Notes Programmation Serveur et PL/SQL :

Utilisation des stored procedures selon l'environnement.

Création de procedure

```
CREATE Procedure nomProc(...) IS
    /* Déclaration de variable */
    BEGIN
    /* Code de la procédure */
    EXCEPTION /* WHEN nom_exception */
    /* Code de l'exception lancée */
    END;
```

Création de fonction

```
CREATE Function nomFunction(...) RETURN typeRetour
     [Deterministic] IS
    /* Déclaration de variable */
    BEGIN
    /* Code de la fonction */
    EXCEPTION /* WHEN nom_exception */
    /* Code de l'exception lancée */
    END;
```

Deterministic -> Délivre les mêmes résultats à chaque fois pour les meme paramètres. Si une fonction est deterministic mais pas précisé, on ne peut pas l'utiliser pour un index.

```
CREATE OF REPLACE FUNCTION ToComparableString(chaine VARCHAR)
    RETURN VARCHAR DETERMINISTIC IS

    chaineInt VARCHAR(2000);

    BEGIN
        chaineInt := lower(chaine);
        return translate(chaineInt, 'éèêëçñ- ''', 'eeeecn');
    END;

SELECT * FROM Ancien
WHERE ToComparableString(ancnom) = ToComparableString(?) -- ? == Pas vu ce que le prof a no
ORDER BY ToComparableString(ancnom);
```

```
CREATE INDEX NomComparableNdx ON ANCIEN(ToComparableString(ancnom))
```

```
select wm_concat(empnom)
    from employe
    group by empdp
```

3ème génération : les BD's actives :

Déclencheur / Triggers

SGBD Actif : SGBD qui effectue des actions lors d'un évènement

Exemple d'évènement : * Connexion de quelqu'un. * Le fait que quelqu'un appele une procédure particulière.

Déclencheur est un triplet ECA Evènement - Condition - Action

On se content de trois évènement qui sont **INSERT**, **UPDATE**, **DELETE**.

```
CREATE OR REPLACE TRIGGER nomTrigger
    {BEFORE | AFTER}
    DELETE | INSERT | UPDATE [of column [,column] ...]
    [OR DELETE | INSERT | UPDATE [of column [,column] ...]]
    [OR DELETE | INSERT | UPDATE [of column [,column] ...]]

    ON nomTable

    [REFERENCING OLD [AS] old NEW [AS] new]
    [FOR EACH ROW]
    [WHEN (condition) ] -- pas souvent utilisé
    bloc_PL/SQL
```

REFERENCING doit aller avec FOR EACH ROW Si pas de for each row, ceci n'a pas de sens car on ira pas voir la valeur d'un tuple **old** et **new** sont des termes que l'on créé.

Si cet évènement là apparait, on va éxécuter un traitement, avant ou après l'éxécution du insert, update, delete.

Exemples :

```
CREATE TABLE TestTrg(
    id int primary key,
    nom varchar(100) not null);
```

Empecher de modifier la clé primaire :

```
CREATE TRIGGER PkTestTrgStable
    BEFORE
    UPDATE of id
    ON TestTrg
        BEGIN
            -- Lancement d'une exception
            RAISE_APPLICATION_ERROR(-20100, 'La PK ne peut pas être
            modifié');

        END;
```

Sequence de clé primaires

```
    CREATE SEQUENCE SeqPourTrig
        START WITH 2
        INCREMENT BY 1;


    SELECT SeqPourTrig.nextVal FROM dual

    CREATE TRIGGER TestTrgAutoincrement
        BEFORE
        INSERT
        ON TestTrg
        REFERENCING NEW as new
        FOR EACH ROW -- Il faut une clé différente à chaque tuple
        BEGIN
            :new.id := SeqPourTrg.nextVal;
        END;

    ALTER TABLE TestTrg
        ADD nomComparable VARCHAR(100);

    UPDATE TestTrg
        SET nomComparable = TocomparableString(nom);
    COMMIT;

    CREATE TRIGGER gereNomComparable
        BEFORE
        UPDATE OF nom OR INSERT
        ON TestTrg
        REFERENCING NEW as new
        FOR EACH ROW
        BEGIN
```

```
            :new.nomComparable := toComparableString(:new.nom);
        END;
```

Pas de langage pour l'ensemble des procédures stockées. Différent pour chaque environnement de développement.

