Informations

- Enseignant : Abdessamad IMINE
  email : abdessamad.imine@univ-lorraine.fr
  Bureau : 124
- Deux séances de 2 heures par semaine
  - Cours (**www.loria.fr/~imine/cours/BD.pdf**)
  - TD
  - TP
- Evaluation
  - Note d'examen (1 partiel : début novembre)
  - Note de TP

Introduction aux Bases de Données

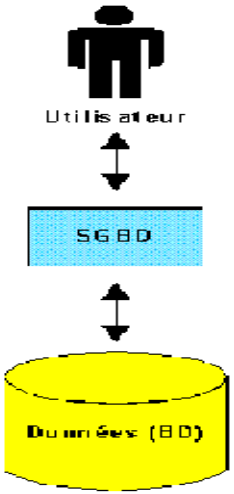
Base de données et système de gestion de bases de données

- BD : un ensemble bien structuré de données relatives à un sujet global et accessible par **plusieurs utilisateurs** à la fois.
- Exemples
  - Une banque stocke les informations sur les clients et leurs dépôts d'épargne dans une BD.
  - BD de réservation de tickets du SNCF.
  - Un service de scolarité stocke les informations relatives aux étudiants inscrits dans une BD...

Système de gestion de bases de données

- **SGBD** : un ensemble de programmes permettant à des utilisateurs de créer et d'utiliser de BDs. Les activités supportées sont :
  - la définition d'une base de données (spécification des types de données à stocker)
  - la construction d'une base de données,

    (stockage des données proprement dites)
  - et la manipulation des données (principalement ajouter, supprimer, retrouver des données).



