Persistance des données

Placentino

May 28, 2015

Contents

Correction Examen janvier :

Ordres SQL :

```
SELECT DISTINCT, tId, tLibelle
    FROM Talent JOIN
        Possession p ON tId = talent
        JOIN Appartenance a ON a.artiste = p.artiste
    WHERE a.groupe = 214

SELECT DISTINCT debut, fin
    FROM Representation JOIN
        Spectacle ON spectacle = sId
    WHERE artiste = 564 OR
            groupe IN(SELECT group
                        FROM Appartenance
                        WHERE artiste = 564)

SELECT sId, sNom
    FROM Spectacle JOIN
        Representation ON spectacle = sId
    GROUP BY sId, sNom
    HAVING COUNT(Distinct lieu) > 1

SELECT      gId, gNom
    FROM Groupe
    WHERE gId NOT IN
    (SELECT groupe FROM Spectacle
        JOIN Representation ON spectacle = sId
        JOIN Lieu ON lieu = lId
            WHERE lNom = '   ')
```

S.Q.L. :

DML :

- **atomicité** : tout se fait ou rien ne se fait
- **intégrité en lecture** : agit comme si les données n'étaient accèdées par personne d'autre.

DCL :

- `COMMIT` validation de transaction

- `ROLLBACK` annuler la transaction

DDL :

- CREATE

```
CREATE TABLE nomTable (
        attrib, type-lg [[NOT] NULL]
        [DEFAULT val]
        [[CONSTRAINT nomCst]libelle]
)
```

- Types de CONSTRAINT :
  - `PRIMARY KEY`,
  - `UNIQUE`,
  - `FOREIGN KEY`,
  - `CHECK(cond)`
- Si on met pas NOT NULL => ça peut être NULL
- `PRIMARY KEY` si avec un attribut
- `PRIMARY KEY` {`attr, attr2, ..`} si en fin d'attributs
- `UNIQUE` si avec un attribut
- `UNIQUE` {`attr, attr2, ..`} si en fin d'attributs
- `CHECK(..)` si avec un attribut QUE CET ATTRIBUT LA !
- `CHECK(attrib, attr2, ..)` si ne fin d'attributs
- `FOREIGN KEY REFERENCES nomTableCible[attrib [,attrib2]]`
- `FOREIGN KEY (liste attributs) REFERENCES nomCible[attrib[, attrib]] [ON DELETE {RESTRICT, SET NULL, CASCADE}]`
- Une transaction accepte des données incohérentes mais évalue action par action. Si une donnée est incohérente, il va la vérifier en fin de transaction (avant le COMMIT)
- DEFFERABLE : attendre pour choisir de controler tout de suite ou en fin `INITIAL {..  }` = si jamais je dis rien au SGBD, il fait ça.

```
DEFERRABLE[INITIALLY {DEFERRED, IMMEDIATE}]
```

```
SET CONSTRAINTS {liste des noms de contraintes, ALL} {IMMEDIATE,
        DEFFERED}

        ///!\\ doit être la première instruction de la transaction
```

ex : SET CONSTRAINTS ALL DEFFERED ne prend en compte QUE les contrainte définies DEFERRABLE

Modification :

- ALTER TABLE

```
ALTER TABLE nomTable
    [ADD nomColonne TYPE [[NOT]NULL][DEFAULT]]
    [DROP nomColonne]
    [ADD constraint nomContrainte DECLARATION CONTRAINTE]
    [DROP constraint nomContrainte]
    [MODIFY nomColonne [[NOT]NULL]]  // Plus de possibilités selon SGBD.
```

/! Si on l'ajoute, cette contrainte doit être respectée sinon SGBD pas content.

- DROP TABLE

```
DROP TABLE nomTable
```

Peut être refusé si la table est la cible d'une clef étrangère car on pénalise une autre table à l'exterieur. Il faut donc d'abord supprimer la clef étrangere.

La sémantique :

- COMMENT ON TABLE

```
COMMENT ON TABLE nomTable IS 'Table exemplification DDL'‘
```

- COMMENT ON COLUMN

```
COMMENT ON COLUMN nomTable.nomColonne IS 'blablabla'‘
```

Le catalogue

Nom de la metabase.

