Base de données active

Les SGBDs doivent offrir des fonctionnalités supplémentaires afin de faciliter l'implémentation d'applications plus avancées et plus complexes, les SGBD relationnels restent efficaces, mais il est très utiles d'identifier des fonctionnalités communes à certaines de ces applications avancées et de créer des modèles qui représentent ces fonctionnalités, le développeur peut utiliser ces fonctionnalités directement si elles conviennent à leurs applications, sans devoir les ré implémenter.

Bases de données passives:

Toutes les actions sur les données sont des invocations explicites de programme d'application. Le SGBD fait ce que le programme lui dit.

Exemples de problèmes:

Contrôle d'inventaire : commander des produits lorsque le stock est sous un seuil prédéfini. Ce comportement peut être implémenté avec une base de données passive par deux méthodes :

1→ ajouter une vérification d'une condition dans chaque programme modifiant l'inventaire.
Une mauvaise solution de point de vue de Génie Logiciel.

2→ le programme d'application doit vérifier l'inventaire périodiquement.
Fréquence de vérification très élevé→manque d'efficacité
Fréquence de vérification faible -→ manque de fiabilité du pgm.

D'où l'idée d'ajouter des fonctionnalités au SGBD pour vérifier la situation précédente.

Bases de données Actives :

→Est une extension des SGBDs traditionnels.

Le SGBDA est en général un SGBD qui contrôle des situations d'intérêts, et déclenche l'action appropriée à l'apparition d'une situation.

Ensemble de primitives d'un SGBD traditionnel+

Définition d'un ensemble de situations + Déclencheurs d'un ensemble de réactions.

Ce comportement est exprimé dans des règles de production (nommé aussi règle d'événement- condition-action) Qui sont définies et stockées dans la base de données.

Avantage de base de données active :

1- Simplifier les programmes d'application : une partie de programme peut être programmée avec les règles actives de SGBD.
2- Augmenter l'automatisation : les actions sont déclenchées automatiquement et sans intervention de l'utilisateur.
3- Augmenter la fiabilité de données : plus d'action de vérification et de réparation, mieux aider à la décision.
4- Augmenter la flexibilité : réduire le coût de développement et de maintenance.

Modèle de règle ECA

Le Modèle ECA – Evénement, Condition, Action- est le modèle généralisé appliqué pour spécifier les règles des bases de données actives, dans ce modèle, une règle est constitué de trois composants :

On event
If condition
Then Action

1- Les événements qui déclenchent la règle.
2- La condition qui détermine l'action définie dans la règle qui doit être exécutée.
Si aucune condition n'est spécifiée, l'action est exécutée dés que l'événement se produit. Si une condition est spécifiée, elle est d'abord évaluée et si le résultat du test est vrai, l'action est exécutée.

3- L'action à exécutée est généralement constituée d'une séquence d'instruction SQL, mais il peut également s'agir d'une transaction ou de l'appel d'un programme externe qui sera lancé automatiquement.

WHEN event occurs
Toujours modifie une ligne d'une table de la base de données.
IF condition holds
Elle est souvent une requête SQL attachée à la ligne de la table.
La condition est vraie si le résultat de la requête n'est pas nul
DO execute action
Des modifications SQL ou appel à une procédure stockée
Exemple:
Evénement
Un client qui n'a pas réglé 3 factures avant la date limite
Condition
Si le crédit de client est inférieur à 20 000 DA
Action
Annuler toutes ses commandes
- Les règles peuvent exprimer des diverses aspects liés aux sémantiques d'application :

1- Des contraintes statiques : ex, les cardinalités, les contraintes d'intégrités référentielles, restriction de valeurs…. ;
Seulement les étudiants réguliers peuvent s'enregistrer à la bibliothèque
Un encadreur ne peut pas encadrer plus de quatre groupes.
Le salaire d'un employé ne peut pas dépasser le salaire de son manager.
2- Des contraintes sur la gestion workflow.
Si la commande est acceptée, une facture proformat est rédigée.
3- Des données historiques :
Exemple : les données mensuelles sur les bons de commandes sont transférées à l'entrepôt des données

4- Implementation des relations génériques:
Un lecteur est soit un enseignant ou un étudiant mais pas les deux en même temps.
5- Donnés dérivées : le nombre des étudiants enregistrés dans un cours doit être parmi les attributs du cours.
6- Contrôle d'accès : un employé ne peut consulter que les données de son département.

Exercice : reformuler les exemples ci-dessous par les règles ECA.

