        ("SELECT quantiteEnStock FROM Article WHERE idArticle = ? ");
        unEnonceSQL.setInt(1,noArticle);
        ResultSet résultatSelect = unEnonceSQL.executeQuery();
        if (résultatSelect.next ()){
          quantiteEnStock = résultatSelect.getInt(1);
        }
      }
      catch (SQLException e) {System.err.println(e.getMessage());}
      finally{unEnonceSQL.close();}
      return quantiteEnStock;
  }

  public static void setQuantiteEnStock (int noArticle, int quantiteEnStock) throws
SQLException {
      PreparedStatement unEnonceSQL = null;
      try {
        Connection uneConnection =
          DriverManager.getConnection("jdbc:default:connection:");
        unEnonceSQL = uneConnection.prepareStatement
        ("UPDATE Article SET quantiteEnStock = ? WHERE idArticle = ? ");
        unEnonceSQL.setInt(1,quantiteEnStock);
        unEnonceSQL.setInt(2,noArticle);
        unEnonceSQL.executeUpdate();
      }
      catch (SQLException e) {System.err.println(e.getMessage());}
      finally{unEnonceSQL.close();}
  }
}
```

Déploiement Oracle

- Charger le code dans la base:

```
loadjava -user Scott/Tiger RoutineServeur.class
```

- Publier sous forme de routine stockée :

```
SQL> CREATE OR REPLACE FUNCTION getQuantiteEnStock(noArticle NUMBER)
  2  RETURN NUMBER
  3  AS LANGUAGE JAVA
  4  NAME ' RoutineServeur.getQuantiteEnStock (int) return int';
  5  /

Function created.
```

- Appeler la fonction en SQL:

```
SQL> select getQuantiteEnStock(10) from dual;

GETQUANTITEENSTOCK(10)
----------------------
                    20
```

- Possibilité de définir un déclencheur utilisant une procédure/fonction écrit en Java

Modèle client/serveur bases de données

Architecture des Application Web

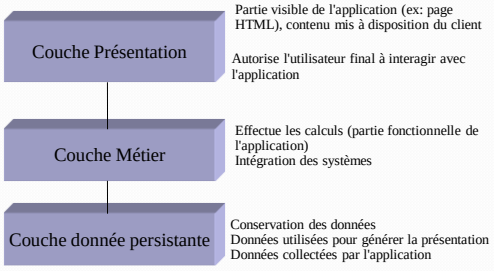
Niveau Logique (architecture trois tier)

- Couche Présentation
- Couche Métier/Application
- Couche Accès aux données

Niveau Physique

- Client
- Serveur Web
- Serveur d'application
- Serveur de données

Architecture trois tier



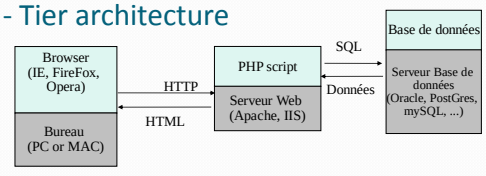
Partie visible de l'application (ex: page HTML), contenu mis à disposition du client

Autorise l'utilisateur final à interagir avec l'application

Effectue les calculs (partie fonctionnelle de l'application)
Intégration des systèmes

Conservation des données
Données utilisées pour générer la présentation
Données collectées par l'application

3 - Tier architecture



Couche Présentation

Couche Présentation *Couche Application* *Couche données*

Exemple PHP

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <h1>Region list</h1>
    <?php
     $link = pg_Connect("host=localhost user=login dbname=login");
     $result = pg_query("SELECT * FROM regions", $link) or
               die("ERROR: Request failed");
     if (pg_numrows($result) == 0)
       print("<h2>Sorry, no region, db is empty.</h2><br>");
     else
       while ($row = pg_fetch_array($result))
       {
         print("<a href=\"BrowseCategories.php?region=".
               $row["id"]."\">".$row["name"]."</a><br>\n"); }
     pg_free_result($result);
     pg_close($link);
    ?>
  </body>
</html>
```

Transactions

Concurrence des accès

- un SGBD est multi-utilisateurs
- une même donnée peut être manipulée par plusieurs utilisateurs à la fois

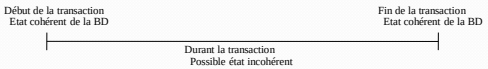
⇒ problèmes d'accès concurrents (perte de mise à jour, lecture incohérente, ...)

⇒ comment garantir la cohérence des données ?