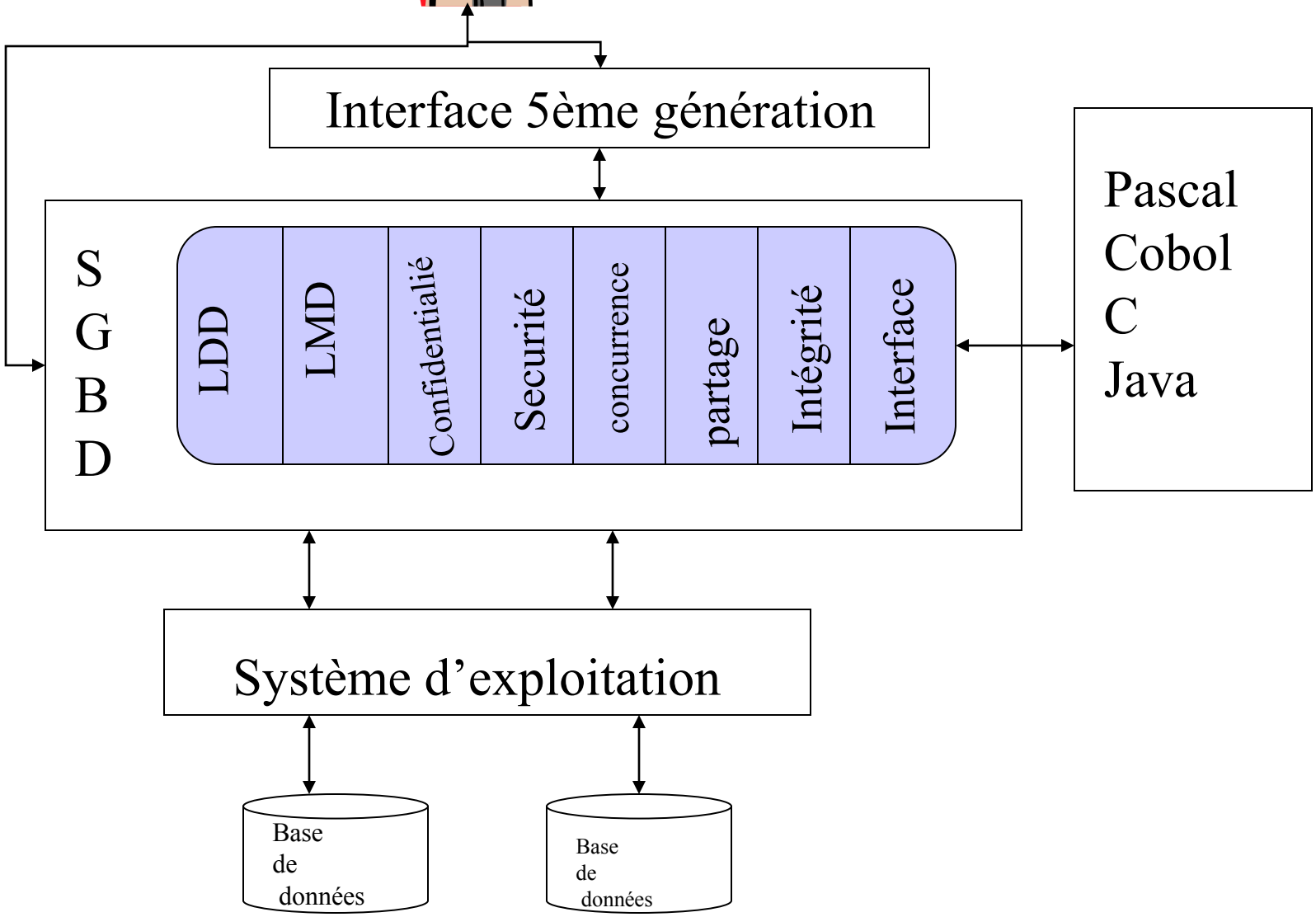
SGBD

- Sépare la partie description des données, des données elles mêmes...
- Cette description est stockée dans un catalogue (également géré dans le SGBD) et peut être consultée par les utilisateurs...
- Les SGBD commerciaux les plus connus sont

  Oracle, Sybase, Ingres, Informix, Access et DB2,
- Les SGBD non commerciaux (free !) les plus connus sont

  MySQL, Postgres, Sybase sous LINUX,
  Oracle sous LINUX...

Architecture Fonctionnelle d'un SGBD



Modèle relationnel de données

- Introduction
- Définition formelle
- Caractéristiques des relations
- Contraintes d'intégrité

Introduction

- 1970 par Codd
- 1ères réalisations (System-R, Ingres), vers 1976.
- Les premiers systèmes commerciaux, au début des années 80.
- Le modèle relationnel est **simple**, facile à appréhender, même pour un non spécialiste.
- Solides bases théoriques: définir de façon formelle les langages de manipulation associés.
- Le **modèle relationnel** représente l'information dans une collection de relations.
  - Une **relation** comme une **table** à double entrée, voire même comme un fichier.
  - Chaque **ligne** de la table (appelée **nuplet** ou **tuple**) peut être vue comme un fait décrivant une entité du monde.
  - Une **colonne** de la table est appelée un **attribut**.