Les triggers d'Oracle: (déclencheurs)

Trigger (*déclencheur*) : routine déclenchée automatiquement par des événements liés à des actions sur la base

• Les triggers complètent les contraintes d'intégrité en permettant des contrôles et des traitements plus complexes.

Syntaxe de trigger d'Oracle

• trigger name
<Oracle-trigger> ::= CREATE TRIGGER <trigger-name>
• trigger time point
{ BEFORE | AFTER }
• triggering event(s)
<list of trigger-events>
ON <table-name>
[[REFERENCING <references>]
• trigger type (optional)
FOR EACH ROW
• trigger restriction (only for for each row triggers !)
[WHEN (<condition>)]]
• trigger body
<actions>
<trigger-event> ::= INSERT | DELETE | UPDATE [OF <column-names>]
<references> ::= OLD AS <old-value-tuple-name> |
NEW AS <new-value-tuple-name>
<actions> ::=<PL/SQL block>

- **Nom de trigger** : *doit être unique dans un même schéma.*

• **Temps de trigger:** *il précise le moment de l'exécution du trigger*
- les triggers AFTER row sont plus efficaces que les BEFORE row parce qu'ils ne nécessitent pas une double lecture des données.

- **Liste des événements de trigger:** *Elle comprend le type d'instruction SQL qui déclenche le trigger :*

DELETE, INSERT, UPDATE On peut en avoir une, deux ou les trois.
Pour UPDATE, on peut spécifier une liste de colonnes. Dans ce cas, le trigger ne se déclenchera que si l'instruction UPDATE porte sur l'une au moins des colonnes précisée dans la liste.
S'il n'y a pas de liste, le trigger est déclenché pour toute instruction UPDATE portant sur la table.
La liste des événements du trigger précise la table associée au trigger :
- une et une seule table
- /pas une vue.

- **Références :**

Les pseudo-variables :NEW et :OLD permettent de se référer aux anciennes et nouvelles valeurs des lignes. :NEW a la valeur NULL après une commande delete et :OLD a la valeur NULL après une commande insert. Elles ne sont évidemment utilisables qu'avec l'option _for each row_.
Une option _REFERENCING_ permet de donner un alias aux variables préfixées par :NEW et :OLD :
CREATE TRIGGER totalAugmentation

```
AFTER UPDATE OF sal ON emp
REFERENCING old as ancien, new as nouveau
FOR EACH ROW
update cumul
set augmentation = augmentation + nouveau.sal - ancien.sal
where matricule = ancien.matr
```

- **types de triggers**

L'option for each row est facultative ; elle indique que le traitement du trigger doit être exécutée pour chaque ligne concernée par la requête déclenchante. Sinon, cette commande n'est exécutée qu'une seule fois pour chaque requête déclenchante.
Il existe **deux** types de triggers différents : les triggers de table (STATEMENT) et les triggers de ligne (ROW). Quelle est la différence ?
• Les *triggers de table* sont exécutés **une seule fois** lorsque des modifications surviennent sur une table (même si ces modifications concernent plusieurs lignes de la table). Ils sont utiles si des *opérations de groupe doivent être réalisées* (comme le calcul d'une moyenne, d'une somme totale, d'un compteur, …). Pour des raisons de performance, il est préférable d'employer ces triggers plutôt que les triggers lignes.
• *Les triggers lignes* sont exécutés **« séparément » pour chaque ligne modifiée** dans la table. Ils sont très utiles s'il faut mesurer une *évolution pour certaines valeurs, effectuer des opérations pour chaque ligne en question*.

Lors de la création de triggers lignes, il est possible d'avoir accès à la valeur *ancienne* et la valeur *nouvelle* grâce aux mots clés *OLD* et *NEW*. Il n'est pas possible d'avoir accès à ces valeurs dans les triggers de table.

Exemple de trigger table :

Ce trigger table enregistre dans une table *log* la trace de la modification de la table Emp_tab. On mémorise ici le moment de la modification et l'utilisateur qui l'a provoqué. Il n'est donc exécuté qu'une seule fois par modification de la table Emp_tab.

```
CREATE TRIGGER log
AFTER INSERT OR UPDATE ON Emp_tab
BEGIN
INSERT INTO log(table, date, username, action)
VALUES ('Emp_tab', sysdate, sys_context('USERENV', 'CURRENT_USER'),
'INSERT/UPDATE on Emp_tab') ;
END ;
```

Ce trigger table enregistre dans une table *log* la trace de la modification de la table Emp_tab. On mémorise ici le moment de la modification et l'utilisateur qui l'a provoqué. Il n'est donc exécuté qu'une seule fois par modification de la table Emp_tab.