Exemples :

CI de empDpt : un salaire ne peut pas diminuer.

```
CREATE TRIGGER SalNeDiminuePas
    BEFORE
    UPDATE of empSal
    ON Employe
    REFERENCING OLD as old NEW as new
    FOR EACH ROW
    BEGIN
        if (:new.empSal < :old.empSal) THEN
            RAISE_APPLICATION_ERROR(-20100,
            'Un salaire ne peut pas diminuer');
        end if;
    END;

Test :  UPDATE Employe set empSal = empSal + 1
            WHERE EmpNo = '050'

            -- OK

        UPDATE Employe set empSal = empSal - 1
            WHERE empNo = '050'

            -- ERREUR
```

Supprimer un trigger : DROP TRIGGER nomTrigger

Performances : Ajouter une colonnes dans Departement pour connaître le nombre d'employe du departement.

Employe : Insert, delete , et update empdpt

```
Alter table departement
    add dptNbEmps int default 0

update departement set dptnbemps = (select count(\*) from
    employe where empdpt = dptno)
```

```
create trigger gerenbemployes
    after
    insert or delete or update of empdpt
    on employe
    referencing OLD as old NEW as new
    for each row
    BEGIN
        IF (INSERTING) THEN
            UPDATE DEPARTEMENT
                SET dptNbEmps = dptnbEmps + 1
                WHERE dptNo = :new.empDpt;
        END IF;

        IF (DELETING OR UPDATING) THEN
            UPDATE DEPARTEMENT
                SET dptNbEmps = dptnbEmps - 1
                WHERE dptNo = :old.empDpt;
        END IF;

    END;
```

Attention : Oracle refusera d'interroger une table en cours de mutation

Exemple de trigger sans for each row

Sécurité

Physique

Acheter du matériel de qualité -> Captain Obvious! Eviter les destructions : Ne pas mettre n'importe ou le matériel -> Captain Obvious[2] Ne pas mettre son serveur aux chiottes.

Accès

Seuls ont accès aux données, les personnes autorisées.

Logique

Ensemble de choses à mettre en oeuvre pour garder les données cohérentes.

Mettre en oeuvre des choses qui vont limiter les conséquences en cas de problème.

Disposition pour minimiser les risques de survenances de problèmes.

Conséquences

- Back Up / Supports externes
- Quand faire un baique heup ?
- Journaling : A chaque modification appliquée, on stocke la modification dans un journal.
- Le jour où il y a un problème : appliquer le journal.
- PCA : Plan de ? d'activités
- PRA : Plan de reprise d'activités

Sécurité d'accès

User - Rôle

-> Tables, View, Fonctions, Procédures.

On donne plutôt des droits sur des vues que des tables.

Pour utiliser une application, l'utilisateur doit s'identifier. L'application a besoin de se connecter à la BD. Le password est-il le même que celui de l'application

Le password est sur un serveur d'identification. Même l'user n'y a pas accès.

Sécurité logique

Pertes d'opérations : Probleme de concurrence d'accès. (même chose que cours systeme). -> Deux transactions qui travaillent en parallèles, une ressource est modifiée par un autre programme.

Introduction d'incohérence :

x = y

T1

```
read x, a
a <- a + 1
write a,x

read y, a
a <- a + 1
write y, a
```

T2 (en parallèle avec T1)

```
read y,a
a <- a * 2
write y, a

read x, a
a <- a * 2
write x,a
```

Non reproductibilité des lectures

Contrôleur de concurrence :

Programme qui va contrôler et gérer les conflits (ex. processus qui accèdent aux mêmes données.

Le contrôleur va diviser les données en un ensemble de granules. (granules de concurrences).

Granules de concurences : unité de données que le controleur vérifie.

Quand on accède à une unité de donnée, le contrôleur fait en sorte que cette unité ne puisse plus être accédée.

Le plus haut : La BD, le plus bas : le tuple.

- BD -> Une seule personne connecté à la BD à un même moment.
- Table
- Page mémoire
- Tuples

PLus le granule est petit, meilleure sera la fluidité de transaction (Moins de conflits) (Augmentation du débit des transactions). Mais plus il est petit, plus le contrôleur a du travail, donc moins de performances.

Techniques

Technique pessimiste / optimiste

Technique pessimiste Le contrôleur empeche l'accès directement. Il empeche que des conflits surgissent.

Technique optimiste Le contrôleur laisse travailler et vérifie au commit. Il y a donc plus de travail de vérification et de gestion de versions.

Chaque SGBD fait un mélange des deux techniques à sa manière.

Verrous La technique pessimiste nécessite la notion de verrou. Un verrou est soit court, soit long et soit partagé, soit exclusif.

Verrou court Le verrou est posé au début de l'action et retiré à la fin de l'action. (DML)

Verrou long Le verrou est posé au début de l'action et retiré à la fin de la transaction. Il faut donc avoir des transactions les plus courtes possibles.