- `SYSTABLES` est une table systeme reprenant toutes les tables de la BD.
  - nom de la table,
  - propriétaire de la table,
  - type de table, info.
  - stockage,
  - ...
- `SYSCOLUMNS`
  - nom colonne,
  - ref. table,
  - type colonne,
  - longueur,
  - le fait que ça puise être null,
  - ...
- `SYSINDEXES`
- `SYSCONSTRAINTS`
- ...
- Pour oracle :
  - `USER_TABLES`
  - `USER_COLUMNS`
  - ...

Schémas externes

CREAT VIEW

```
CREAT VIEW nomVue [(liste de noms de colonnes)]

    AS

SELECT ...
    [WITH CHECK OPTION]
```

Contrainte sur les opérations à réaliser et veillera que si on modifie ligne, cette ligne là satisfait encore la condition du select de la VUE

Ceci est une instruction qui créé un **schéma externe.** Une vue va apparaitre comme une table, `l'utilisateur n'en sait rien.`

`Exemple :`

```
CREATE VIEW Manager(mgrNo, mgrNom, mgrSexe)
 AS
SELECT DISTINCT empNo, empNom, enmpSexe
FROM Employe
    JOIN Departement ON empNo = dptMgr;
```

Certaines vues ne peuvent être modifiable (exemple : si on tente de modifier `INITIAL` {.. } = si jamais je dis rien au SGBD, il fait ça. Une "aggration function", rajouter 5euros a la masse salariale calculée par `sum(empsal)`

Schémas internes

`Cluster (regroupement) : Possibilité de relier, stocker ensemble plusieurs tables dans le même fichier physique.`

`Les factures sont liées au ligne_factures et donc la jointure des deux est très fréquentes. Pour diminuer le cout de ces jointures, on va disposer ces deux tables dans un cluster basé sur une clé.`

```
Facture
 Ligne
 Ligne
Facture
 Ligne
 Ligne
 Ligne
 ...
```

=> Ce sont des manipulations purement techniques, qui n'auront AUCUN impact sur le schéma conceptuel, donc aucun impact sur les schémas externes, ..

Notion d'Index :

- On évitera de créer un index sur une petite table,
- on évitera aussi sur des attributs qui ont très peu de valeurs(ex: empSexe),

- on évite aussi sur des critères de recherches qui ne sont pas > ou <,
- on évitera de créer des indexes inutiles.

=> Les indexes POURRAIENT nous donner de bons résultats en lecture. Mais cela va détériorer les résultats en modifications (il faut ajouter tuple + mise à jour de l'index).

```
CREATE [UNIQUE] INDEX nomIndex
    ON nomTable(list attributs)
```

- UNIQUE va créer une contrainte qui apparait au niveau conceptuel, du monde réel, ce n'est pas du niveau technique.

```
[ON fct(attr)] // possible en Oracle QUE SI la fonction est
```

deterministe (f(x) : a)

```
DROP INDEX nomIndex;
```

- Nom d'objets :

Instance (esidb, ORCL, ..)

à l'interieur d'instance : on a des schémas. Ex :

```
schéma ADT = espace de jeu personnel.
schéma SYS, schéma g39631, ...
```

- Oracle : chaque user a un shéma
- POSTGRES : on créé schémas +utilisateurs puis on associe
- pour ADT.Employe => CREATE SYNONYM Emps FOR ADT.Employe ou sur nos propres objets.
- DROP SYNONYM synonyme

Pour tout lemonde => CREATE PUBLIC SYNONY emp FOR ADT.Employe

DROP PUBLIC SYNONYM synonyme

Retour au DCL :

- Privilèges (droits sur d'autres environnement)

- USER
- ROLE

Les droits ont tendance à s'ajouter. On ne sait pas filter de manière générale les droits. On recoit nos privilèges + ceux accordé au différents role.