⇒ contraintes de temps réel

Transaction

- Une transaction est une séquence d'opérations qui accèdent ou modifient le contenu d'une base de données .
  - Granularité des opérations de lecture/modification de la base: table, n-uplet, donnée, ...
- Une transaction permet de passer d'un état cohérent de la base à un autre état cohérent



Transaction

- Une transaction peut être validée ou annulée
- Le début d'une transaction peut être implicite ou explicite
- La fin d'une transaction peut être implicite ou explicite
- Contraintes sur les transactions:
  - Vitesse d'exécution
  - Sécurité

Transaction

Commandes de transaction

- Début de transaction implicite
  - Début d'une session
- Un début de transaction explicite est définie par :
  - `COMMIT` (fin de la transaction précédente)
  - `BEGIN_TRANSACTION` (SQL standard, pas Oracle)
- Une fin explicite est définie par :
  - `ROLLBACK` (annulation)
  - `COMMIT` (validation)
- Une fin implicite est définie par :
  - Exécution d'une commande du langage de définition de données
  - Fin d'une session

Exemple de transaction

```
-- livraison de 10 exemplaires du produit 113
-- les stocks sont mis à jour

COMMIT;

INSERT INTO
  livraison (idProduit, quantité, idClient)
  VALUES    (113, 12, 1125);

UPDATE produit SET quantité=quantité - 12
WHERE idProduit = 113;

COMMIT COMMENT 'Virement effectué'; -- Ou ROLLBACK
```


Transaction : propriétés ACID

- **Atomicité :** Une transaction est indivisible. Elle est soit complètement exécutée soit pas du tout (unité atomique de traitement)
- **Cohérence :** une transaction doit effectuer une transition d'un état cohérent de la base à un autre état cohérent (par ex: pas de violation de contrainte d'intégrité). La cohérence peut être non respectée pendant l'exécution d'une transaction. En cas d'échec de la transaction, la base doit retourner dans l'état cohérent initial

Transaction : propriétés ACID

- **Isolation :** le résultat d'un ensemble de transactions concurrentes et validées correspond à une exécution successive des mêmes transactions (pas d'inférence entre les transactions)
- **Durabilité :** après la fin d'une transaction, les mises à jour sont définitives même en cas de problèmes matériels (mécanisme de reprise en cas de panne)

Etats des données avant COMMIT ou ROLLBACK

Comportement standard:

- L'état précédent de la base peut être récupéré,
- L'utilisateur courant peut voir le résultat de ses mises à jour de données,
- Les autres utilisateurs ne peuvent généralement pas voir les mises à jour effectuées par l'utilisateur courant,
- Certaines données peuvent être verrouillées,
- Les autres utilisateurs ne peuvent modifier les données verrouillées.

Etat après le COMMIT

- Les modifications de la base sont permanentes,
- L'etat précédent de la base est définitivement perdu,
- Tous les utilisateurs peuvent voir le résultat des opérations de DML,
- Les verrous sont relachés,
- Tous les points de sauvegarde sont effacés.

Transaction

Validation de transaction partielle

- On peut découper une transaction en créant des points d'arrêts (savepoint).
  - `SAVEPOINT S1;`
- Ces points d'arrêts permettent d'annuler seulement une partie de la transaction. La commande `ROLLBACK` permet de retourner à un point d'arrêts:
  - `ROLLBACK TO S1;`

Contrôle de la concurrence

Objectif: synchroniser les transactions concurrentes de façon à maintenir la cohérence de la BD tout en minimisant les restrictions d'accès.

Principes:

- Exécution simultanée des transactions
  - Opération 1 de la transaction T1, opération 1 de la transaction T2, opération 2 de la transaction T1, ...

  ⇒ permet d'améliorer la performance du SGBD
- L'exécution simultanée de transactions concurrentes doit être équivalent à une exécution non simultanée (propriété Isolation)

Contrôle de la concurrence

Opérations élémentaires d'accès à la base de données

Lire(X)

Lit la granule X, qui sera généralement stockée dans une variable (par convention la variable sera aussi nommée X)

Ecrire (X)

Ecrit la valeur de la variable X dans la granule X

Exemple de transactions concurrentes

Transaction T1	Transaction T2
`Lire(X);`	`Lire(X);`
`X = X - 500;`	`X = X + 1000;`
`Ecrire(X);`	`Ecrire(X);`
`Lire(Y);`	
`Y = Y + 500;`	
`Ecrire(Y);`	
Accède aux granules X et Y	Accède à la granule X