Relation et schéma de relation

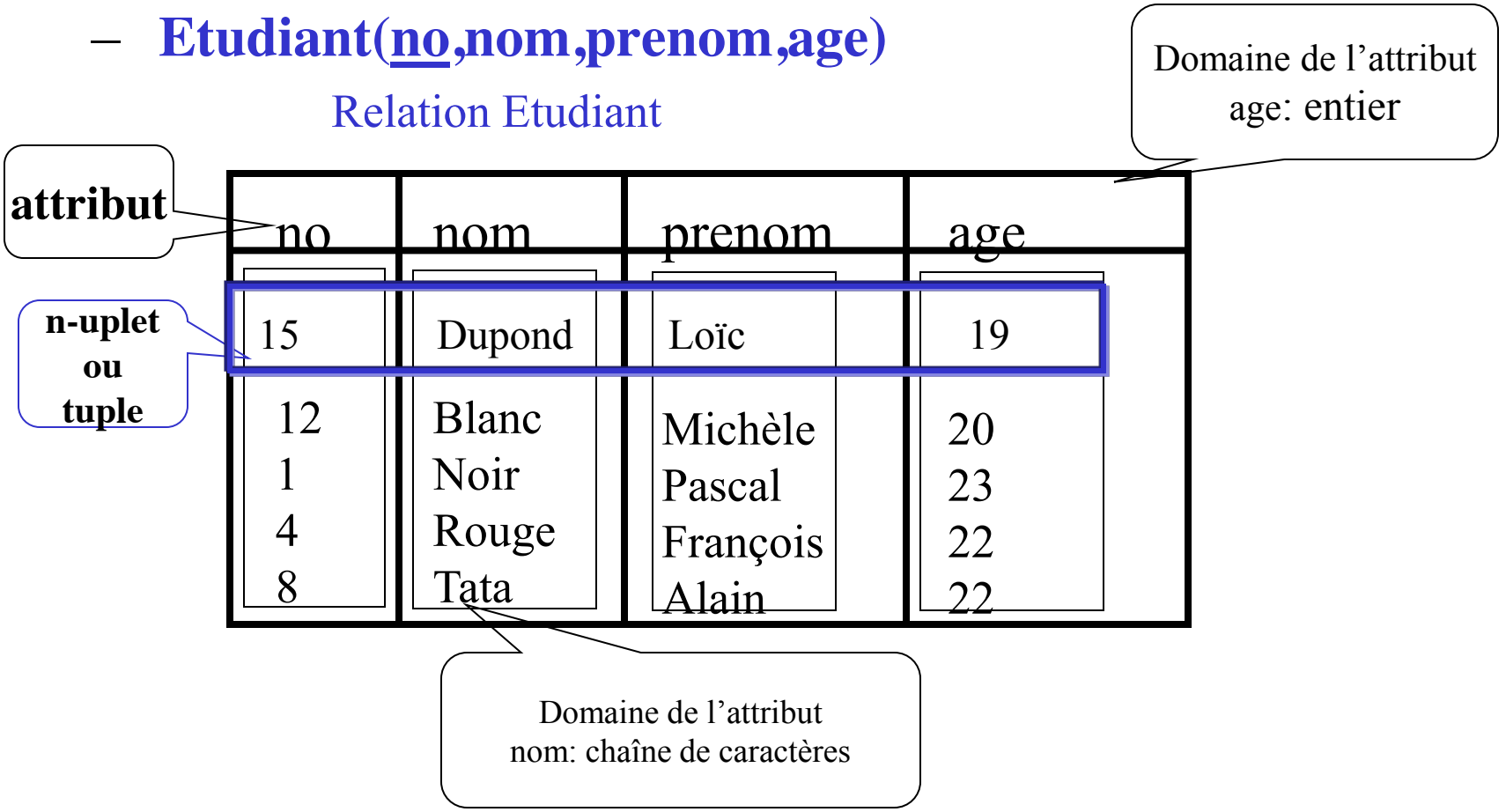
- **Schéma de relation R**
  - R(A1, A2, ..., An) est un ensemble d'attributs R = {A1, A2, ..., An}; chaque **attribut** Ai est associé à un domaine Di
- **Domaine** est un ensemble de valeurs atomiques.

- Une **relation** r sur le schéma de relation R(A1, A2, ..., An) est un ensemble de n-uplets r= {t1, t2, ..., tn} :

  r est souvent appelée **extension** (ou instance) du schéma R.

Schéma de la relation Etudiant:

– **Etudiant(no,nom,prenom,age)**



Extension d'une relation (ou instance): ensemble de ses n-uplets

BD relationnelle: ensemble de relations (variables dans le temps)

Schéma relationnel: ensemble des schémas des relation de la BD

Caractéristiques des relations

- Une **relation** est un ensemble de n-uplets: il n'y a pas d'ordre sur les n-uplets,
- Un **n-uplet** est un **séquence ordonnée** d'attributs...
- Une **valeur d'attribut** est **atomique** mais peut être éventuellement nulle (valeur particulière qui indique que la valeur est manquante).