On accord un privilège en utilisant :

```
GRANT {all, (liste de privilèges)}
ON nomObjet
TO {liste user-roles, public}

table : SELECT
        UPDATE
        DELETE
        INSERT
```

Ce sont bien 4 privilèges SEPARES.

- REFERENCES : Je peux donner, sur ma table, a quelqu'un le droit de pouvoir faire une foreign key en specifiant tel ou tel attribut.
- EXECUTE : Droit sur une fonction ou une procédure -> si j'écris une fonction qui modifie MA table, je peux donner les droits d'utiliser cette fonction meme si vous n'avez pas les droits de modifier la table.
- [WITH GRANT OPTION] Si on nous donne un privilège on donne une clef, avec cette option, on donne "le moule de la clef". On peut donc donner ces privilèges à autrui.

Revoquer un privilège

```
REVOKE {ALL, privilège ... } ON nomObjet FROM {users ..., public}
        [CASCADE CONSTRAINTS]
```

Il va aller tuer toutes les clefs étrangères qui ont été construites grace au privilège, sinon REVOKE erreur dans le cas d'existances de ces clefs.

- GRANT ALL = donner passe partout.
- REVOKE ALL = rendre le passe partout. Mais il garde ses privilèges particuliers si il en a.

/! Question d'examen: si on revoke un droit qui avait "with grant option", les "sous-droits" sont ils revoke?

Code d'erreur (gestion des erreurs) :

```
code 0 : TOUT EST OK.
code < 0 : ERREUR, inexecutable
    => détails par valeurs : - syntax, ...
code > 0 : WARNING
    => compris, executé mais des choses sont suspectes.
    ex : supprimer des tuples mais n'a trouvé aucun tuple correspondant.
```

Embedded SQL

Plus qu'un simple outil d'apprentissage car deprecated.

Ce sont des conventions pour pouvoir utiliser sql à l'interieur d'un language de 3 eme generation(C, cobol, ..).

On va écrire un programme dans un language particulier => Host Program, c'est le programme qui recoit (du SQL).

On parlera donc de host-language, c'est le language qui recoit les incersions sql (C++, cobol, ..)

La page de code est donc un melange c++, sql => mais le compilateur n'a besoin QUE de c++. On va donc passer par une pré-compilation faite par un précompilateur. Celui-ci doit connaitre le SGBD utilisé pour connaitre les "drivers", les librairies dynamiques utilisées pour gèrer celui-ci.

Deux modes de précompilations :

- vérif. syntaxique : Vérifie si la syntaxe SQL est correcte.
- vérif. sémantique : Vérifie que tous les éléments existent et qu'on y a accès.

Il faut donner au pré-compilateur le numero d'user, le pw et l'adresse du SGBD.

Chacune des instructions SQL commencera par :

- C++ :

```
EXEC SQL
;
```

- COBOL :

```
EXEC-SQL
END-EXEC
```

On peut mettre “Une requete SQL maximum” dans l’instruction EXEC.

Maintenant, il faut faire passer des informations du language au embedded SQL à l’aide de : **HOST-VARIABLES**, cette variable est faite en host-language et peut être utilisé dans le code sql en utilisant

`:hvariable.`

Cette variable doit avoir un type coompatible avec l’attribut auquel on veut faire correspondre cete variable (voir la doc du précompilateur) (elle peut apparaitre dans le INTO, WHERE, HAVING, INSERT à la place des values)

Ex.: avant de modifier salaire employé, retrouver son nom et son salaire avant de le modifier.

```
EXEC SQL
    SELECT empNom, empSal
    INTO :nom, :sal
    FROM Employe
    WHERE empNo = :num
;
```

Probleme avec les variables qui pourraient etre NULL; il faut donc sans prémunir à l’aide d’indicateur.

```
EXEC SQL
    SELECT empNom, empSal
    INTO :nom, :sal:indSal
    FROM Employe
    WHERE empNo = :num
;
```

`indSal` doit etre déclaré dans le programme hote en binaire pure sur `2 octets`

```
short indsal;
```

```
PIC 9(4) BINARY.
```

Ne pas confondre : “ne rien recevoir” et “recevoir rien” => liste vide et NULL.