Exemple d'exécution simultanée

```
X = 2000
Y = 0
```

Transaction T1	Transaction T2
`Lire(X);`	
`X = X - 500;`	
`Ecrire(X);`	
	`Lire(X);`
	`X = X + 1000;`
	`Ecrire(X);`
`Lire(Y);`	
`Y = Y + 500;`	
`Ecrire(Y);`	

```
X = 2500
Y = 500
```


Contrôle de concurrence

Nécessité de contrôler les transaction pour éviter les incohérence: ici perte de mise à jour

```
X = 2000
Y = 0
```

Transaction T1	Transaction T2
`Lire(X);`	
`X = X - 500;`	
	`Lire(X);`
	`X = X + 1000;`
`Ecrire(X);`	
`Lire(Y);`	
	`Ecrire(X);`
`Y = Y + 500;`	
`Ecrire(Y);`	

La mise à jour de la granule X par la transaction T1 est perdue

```
X = 3000
Y = 500
```


Introduction d'incohérence

Contrainte: X = Y

```
X = 100
Y = 100
X = Y
```

Transaction T1	Transaction T2
`Lire(X);`	
`X = 2 * X;`	
`Ecrire(X);`	
	`Lire(X);`
	`X = X + 1;`
	`Ecrire(X);`
	`Lire(Y);`
	`Y = Y + 1;`
	`Ecrire(Y);`
`Lire(Y);`	
`Y = 2 * Y;`	
`Ecrire(Y);`	

```
X = 201
Y = 202
X != Y
```

Mise à jour temporaire (dirty read)

Transaction T1	Transaction T2
`Lire(X);`	
`X = X + 500;`	
`Ecrire(X);`	
	`Lire(X);`
	`X = X + 1000;`
	`Ecrire(X);`
`ROLLBACK;`	

T2 a lu une valeur temporaire de X et s'en est servi pour effectuer une mise à jour

Lecture non répétable

Transaction T1	Transaction T2
	`S = 0;`
	`Lire(X);`
	`S = S + X;`
`Lire(X);`	
`X = X - 500;`	
`Ecrire(Y);`	
`COMMIT;`	
	`Lire(X);`
	`S = S + X;`

La valeur de X lu par T2 à la fin de la transaction n'est pas la même que celle lu au début (elle peut même avoir été supprimée)

Lecture fantôme

Transaction T1	Transaction T2
	`S = 0;`
	`Lire(X); // Select max(X) from P1;`
`INSERT INTO P1 VALUES (500000);`	
	`S = S + X;`

Le contenu de la table P1 a été modifié. La valeur précédemment lue n'est plus le maximum de la valeur X de la table P1

Degrés d'isolation SQL-92

On peut préciser si l'on souhaite éviter ou non les cas suivants:

- Lecture sale : Une transaction lit des données écrites par une transaction concurrente mais non validée
- Lecture non reproductible : Une transaction lit de nouveau des données qu'elle a lues précédemment et trouve que les données ont été modifiées par une autre transaction (qui a validé depuis la lecture initiale)
- Lecture fantôme : Une transaction ré-exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé à cause d'une transaction récemment validée

Exécution

Exécution: ordonnancement des opérations d'un ensemble de transaction

Exemple :

3 transactions T_1, T_2, et T_3

3 granules X, Y et Z

T_1 : Lire(X), Ecrire(X)

T_2 : *Ecrire(X), Ecrire(Y), Ecrire(Z)*

T_3 : **Lire(X), Lire(Y), Lire(Z)**

Exemple d'exécution:

$K = E_2(X), L_1(X), \mathbf{L_3(X)}, E_1(X), E_2(Y), \mathbf{L_3(Y)}, E_2(Z), \mathbf{L_3(Z)}$

Exécution en série

Exécution sérielle: exécution sans entrelacement d'opérations de différentes transactions. Equivalent à une exécution successive par un unique utilisateur des différentes transactions.

Exemple:

$K = L_1(X), E_1(X), E_2(X), E_2(Y), E_2(Z), L_3(X), L_3(Y), L_3(Z)$

T_1 T_2 T_3

Exécutions correctement synchronisées

- Deux exécutions sont équivalentes si elles ont les mêmes transactions, l'ordre des opérations des transactions est le même et qu'elles produisent les mêmes effets sur la base
- Sérialisabilité
  Les exécutions concurrentes sont correctes si et seulement si leur résultat est équivalent à celui d'une exécution sérielle (non simultanée)
- Critères couramment accepté (cas particulier les bases de données distribués)

Exécutions équivalentes

T_1 : Lire(X), Ecrire(X)
T_2 : Ecrire(X), Ecrire(Y), Ecrire(Z)
T_3 : Lire(X), Lire(Y), Lire(Z)