Exemple de trigger ligne :

Une table cumul sert à enregistrer le cumul des augmentations dont ont bénéficié les employés d'une entreprise.
Exemple:
Soit un système de gestion d'un entrepôt avec deux relations:
-Inventory(Part,PartOnHand,ReorderPoint,ReorderQty)
- PendingOrders(Part,Qty,Date)

Contrainte:il existe au maximum une commande par part dans PendingOrders (Part is the key of
PendingOrders)
La regle de reapprovisionnement genere une nouvelle commande si la quantité PartOnhand d'un part donné est inferieur à ReorderPoint

```
CREATE TRIGGER Reorder
AFTER UPDATE OF PartOnHand ON Inventory
FOR EACH ROW
WHEN (New.PartOnHand < New.ReorderPoint)
DECLARE NUMBER X;
BEGIN
SELECT COUNT(*) INTO X
FROM PendingOrders
WHERE Part = New.Part;
IF X = 0 THEN
INSERT INTO PendingOrders
VALUES (New.Part, New.ReorderQty, SYSDATE)
ENDIF;
END;
```

Oracle: Example of Row-Level After Trigger (2)

Part	PartOnHand	ReorderPoint	ReorderQty
1	200	150	100
2	780	500	200
3	450	400	120

[2] Transaction executed on October 10, 2000

```
UPDATE Inventory
SET PartOnHand = PartOnHand - 70
WHERE Part = 1
```

Result: insertion of (1,100,2000-10-10) into PendingOrders
Transaction executed the same day

```
UPDATE Inventory
SET PartOnHand = PartOnHand - 60
WHERE Part >= 1
```

Result: insertion of (3,120,2000-10-10) into PendingOrders

- **restriction de trigger (seulement pour chaque ligne de trigger!)**

Il est possible d'ajouter une clause WHEN pour restreindre les cas où le trigger est exécuté.
WHEN est suivi de la condition nécessaire à l'exécution du trigger.
Cette condition peut référencer la nouvelle et l'ancienne valeur d'une colonne de la table (new et old ne doivent pas être préfixés par _ :_ comme à l'intérieur du code du trigger.

Exemple :

```
create or replace trigger modif_salaire_trigger
before update of sal on emp
for each row
when (new.sal < old.sal)
begin
raise_application_error(-20001, 'Interdit de baisser le salaire ! (' || :old.nome || ')');
end;
```

raise_application_error est une instruction Oracle qui permet de déclencher une erreur en lui associant un message et un numéro (compris entre -20000 et -20999).

La fin d'un trigger (les exceptions)

Si une erreur se produit pendant l'exécution d'un trigger, toutes les mises à jour produites par le trigger ainsi que par l'instruction qui l'a déclenché sont défaites.

On peut introduire des exceptions en provoquant des erreurs.

. Une exception est une erreur générée dans une procédure PL/SQL.

. Elle peut être prédéfinie ou définie par l'utilisateur.

. Un bloc PL/SQL peut contenir un bloc EXCEPTION gérant les différentes erreurs possibles avec des clauses WHEN.

. Une clause WHEN OTHERS THEN ROLLBACK; gère le cas des erreurs non prévues.

Exceptions prédéfinies : quelques exemples

NO_DATA_FOUND : cette exception est générée quand un SELECT INTO ne retourne pas de lignes

DUP_VAL_ON_INDEX : tentative d'insertion d'une ligne avec une valeur déjà existante pour une colonne à index unique

ZERO_DIVIDE : division par zéro

Exemple :

employe(numserv,....)

service(numserv,...)

```
/* vérifier que le service de l'employé existe bien */
CREATE TRIGGER verif_service
BEFORE INSERT OR UPDATE OF numserv ON employe
FOR EACH ROW WHEN (new.numserv is not null)
DECLARE
noserv integer;
BEGIN
noserv:=0;
SELECT numserv
INTO noserv
FROM SERVICE
WHERE numserv=:new.numserv;
IF (noserv=0)
THEN raise_application_error(-20501, 'N° de service non correct');
END IF;
END;
```

Remarque :

• Un trigger ligne ne peut accéder à une table mutante. «est la table qui est en train d'être modifiée»

• Un trigger par lignes ne peut pas interroger ou modifier une table mutante**.**

Il faut donc comprendre qu'**il n'est pas permis** de consulter ou modifier une table mutante. Cette restriction préservera le trigger ligne de lire des données inconsistantes. Donc votre trigger ligne ne devrait jamais accéder à la table sur laquelle il porte autrement que par OLD et NEW.