Si indSal a une valeur negative - on a reçu NULL. Sinon ”ok”.

Zones de communications :

Cette zone peut être utile (et à déclarer d'office) et est faite d'une pseudo-instruction(= écrire ceci n'ajoute pas l'execution de qqchose au moment de l'execution. On donne un ordre au précompilateur) :

```
EXEC SQL
    INCLUDE SQLCA
;

struct SQLCA {
    ..
    ..
    long sqlcode;
};

EXEC SQL
    SELECT empDpt, empSal
    INTO :dpt,:empsal
    FROM Employe
    WHERE empNom = :nom
;
```

=> Cette instruction ne fonctionnera que si : il ne renvoie rien ou si il ne renvoie qu'un résultat.

Gestion des erreurs (du code de retour) :

Deux manières de faire :

- Tester SQLCA.sqlcode
  - negatif = erreur (plusieurs tuples)
  - 0 = OK
  - positif = warning (pas de tuple = 100)
- Dire au précompilateur quoi faire dans tel ou tel cas.

```
EXEC SQL
    WHENEVER
        {SQL WARNING, SQL ERROR, NOT FOUND}
        {CONTINUE, GO TO host-label} // Tu vas à une gestion d'erreur et
                                     // tu fermes le programme.
;
```

Exemple en Oracle et C++ :

- `VARCHAR`,
- `VARCHAR2(100)`,
- `CHAR(100)`

pstring(varchar) = 2 octets pour dire combien de caractères suivent != cstring(varchar2)

- `VARCHAR maHostVariable[100];` EQUIVALENT A :

```
struct {
    unsigned short len;
    unsigned char arr[100];
} maHostVariable[100];
```

Exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
    int nb;
EXEC SQL END DECLARE SECTION;

int main() {

    EXEC SQL WHENEVER SQLERROR GOTO endLabel;

    EXEC SQL INCLUDE SQLCA.H;
        SELECT COUNT(empNo)
        INTO :nb
        FROM ADT.Employe
    ;
}
```

Curseur :

- Déclaration :

```
DECLARE nomCurseur CURSOR FOR
    SELECT ...
```

Exemple :

```
EXEC-SQL
    DECLARE monCurseur CURSOR FOR
        SELECT *
        FROM ADT.Employe
        WHERE empDpt = :dpt
        ORDER BY empNom
;
```

- Ouverture (en embedded SQL) :

```
EXEC-SQL
    OPEN nomCurseur
;
```

- Fermeture :

```
EXEC-SQL
    OPEN nomCurseur
;
```

- Lecture :

```
EXEC-SQL
    FETCH nomCurseur INTO :v1, :v2, ...
;
```

- EOF :

```
sqlca.sqlcode == 100
```

- Plan d'execution : programme procédurale qui arrive au résultat.

===== Tout ça c'est du "static SQL" =====

On peut utiliser du static SQL lorsque le plan d'execution est calculable à la pré-compilation. Mais souvent, ça ne suffit pas.

===== qui est en opposition au "Dynamic SQL" =====

Souvent fait en deux étapes :

- PREPARE : Construction de la requete dans une String

```
EXEC SQL PREPARE statementNom       // Nom au résultat de la préparation
    FROM { :var, lit }
;
```

- EXECUTE : Faisable autant de fois que l'ont veut.

```
EXEC SQL EXECUTE statementNom
    [USING liste host-variables]
;
```

En dynamique, pour un tuple ou plusieurs : curseur OBLIGATOIRE

- Pour faire un select :

```
string num;
string requete = " SELECT empNom, empNo " +
                 "FROM ADT.employe "+
                 "WHERE empDpt = :no";

stovarchar (statement, requete);
EXEC SQL PREPARE creeTable FROM :statement;
EXEC SQL DECLARE monCurseur CURSOR FOR creeTable;
ask("...", num);

stovarchar(eno, num);