Contraintes d'intégrité

- Il existe un certain nombre de propriétés qui doivent être respectées pour chaque instance d'un schéma de relation afin de préserver la **cohérence** des informations stockées dans la base. Ces propriétés sont appelées **contraintes d'intégrité**.
- Les dépendances entre données sont des contraintes d'intégrité qui spécifient les relations entre les valeurs des attributs. Elles sont très utiles dans la conception de bases de données.

Contraintes d'intégrité

- Schéma de relation R(U,P)
  - U: liste des attributs et leur domaines
  - P: liste des contraintes d'intégrité CI

- A chaque n-uplet est associé une **clé** qui l'identifie de manière unique.
  - Etudiant(**no,nom,prenom,age**), l'attribut **no** est une **clé primaire**
  - Unicité des valeurs de clé (un seul nuplet étudiant peut avoir une valeur de **no** donnée).
  - Une clé primaire ne peut contenir de valeur nulle.
- Les n-uplets d'une relation sont **distincts** deux à deux

Contraintes d'intégrité

- **CI référentielle entre deux relations**.
  - vérifier que l'information utilisée dans un n-uplet pour désigner un autre nuplet est valide, notamment si le n-uplet désigné existe bien.

    Employe(<u>no_emp</u>, no_emp, adresse_emp, rôle, no_dept)

    Departement(<u>no_dept</u>, nom_dep)
  - un n-uplet de Employe référence un n-uplet de Departement via l'attribut no-dept (numéro de département) ,

Dépendances fonctionnelles entre données

Mauvais schéma relationnel

Relation Produit

prod_id	libellé	pu	qte	dep_id	adr	volume
P1	K7	23.5	300	1	Nancy	9000
P1	K7	23.5	500	2	Laxou	6000
P3	Vis	10	900	4	Metz	2000

- **Redondance** : pu et libellé d 'un produit apparaissent pour chaque occurrence d'un produit ….
- **Insertion** : peut-on insérer un produit sans dépôt ?
- **Risque d'introduction d 'incohérence** : insertion ou mise à jour d'un n-uplet concernant le produit P1
- **Risque de perte d 'information** : suppression du produit P3, on perd son nom, pu et les info. relatives au dépôt 4.

Solution

Pour résoudre ces problèmes, il faut :

- **Etudier la dépendance** entre les données
- **Décomposer** les relations
- **Normaliser** les relations

On obtient ainsi des schémas qui évitent les anomalies citées précédemment.

Dépendances fonctionnelles

Soit R(X,Y,Z) un schéma relationnel

- $X \rightarrow Y$
  X détermine Y ou Y dépend fonctionnellement de Y
- Si étant donnée une valeur de X, il ne lui est associé qu'une seule valeur de Y (pour toute extension de R)
- Une dépendance fonctionnelle est une propriété définie sur l'intention du schéma et non sur son extension
  - Elle est invariante dans le temps et ne peut être extraite à partir d'exemples
  - Elle est extraite de la connaissance de l'application

Exemple

Produit(prod_id, libellé, pu, dep_id, qte, adr, voulme)

DF:

prod_id → libellé

prod_id → pu

dep_id → adr

dep_id →volume

prod_id, dep_id → qte

Ces dépendances fonctionnelles doivent être vérifiées pour chaque extension de la relation Produit

Propriétés des dépendances fonctionnelles

- R(U,F) : U liste d'attributs de R et F une liste de dépendance fonctionnelles vraies dans R.
- On appelle R(U,F) la relation universelle.
- **Axiomes Armstrong** :
  - Réflexivité: $Y \subseteq X \Rightarrow X \rightarrow Y$ (dépendance triviale)
  - Augmentation: $X \rightarrow Y \Rightarrow X \cup Z \rightarrow Y \cup Z$
  - Transitivité: $X \rightarrow Y$ et $Y \rightarrow Z \Rightarrow X \rightarrow Z$

  Avec $X, Y, Z \subseteq U$

Propriétés des dépendances fonctionnelles

- Règles simplifiées :
  - Union : $X \rightarrow Y$ et $X \rightarrow Z \Rightarrow X \rightarrow Y \cup Z$
  - Pseudo-transitivité:
    $X \rightarrow Y$ et $Y \cup W \rightarrow Z \Rightarrow X \cup W \rightarrow Z$
  - Décomposition: $X \rightarrow Y$ et $Z \subseteq Y \Rightarrow X \rightarrow Z$

  Avec $X, Y, Z \subseteq U$

Ces règles peuvent être déduites à partir des axiomes d'Armstrong.