Exemple :

```
CREATE OR REPLACE TRIGGER Emp_count
AFTER DELETE ON Emp_tab
FOR EACH ROW
DECLARE
N INTEGER;
BEGIN
```

```
SELECT COUNT(*) INTO n FROM Emp_tab;
DBMS_OUTPUT.PUT_LINE('There are now ' || n || 'employees') ;
END ;
```

Dans cet exemple, la table mutante (celle qui est modifiée par DELETE) est Emp_tab. Or, une clause SELECT porte justement sur cette table. Le risque est d'obtenir un résultat complètement incohérent. Cette consultation est donc interdite.

- Il est autorisé de modifier les valeurs des colonnes au travers de NEW (et donc écrire dans le code PL/SQL :new.val:=2 par exemple), uniquement dans des triggers lignes BEFORE.

Action :

Le corps de trigger est un block PL/SQL , toutes les commandes SQL et PL/SQL peuvent être utilisées dans le ce block.

des structures if additionnelles permettent d'exécuter certaines partie de block PL/SQL dépend du type de l'événement déclencheur du trriger : il existe trois structures : if inserting , if updating < colonne> et if deleting.

```
Begin
if inserting then
<PL/SQL block>
end if ;
if updating then
<PL/SQL block>
end if ;
if deleting then
<PL/SQL block>
end if ;
end;
;
```

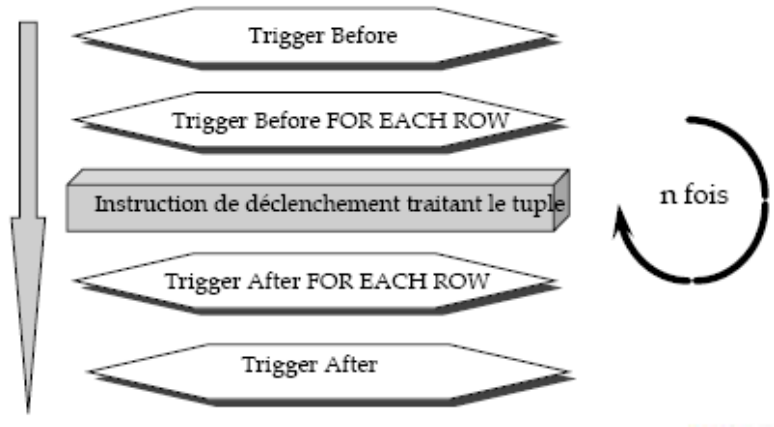
Exemple :

Logging actions, on tables:

```
create trigger LOG EMP
after insert or update or delete on EMP
begin
if inserting then
insert into EMP LOG values(user, 'INSERT', sysdate);
end if ;
if updating then
insert into EMP LOG values(user, 'UPDATE', sysdate);
end if ;
if deleting then insert into EMP LOG values(user, 'DELETE',
sysdate);
end if ; end ;
```

Ordre de déclenchement.

Il peut y avoir plusieurs triggers sur le même événement.



Activation – Désactivation et Groupage des règles :

Une base de données active doit permettre aux utilisateurs d'activer des règles, de les désactiver et de les supprimer en les désignant par leurs noms.
Une règle désactivée ne sera pas déclenchée par l'événement déclencheur , cette caractéristique autorise à désactiver sélectivement des règles pendant les périodes où ne sont pas nécessaires.
La commande ***activate*** réactive la règle.
La commande **drop** éfface la règle du système.
Il est également possible de grouper les règles en ensembles nommés, pour que l'ensemble puisse être activé, désactivé ou supprimé.

```
ALTER TRIGGER [schema.]nom_trig
{ENABLE | DISABLE}
```

Problèmes de conception de BDDActive :
L'une des difficultés à la généralisation des règles actives , réside dans le fait qu'il s'agit de techniques relativement difficiles à utiliser en termes de conception, par exemple il n'est pas évident de vérifier la cohérence d'un ensemble de règles , autrement dit de s'assurer que deux ou plusieurs règles ne sont pas conflictuelles .
Il n'est pas simple de garantir qu'un ensemble de règles se terminera dans toutes les circonstances.
Exemple :
R1 : create trigger T1
After insert on table1
For each row
Update table2
Set attribut1=..;
R2 : create trigger T2
After update attribut1 on table T2
For each Row
Insert into table1 values(……..);