EXEC SQL OPEN monCurseur;
EXEC SQL FETCH monCurseur INTO :eNom, :eNum;

while (sqlca.sqlcode != 100)
{
    cout << enom.arr << " " << enum.arr << endl;
    SQL FETCH monCurseur INTO :eNom, :eNum; u
}
EXEC SQL CLOSE monCurseur;
```

Pour éviter les injections SQL, utiliser une host variable qui est purement formelle (non déclarée dans le host language)

Stored procedures et functions, BD actives

Programmation sur le serveur

Pour ça, il faut un SGBD qui travaille en client-server :

- Oracle
- mySQL
- Derby (java DB)
- postGreSQL
- DB2

Non client-server :

- Access
- SQLlite
- OracleLite
- 1ère génération :
  - Client-server : Le poste possède un petit logiciel client qui prépare la requete, connexion, encapsule, ... Requete transite sur le réseau et retour de résultat.
  - Non client-server : Le contenu transite sur le réseau et la requete reste local, le traitement se fait là. TRANSFERT COLOSSALE.
- 2ème génération : On amène la logique metier (ou des parties) sur le serveurs. => Traitement spécifique au business qu'on implémente ex : banque, calcul d'emprunt est executé chez le client mais est défini par la banque elle même dans le business. Il faut donc les écrire sur le serveur et dans un language :
- Pour Oracle : java(déconseillé) ou PL/SQL(ProgrammingLanguage/SQL). Le serveur créé du "code stocké" quand il compile un code valide à l'aide de :
  - Stored-procedures

    ```
    CREATE PROCEDURE nomProcedure(liste paramètres IN | OUT | IN OUT) IS
    (Code spécifique à l environnement code PL/SQL)
    ```

  - Stored-function

    ```
    CREATE FUNCTION nomFonction(liste paramètres)
        RETURN typeDeRetour
        [DETERMINISTIC]
    ```

On assure, promet renvoie tjs meme valeur si meme paramètre. (Code spécifique à l environnement code PL/SQL)[1] (valable pour tout SGBD).

[1] deterministic

- IN = passage par valeur
- OUT = écriture
- IN OUT = “passage par adresse”

```
SELECT empNo, maFunction(empNom, empSexe) FROM Employe
```

Les fonctions ne peuvent PAS modifier les données (les procédures le peuvent) pour assurer l'intégrité en lecture.

Pour les procédures :

```
EXECUTE maProcédure(valeurs ...)
```

PL/SQL :

Toutes les références sont sur poesi

1. Assigniation :
   - `:=`
2. Comparaison :
   - `=`
3. Separateur d'instruction :
   - `;`
4. Commentaire :
   - `/* */` OU `--`
5. Structure de block :

```
[DECLARE
    -- Déclarations]

BEGIN
    -- Instructions pl/SQL
[EXCEPTION
    -- Gestion des erreurs / RC]
END;
```

6. Data types :

```
char(n)
varchar(n)
...
cursor
Tout ces types ACCEPTENT l'absence de valeur.

ex :
nombre int; OU nombre int := 12;
nom Employe.empNom%type;        // Donner le type qui j'ai donné à un
                               // attribut de la table employe
```

7. Structure de controle :

```
IF condition THEN
    -- instructions
[ELSE
    -- instructions]
END IF;

WHILE condition LOOP
    -- Instructions
END LOOP;
```

Exemples

```
CREATE [OR REPLACE] FUNCTION LibParSexe(sexe char) -- Dans les parametres
            RETURN varchar IS           -- pas de longueur
        BEGIN
            IF sexe = 'M' THEN
                RETURN 'Masculin';
            ELSE
                RETURN 'Feminin';
            END IF;
        END;
```

Utilisation en fonction normale :

```
Select empNo, empNom, libParSexe(empSexe)
    FROM Employe;
```

Pour la tester, utiliser une table créé par oracle appellée DUAL