Typologie des dépendances fonctionnelles

- $X \rightarrow Y$ est **triviale** si $Y \subseteq X$

  Exemple : prod_id, libellé $\rightarrow$ libellé

- $X \rightarrow Y$ est **élémentaire** si Y ne dépend pas fonctionnellement d'une partie de X

  (si pour tout $X' \subset X$, $X' \rightarrow Y$ n'est pas vraie)

  Exemple : prod_id, libellé $\rightarrow$ pu ??

  prod_id, dep_id $\rightarrow$ qte ??

Typologie des dépendances fonctionnelles

- X $\rightarrow$Y est **canonique** si sa partie droite ne comporte qu'un seul attribut..
- Un ensemble de DF est canonique si chacune de ses dépendances est canonique (décomposition)
- X$\rightarrow$Y est **directe** si:
  - elle est élémentaire et
  - Y ne dépend pas transitivement de X

Graphe de dépendances fonctionnelles

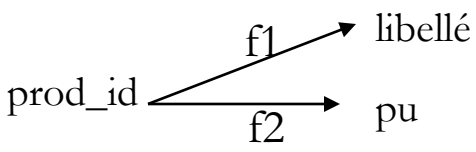
- Nœuds: les attributs impliqués dans les dépendances
- Arcs: les dépendances elles-mêmes
  - de la partie gauche vers la partie droite
  - Ex: DF canonique..

f1: prod_id → libellé

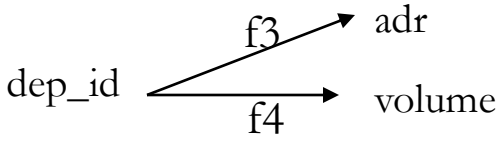
f2: prod_id → pu

f3: dep_id → adr

f4: dep_id → volume



Graphe de dépendances fonctionnelles



Graphe de dépendances fonctionnelles

Ex: DF non canonique..

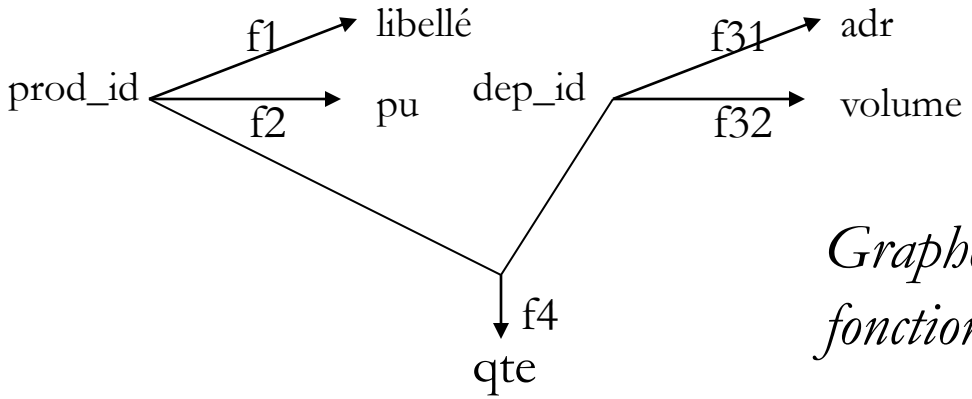
f1: prod_id → libellé

f2: prod_id → pu

f3: dep_id → adr, volume

f4: dep_id, prod_id → qte

Rendre f3 canonique



Graphe de dépendances fonctionnelles

Fermeture

- **Fermeture** (ou clôture) d'un ensemble de DF : l 'ensemble F' de dépendances fonctionnelles obtenu par application successives des axiomes d 'Armstrong +les règles ...
- **Fermeture transitive** d'un ensemble de DF élémentaires : l'ensemble F+ de DF élémentaires obtenu par application de la transitivité ...