$K_1 = E_2(X), \mathbf{L_1(X)}, \mathbf{L_3(X)}, E_1(X), E_2(Y), L_3(Y), E_2(Z), L_3(Z)$
$K_2 = E_2(X), \mathbf{L_3(X)}, \mathbf{L_1(X)}, E_1(X), \mathbf{E_2(Y)}, \mathbf{L_3(Y)}, E_2(Z), L_3(Z)$
$K_3 = E_2(X), L_1(X), L_3(X), E_1(X), \mathbf{L_3(Y)}, \mathbf{E_2(Y)}, E_2(Z), L_3(Z)$

K_1 et K_2 sont 2 exécutions équivalentes.
K_2 et K_3 ne sont pas équivalentes.

$K_s = E_2(X), E_2(Y), E_2(Z), L_3(X), L_3(Y), L_3(Z), L_1(X), E_1(X)$
$= T_2, T_3, T_1$

K_1 et K_2 sont équivalent à K_s qui est une exécution en série. K_1 et K_2 sont sérialisable.

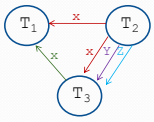
Graphe de précédence

Le graphe (orienté) de précédence $G_k = \{S,A\}$ pour une exécution K est défini tel que :

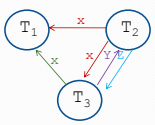
- S = ensemble des transactions de l'exécution K
- Il existe un arc entre (T_i,T_j) si
  - T_i lit/écrit la granule X avant une écriture dans X de T_j
  - T_i écrit une granule X avant une lecture de X de T_j
  - On dit que T_i précéde T_j

Graphe de précédence

K2 = E2(X), L3(X), L1(X), E1(X), E2(Y), L3(Y), E2(Z), L3(Z)
K3 = E2(X), L1(X), L3(X), E1(X), L3(Y), E2(Y), E2(Z), L3(Z)



Graphe de précédence de K_2



Graphe de précédence de K_3

On peut prouver qu'une exécution est sérialisable si et seulement si le graphe de précédence ne contient aucun cycle.
K_3 contient un cycle, donc l'exécution n'est pas sérialisable.

Degrés d'isolation

- Sérialisabilité totale coûte cher
- Elle n'est pas nécessaires pour toutes transactions
- Les SGBD et SQL-92 offrent dès lors différents niveaux d'isolation de transactions
  - à utiliser avec des précautions

Degrés d'isolation SQL-92

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y
Serializable	N	N	N

Oracle isolation levels

Read committed	Chaque requête ne voit que les données validées (isolation level d'Oracle par défaut)
Serializable	Voit les données telles qu'elles étaient au début de la transaction. Une transaction voit aussi ses propres modifications.
Read-only	La transaction voit les données telles qu'elles étaient au début de la transaction. Toutes commandes DML est interdites.

Oracle isolation levels

Commande SQL (Oracle):