```
SELECT LibParSexe('M'), LibParSex('F'), LibParSex(null)
    FROM DUAL
```

```
CREATE [OR REPLACE] FUNCTION LibParSexe(sexe char) // Dans les parametres
            RETURN varchar IS           // pas de longueur
        BEGIN
            IF sexe = 'M' THEN
                RETURN 'Masculin';
            ELSE
                IF sexe = 'F' THEN
                    RETURN 'Feminin';
                ELSE
                    -- RETURN '****';
                    -- RAISE APPLICATION_ERROR(-20100, 'Messsage d''erreur')
                    -- doit être entre -20 000 et -20 200
                END IF;
            END IF;
        END;

    SELECT * FROM user_errors -- permet de voir les erreurs de compilation de function

CREATE OR REPLACE FUNCTION rechEmp(eno Employe.empNo%TYPE)
    RETURN Employe.empNom%TYPE IS

    nom Employe.empNom%Type;

        BEGIN
            SELECT empNom INTO nom
                FROM Employe
                WHERE empNo = eno;
            RETURN nom;
        END;

    SELECT rechEmp('050') FROM DUAL

SET SERVEROUTPUT ON -- Active les messages d'erreurs pour la session

CREATE OR REPLACE FUNCTION rechEmp(eno Employe.empNo%TYPE)
    RETURN Employe.empNom%TYPE IS

    nom Employe.empNom%Type;

        BEGIN
            DBMS_OUTPUT.PUT_LINE('Je commence');
            SELECT empNom INTO nom
                FROM Employe
                WHERE empNo = eno;
```

```
            DBMS_OUTPUT.PUT_LINE('Je termine avec ');
            RETURN nom;
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Personne');
                // RETURN ou APPLICATION_ERROR
        END;

    SELECT rechEmp('050') FROM DUAL

CREATE OR REPLACE PROCEDURE
modHierarchie(nvFils Departement.dptNo%TYPE,
                nvPa Departement.dptNo%TYPE) IS

    courant Departement.dptNo%TYPE;
    WHILE ( courant is not null AND courant != nvFils) LOOP
        SELECT dptAdm INTO courant
            FROM Departement
            WHERE dptNo = courant;
    END LOOP;

    IF courant is null THEN
        UPDATE Departement
            SET dptAdm = nvPere
            WHERE dptNo = nvFils;
    ELSE
        RAISE_APPLICATION_ERROR(-20155, 'Un dpt ne peut avoir un aieul
        un de ces descendants');
    END IF;
```

Curseurs :

```
    CURSOR nomCurseur IS
        SELECT ...;

    OPEN nomCurseur;

    FETCH nomCurseur INTO liste hvar;

    CLOSE nomCurseur;

    nomCurseur%FOUND
```

Problème pouvant être rencontré dans le monde réel :

- trouver les étudiants de 1ère triés par le nom :

```
SELECT * FROM Etudiant
    WHERE etuAn = 1
    ORDER BY etuNom;
```

- les noms selons leur code caractère :
  - Dupont
  - DUPON
  - DU PONT
  - dUPonT

Il faudrait donc écrire une fonction qui permet de produire l'élément de comparaison.

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

            chaineInt VARCHAR(2000);
            BEGIN
                chaineInt := lower(chaine);
                return translate(chaineInt, 'éèêëïöüçñ- ''', 'eeeeioucn');
            END;

    SELECT * FROM Ancien
        ORDERBY ToComparableString(ancNom)
```

ou pour les recherche

```
    SELECT * FROM Ancien
        WHERE ToComparableString(ancNom) = ToComparableString(?)
```

Créons un index :

```
    CREATE INDEX NomComparableNdx ON Ancien(ToComparableString(ancNom))
```

3 ème génération : BDD Actives

Un SGBD actif est un sgbd capable de réagir à la survenance d'évenements. - Interdire ou autoriser des accès

Déclencheurs - triggers

- est un triplet E-C-A (pour Evenement - Condition - Action)
- On va se contenter de 3 évenements qui sont INSERT, UPDATE, DELETE

```
CREATE TRIGGER nomTrigger
    {BEFORE, AFTER} -- en parlant de l'action elle même
    {DELETE, INSERT, UPDATE} [OF column [,OF column]..]
    [OR {DELETE, INSERT, UPDATE} [OF column [,OF column]..]]
    [OR {DELETE, INSERT, UPDATE} [OF column [,OF column]..]]
ON table [REFERENCING OLD AS oldName NEW AS newName]
[FOR EACH ROW] -- Reaction pour chacune des lignes
[WHEN (condition)]
bloc_PLSQL;
```