Verrou partagé File d'attente de ressources?

Verrou exclusif Ne permet pas qu'un autre verrou soit poser sur un granule. Si un verrou exclusif est déjà posé, on ne peut pas posé d'autre verrou.

Problèmes de Deadlock

Les verrous peuvent produire des deadlock. Soit deux transactions : T1 et T2. T1 accède à A et pose un verrou long exclusif. T2 accède à B et pose un verrou long exclusif. T1 demande d'accéder à la ressource B (bloqué). T2 demande d'accéder à la ressource A (bloqué). -> Deadlock.

Les SGBDs ne réagissent pas de la même manière aux deadlocks.

Soit il peut y avoir un time-out, soit le controleur de concurrence peut gérer cette problématique en envoyant un signal pour 'inviter le process à se suicider'

Dirty Read (lecture sale) :

On pourrait travailler en lisant le 'brouillon' non commité de quelqu'un d'autre.

On voudrait être protégé du dirty read. Phantom Read : Entre deux read, quelqu'un a ajouté un tuple.

L'isolation

Plus le degré d'isolation est élevé, plus on est protégé des effets pervers des autres. Mais moins de performances. En technique pessimiste, on peut gérer les degrés 1 et 2 en utilisant les 4 types de verrous.

- 0 : Dirty Read : Aucune isolation, seule les actions sont isolées.
- 1 : Committed Read (par défaut sur oracle) : Avec JavaDB, on est bloqué tant que le premier ne fait pas de commit car JavaDB travaille essentiellement en pessimiste. A ce niveau là, on s'assure de ne pas avoir de Dirty Read.
- 2 : Repeatable Read (a partir du moment où on a aquit une donnée, on récupère les mêmes données) - Permet le phantom read. Nous assure de la reproductibilité des lectures. Ne permet pas de voir deux valeurs différentes pour la même ressource.
- 3 : Serializable : Comme si les transactions s'effectuait les une derrières les autres. (Pas de phantom read) - le mieux mais le plus cher. Ce niveau ne peut pas être géré en pessimiste.

La norme demande que le degré d'isolation par défaut soit le 3. Mais les SGBDs mettent souvent le degré 1 par défaut.

Le degré 3 ne protège pas de tout ! La probabilité qu'une erreur se produise est infime, mais existe.

```
SELECT ...
...
FOR UPDATE [of liste attributs]
```

Exemple d'implémentation en full pessimiste (JavaDB)

- Dirty Read : Faut-il poser des verrous ? Oui.
  - Verrous en écriture : dépot de verrou long exclusif
- Committed Read : Ce qu'offre le Dirty Read + :
  - Verrous en lecture : dépot de verrou court partagé
- Repeatable Read : Ce qu'offre le Committed Read + :
  - Verrous en lecture : dépot de verrou long patagé (On empeche pas les autres de lire).
- Serializable : Impossible en purement pessimiste. Il faut déposer d'autres types de verrous (verrou prédicatif).

Conception des applications et gérer la sécurité

Un peu de tout Normes :

- 1986 : SQL-87
- 1989 : SQL-1
- 1992 : SQL-2
- 1999 : SQL-3 (Récursivité, notion de déclencheur, notion d'objet)
- 2003 : Séquence et attributs de type xml
- 2006 : Plus de noms spécifiques
- 2008 : Plus de noms spécifiques

Branchements dans un select :

```
CASE expression:
    WHEN valeur THEN
            expression
    [...]
```

```
SELECT empno, empnom, CASE empsexe
                        WHEN 'F' THEN 'Féminin'
                        WHEN 'M' THEN 'Masculin'
                        ELSE 'Oufti...'
                        END,
                      empsal
                FROM Employe
```

```
SELECT empdpt, count(empno)
    from employe
    group by empdpt
    having count(empno) >= ALL (select count(empno) from employe group by empdpt)
```

```
WITH V1 (e, nb) AS
    (select empdpt, count(*) from employe group by empdpt)

SELECT e, nb, FROM Q1
    WHERE nb = (select max(nb) FROM V1);
```

Select hierarchique (récursion) Retrouver la liste des enfants des département.

```
WITH V2 (dno, dlib, dpere) AS
    (SELECT dptNo, dptLib, dptAdm
        FROM DEPARTEMENT
        WHERE dptAdm is null
    UNION ALL
    SELECT (dptNo, dptLib, dptAdm)
        FROM DEPARTEMENT
        JOIN V2 v ON v.dno = dptAdm)

SELECT * FROM V2
```

Transactions imbriquées :

```
    SAVEPOINT nomSave1

    SAVEPOINT nomSave2

    ROLLBACK TO nomSave1
```