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SET TRANSACTION READ ONLY;
```

Contrôle de concurrence

Le contrôle de concurrence permet de s'assurer que seul des exécutions sérialisables soient générées.

2 types de techniques existent pour garantir la sérialisation des opérations:

- Techniques optimistes: détection des conflits et annulation de leur effet si possible (permutation d'opérations). Sinon annulation de transactions.
  Valable uniquement si les conflits sont rares. Il est alors acceptable de réexécuter occasionnellement certaines transactions.

- Techniques pessimiste: empêcher l'apparition des conflits de mise à jour en verrouillant les objets de la base.
  On suppose dans ce cas que les conflits sont fréquents et qu'il faut les traiter le plus rapidement possible.

Contrôle de concurrence

Les techniques de contrôle de concurrence pessimistes sont les plus souvent utilisées.

Elles consistent à poser des verrous sur les objets de la base.

Verrou

Un verrou est une variable d'état associée à un objet X de la base et indiquant son état vis à vis des opérations de lecture/écriture

- Verrou binaire:

  2 états :
  - verrouillé (pose de verrou): l'accès à un objet est bloqué dès qu'il est lu ou écrit par une transaction
  - libre: le verrou sur un objet est libéré à la fin de la transaction

- Verrou ternaire :

  3 états :
  - verrou partagé ("shared lock"): plusieurs transactions peuvent lire l'objet
  - verrou exclusif ("exclusive lock"): réserve l'objet en écriture pour la transaction qui pose ce verrou
  - libre.

- Un verrou peut toujours être posé sur une donnée non verrouillée, la donnée est alors verrouillée
- Plusieurs verrous en lecture peuvent être posés sur une donnée déjà verrouillée en lecture
- Si une donnée est verrouillée en écriture, aucun autre verrou ne peut être posé tant que ce verrou n'a pas été supprimé

- Deux verrous sont dits compatibles si deux transactions qui accèdent à la même donnée peuvent obtenir les verrous sur cette donnée au même moment.

Verrouillage de ressources

Un verrou peut intervenir à différents niveau de la base:

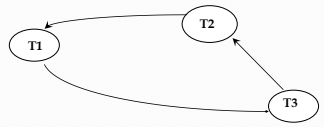
- sur la base données,
- sur les tables,
- sur les n-uplets,
- sur les données.

Verrouillage à deux phases

- Chaque transaction verrouille l'objet avant de l'utiliser
- Quand une demande de verrou est en conflit avec un verrou posé par une autre transaction en cours, la transaction qui demande doit attendre
- Quand une transaction libère son premier verrou, elle ne peut plus demander d'autres verrous
- Toute exécution obtenue par un verrouillage à deux phases est sérialisable

Problème avec le verrouillage: interblocage

Verrou mortel



Résolution du verrou mortel

- Prévention
  - définir des critères de priorité de sorte à ce que le problème ne se pose pas
  - exemple : priorité aux transactions les plus anciennes
- Détection
  - gérer le graphe des attentes
  - lancer un algorithme de détection de circuits dès qu'une transaction attend trop longtemps
  - choisir une victime qui brise le circuit

Améliorations du verrouillage

- Relâchement des verrous en lecture après opération
  - non garantie de la reproductibilité des lectures
  + verrous conservés moins longtemps
- Accès à la version précédente lors d'une lecture bloquante
  - nécessité de conserver une version (journaux)
  + une lecture n'est jamais bloquante

Oracle durée des verrous

- Tous les verrous acquis durant une transaction reste actif tous le temp de la transaction.
- Les verrous sont relachés lors d'un commit ou rollback

Oracle DML Locks

Les opérations de manipulation de données peuvent acquerir un verrou à deux niveaux: ligne ou table.

Table Locks

- RS: row share

Les autres transactions peuvent modifier la table mais pas la verrouiller en mode exclusif

- RX: row exclusive

Comme row share mais la table ne peut pas être verrouiller en share

- S: share

Accès concurrents autorisés sur la table mais pas les modifications

- SRX: share row exclusive

Les accès sont autorisés sur la table mais pas les modifications ou de la verrouiller en share

- X: exclusive

Seuls les accès à la table sont autorisés

* Attends si une autre transaction a déjà posé le verrou

SQL Statement	Mode of Table Lock	Lock Modes Permitted? RS	RX	S	SRX	X
SELECT...FROM table...	none	Y	Y	Y	Y	Y
INSERT INTO table ...	RX	Y	Y	N	N	N
UPDATE table ...	RX	Y*	Y*	N	N	N
DELETE FROM table ...	RX	Y*	Y*	N	N	N
SELECT ... FROM table FOR UPDATE OF ...	RS	Y*	Y*	Y*	Y*	N
LOCK TABLE table IN ROW SHARE MODE	RS	Y	Y	Y	Y	N
LOCK TABLE table IN ROW EXCLUSIVE MODE	RX	Y	Y	N	N	N
LOCK TABLE table IN SHARE MODE	S	Y	N	Y	N	N
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE	SRX	Y	N	N	N	N
LOCK TABLE table IN EXCLUSIVE MODE	X	N	N	N	N	N

Ordonnancement par estampillage

On affecte

- à chaque transaction t une estampille unique TS(t) dans un domaine ordonné.
- à chaque granule g
  - une étiquette de lecture EL(g)
  - une étiquette d'écriture EE(g)

qui contient l'estampille de la dernière transaction la plus jeune (max(TS(t))) qui a lu (respectivement) écrit g.

Ordonnancement par estampillage

La transaction t veut lire g :

- Si TS(t) ≥EE(g),la lecture est acceptée et
  EL(g) := max(EL(g), TS(t))
- Sinon la lecture est refusée et T est relancée avec une nouvelle estampille (plus grande que tous les autres)

La transaction t veut écrire g :

- Si TS(t) ≥max(EE(g), EL(g)),l'écriture est acceptée et
  EE(g) := TS(t)
- Sinon l'écriture est refusée et T est relancée avec une nouvelle estampille (plus grande que tous les autres).

Bilan Estampillage

- Approche optimiste
  - coût assez faible
  - détecte et guérit les problèmes
- Guérison difficile
  - catastrophique en cas de nombreux conflits
  - gère mal les pics d'accès