Exemple :

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

CREATE TRIGGER PkTestTrgStable
    BEFORE
    UPDATE of id
    ON TestTrg
    BEGIN
        RAISE_APPLICATION_ERROR(-20100, 'La PK ne peut pas être modifiée');
    END;

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

```
select SeqPourTrg.nextval ...
=> n'utilise pas la notion de transaction
```

On va créer un trigger qui va nourrir la clé primaire :

```
CREATE TRIGGER TestTrgAutoInc
    BEFORE
    INSERT
    ON TestTrg
    REFERENCING NEW AS new
```

```
    FOR EACH ROW
    BEGIN
        :new.id := SeqPourTrg.nextval;
    END;

ALTER TABLE TestTrg
    ADD nomComparable VARCHAR(100);

UPDATE testTrg
    SET nomComparable := ToComparableString(nom);

CREATE OR REPLACE TRIGGER GereNomComparable
    BEFORE
    UPDATE OF nom
    OR INSERT
    ON TestTrg
    REFERENCING NEW AS new
    FOR EACH ROW
    BEGIN
        :new.nomComparable := ToComparableString(:new.nom)
    END;

CREATE TRIGGER SalNeDiminuePas
    BEFORE
    UPDATE OF
    ON Employe
    REFERENCING OLD as old NEW as new
    FOR EACH ROW
    BEGIN
        IF(:new.empSal < :old.empSal) THEN
            RAISE_APPLICATION_ERROR(-20100, "NONNNNNN");
        END IF;
    END;
```

Securité

- Physique : Acheter du matériel de qualité, éviter les destructions, ne pas mettre le matériel n'importe où.
- Accès : Ont accès aux données uniquement les personnes autorisées.
- Logique : ensemble des choses à mettre en oeuvre pour assurer que les données restent cohérentes.

Prévention :

Physique : disposition pour minimiser les risques de survenance des pb's ET mettre en oeuvre des choses pour minimiser les conséquences de pb's.

Conséquence :

Mettre en oeuvre :

- Backup
- Journaling : a chaque modification appliquée, on stock (versionning)

Problèmes de conccurence d'accès :

- Perte d'opération : si deux programmes tournent, peut être qu'un des deux programmes ne sera pas tenu en compte.
- Introduction d'incohérences : Si une C.I. au niveau de notre BD A = B A = 17 B = 17, par exemple, si les deux progrmames travaillent en parellèle ils peuvent faire que la CI ne soit pas respectée.

Verrous (4 types) :

NOTION DE LONGUEUR LIE A LA TRANSACTION

- courts : action
- long : transaction
- partagé (ex. une lecture, un autre fait une lecture) si il est posé sur un granule permet quand meme de déposer un autre verrou partagé.
- Exclusif

nombre de verrous MINIMUM pour ravoir du pessimiste.

Isolation :

- 0 Dirty read :
- 1 ICI :> Pas de dirty Read (niveau par défaut de Oracle à l'école)
  - Commited read; On ne peut avoir que les données commited des autres transactions.

- 2 ICI :> Permet le phantom read
  - Repeatable read
- 3 ICI :> Pas de phantom read
  - Serializable ; niveau le plus haut.

Quand on choisi un degré d'isolation, elle a un effet sur NOTRE transaction pas sur celle des autres. Le résultat obtenu, le gain(ou la parte) que nous allons acquérir avec notre choix de degré d'isolation est completement indépendant du degré d'isolation autour de nous.

Exemple d'implémentation en full pessimistic :

- JavaDB pour 0, 1 et 2
  0. il faut des verrous, en écriture dépot et verrous long exclusif.
  1. le minimum du dirty read doit aussi être fait + en lecture un verrou court partagé.
  2. le minimum du commited read + en lecture un verrou long partagé.
  3. . . . ??