Exemple de fermeture

- Exemple

  F={A $\rightarrow$ B ; B $\rightarrow$ C ; D $\rightarrow$ E}

  Enrichir F par d'autres DF en utilisant les axiomes de Amstrong et les autres règles

  F' = F $\cup$ {A $\rightarrow$ C ; A,D $\rightarrow$ E }

Exemples de fermeture transitive

- Exemple 1:

  F={dep_id → adr, adr → volume}

  F+ = F ∪ {dep_id → volume }

- Exemple 2:

  – Avion(no_avion, type, constructeur, capacité, propriétaire)

  F= { f1: no_avion → type,

  f2: type → capacité ,

  f3: type → constructeur, f4: no_avion → propriétaire}

  F+ = ??

Comparaison des DF

- Deux ensembles de DF élémentaires F et F' sont équivalents s'ils ont la même fermeture transitive
- Exemple :
  - Avion(no_avion, type, constructeur, capacité, propriétaire)

F1= { no_avion → type, type → capacité ,
type → constructeur, no_avion → propriétaire}

F2= { no_avion → type, type → capacité , type → constructeur,
no_avion → propriétaire, no_avion → constructeur}

Est-elle vraie cette égalité F1+ = F2+ ???

Couverture minimale

- La couverture minimale d'un ensemble E de DF élémentaires est un ensemble Ê de DF élémentaires tel que:
  - Ê est inclus dans E,
  - Ê est équivalent à E (Ê+ = E+)
  - Ê est minimal, i.e., il n'existe pas E' ⊂ Ê tel que E' soit équivalent à E.

Ê contient les DF élémentaires les plus «pertinentes»...

Exemple 1

E= { f1: A $\rightarrow$ B,

f2: B,C $\rightarrow$ D ,

f3: D $\rightarrow$ E,

f4: A,C $\rightarrow$ D,

f5: A,C $\rightarrow$ E}

Ê = { f1: A $\rightarrow$ B, f2: B,C $\rightarrow$ D , f3: D $\rightarrow$ E}

On a supprimé f5 et f4

Exemple 2

E= { f1: no_avion → type,

f2: type → capacité ,

f3: type → constructeur,

f4: no_avion → propriétaire}

Ê = ??

Clé d'une relation

- Informelle: un ensemble d 'attributs dont les valeurs permettent de distinguer les n-uplets de la relation.
- Formelle: un ensemble d'attributs X constitue une **clé minimale** pour une relation R(X, Y, Z) ssi :
  - $X \rightarrow Y \cup Z$ (unicité)
  - $X \rightarrow Y \cup Z$ est élémentaire (minimalité)
- Une relation accepte plusieurs clés:
  - on choisit une pour être **clé primaire et** les autres sont des clés candidates…
- Une **clé étrangère** est un attribut (ou un groupe d'attributs) appartenant à R1 et qui apparaît comme clé primaire de R2.

Clé d'une relation

Les deux propriétés suivantes permettent de faciliter la recherche de clé minimale :

- **P1 :** tout attribut de R qui ne figure pas dans le membre droit d'une DF non triviale de F, doit appartenir à toute clé de R
- **P2 :** si l'ensemble des attributs de R qui ne figurent pas en membre droit d'une DF non triviale de F est une clé, alors R possède une clé minimale unique composée de l'ensemble de ces attributs

ATTRIBUTS

- A est **attribut clé** s'il appartient à au moins une clé de R
- A est **attribut non clé** s'il n'appartient pas à aucune clé de R
- A est **transitivement dépendant** de X s'il existe Y, Y ne contenant pas A tel que $X \rightarrow Y$, $Y \rightarrow A$, non $(Y \rightarrow X)$ et Y n'est pas inclus dans X
- A est **directement dépendant** de X s'il n'est pas transitivement dépendant
- A est **pleinement dépendant** de X si $X \rightarrow A$ est une DFE
- A est **partiellement dépendant** de X si $X \rightarrow A$ n'est pas une DFE

Projection d'une relation

- Projection

  Extraire une ou plusieurs colonnes d'une table.

  Notation : $\Pi_y(R) = S(Y)$

  Définition informelle :

  La relation S est formée par les colonnes Y de la relation R.

Projection d'une relation

- Exemple de projection

Produit

prod_id	nom	pu
p1	A3	10.0
p2	crayon	9
p3	stylo	15
p4	A4	10.0

- Π_{nom}(Produit) = A3, crayon, Stylo, A4

Jointure de relations

- Jointure

  Faire la jonction entre deux tables ayant au moins un attribut commun.

  Notation : $R = R_1(X) \infty R_2(Y)$ tel que $X \cap Y \neq \varnothing$

  Définition informelle :

  Relier les tuples de R_1 et R_2 pour construire R.

Jointure de relations

- Exemple de jointure

  FILM1 (titre, année, durée, type, studio)

  FILM2 (titre, année, acteurPr)

$$FILM = FILM1 \infty FILM2$$

FILM (titre, année, durée, type, studio, acteurPr)

Décomposition d'une relation en sous-relations

- R(X,Y,Z) est décomposable selon (X,Y) et (X, Z) s'il existe deux relations R1 et R2 telles que :
  - $X \rightarrow Y$
  - $R1 = \Pi_{X,Y}(R)$ et $R2 = \Pi_{X,Z}(R)$
  - $R = R1 \infty R2$ (R est la jointure de R1 et R2)
- le processus de décomposition est **réversible**
  - relation initiale est obtenue par jointure

  => la décomposition doit être sans perte d'information et ne doit pas engendrer de nouveaux tuples lors de la reconstitution de la relation initiale...

Formes normales

- La première forme normale 1FN:
  - une relation est 1FN si chacun de ses attributs a une valeur **atomique**..
- *Personne(id, nom, lesDiplômes)*
  - où lesDiplômes sont l'ensemble des diplômes obtenus par une personne : n'est pas en 1FN

Formes normales

- La première forme normale 1FN:
  - une relation est 1FN si chacun de ses attributs a une valeur **atomique**..
- *Personne(id, nom, lesDiplômes)*
  - où lesDiplômes sont l'ensemble des diplômes obtenus par une personne : n'est pas en 1FN
    Solution : Décomposer la relation
  - Personne(id, nom) en 1FN
  - Diplôme(id, unDiplôme) en 1FN

Formes normales

- La deuxième forme normale 2FN:
  - une relation est en 2FN ssi
    - elle est en 1FN,
    - tout attribut n'appartenant pas à une clé ne dépend pas d'une partie de la clé de R.
- *Stock(<u>prod_id, dep_id</u>, libellé, qte)*

  prod_id, dep_id $\rightarrow$ libellé

  prod_id $\rightarrow$ libellé

  Stock n'est pas en 2FN

2FN

- *Stock(prod_id, dep_id, libellé, qte)*

  Produit(prod_id, libellé) 2FN

  Stock(prod_id, dep_id, qte) 2FN

- 2FN permet de supprimer une certaine redondance (libellé..)

2FN

Pas toutes les redondances

Avion(no_avion, constructeur, type, capacité, propriétaire)

F= {f1: no_avion → type, f2: type → capacité ,

f3: type → constructeur, f4: no_avion → propriétaire}

Relation Avion

no_avion	constructeur	type	capacité	propriétaire
AH32	Boeing	B747	C1	AirFrance
FM34	Airbus	A320	C2	BritishAirways
BA45	Boeing	B747	C1	EgyptAir

Où est la redondance ?

3FN

- La troisième forme normale 3FN :
  - une relation est en 3FN ssi
    - elle est en 2FN,
    - tout attribut n ’appartenant pas à une clé ne dépend pas d ’un attribut non clé
      (ou tout attribut n ’appartenant pas à une clé ne dépend pas transitivement de la clé)

3FN

Relation Avion

no_avion	constructeur	type	capacité	propriétaire
AH32	Boeing	B747	C1	AirFrance
FM34	Airbus	A320	C2	BritishAirwyas
BA45	Boeing	B747	C1	EygptAir

Avion non en 3FN:

type → capacité

type → constructeur type non clé..

Avion(no_avion,type, propriétaire)

Modèle(type, constructeur, capacité) en 3FN

3FN

- Ne permet pas d'éliminer toutes les redondances..
- Ex: *Code_postal(<u>ville,rue</u>, code) en 3FN*

ville, rue→ code et code → ville

Relation Code_postal

code	ville	rue
75000	Paris	Leclerc
54000	Nancy	Leclerc
54000	Nancy	4 églises
54500	Vandoeuvre	Aiguillettes

3FN

- Ne permet pas d'éliminer toutes les redondances..
- Ex: *Code_postal(<u>ville,rue</u>, code) en 3FN*

ville, rue→ code et code → ville

Relation Code_postal

code	ville	rue
75000	Paris	Leclerc
54000	Nancy	Leclerc
54000	Nancy	4 églises
54500	Vandoeuvre	Aiguillettes
54000	Paris	Nabécor

Forme normale de Boyce and Codd

- Une relation R est en 3FN de BCNF ssi:
  - elle est 3FN,
  - et les seules dépendances fonctionnelles **élémentaires** qu'elle comporte sont celles où une **clé** détermine un **attribut**
- code_ville(<u>code</u>, ville) ; code_rue(<u>code</u>, rue) en BCNF
- DF : ville, rue→ code a disparu..
- *Théorème:* toute relation admet une décomposition en BCNF sans perte d 'information..
- BCNF décomposition ne préserve pas généralement les DF

Résumons

- 1FN Tous les attributs sont atomiques
- 2FN 1FN + tout attribut non clé dépend entièrement de chaque clé
- 3FN 2FN + pas de dépendance entre attributs non clé
- BCNF chaque partie gauche d'une dépendance est une clé...

Axiomes Armstrong

- R(U,F) est la relation universelle
  - U liste d'attributs de R et F une liste de dépendance fonctionnelles vraies dans R.
- **Axiomes Armstrong** :
  - Réflexivité: $Y \subseteq X \Rightarrow X \rightarrow Y$ (dépendance triviale)
  - Augmentation: $X \rightarrow Y \Rightarrow X \cup Z \rightarrow Y \cup Z$
  - Transitivité: $X \rightarrow Y$ et $Y \rightarrow Z \Rightarrow X \rightarrow Z$

  Avec $X, Y, Z \subseteq U$

Règles simplifiées

- Union : $X \rightarrow Y$ et $X \rightarrow Z \Rightarrow X \rightarrow Y \cup Z$
- Pseudo-transitivité:
  $X \rightarrow Y$ et $Y \cup W \rightarrow Z \Rightarrow X \cup W \rightarrow Z$
- Décomposition: $X \rightarrow Y$ et $Z \subseteq Y \Rightarrow X \rightarrow Z$

Avec $X, Y, Z \subseteq U$

Ces règles peuvent être déduites à partir des axiomes d'Armstrong.

Conception de schémas relationnels

- Deux algorithmes pour concevoir un schéma en 3FN :
  - Algorithme de synthèse: basé sur le graphe minimal de dépendance...
    - Bottom-Up design
  - Algorithme de décomposition : basé sur la fermeture des dépendances
    - Top-Down design

Algorithme de synthèse

En entrée: R(U, F) /* relation universelle*/

En sortie: S = {Ri} 1 ≤ i ≤ n, schéma relationnel où chaque Ri est en **3NF**

début algorithme

1. Rechercher une couverture minimale F_0 de F

2. Partitionner F_0 en groupes G1,…, Gk telles que toutes les dépendances fonctionnelles d'un même groupe aient la même partie gauche.

3. Pour chaque groupe Gi construire une relation Ri dont la clé est la partie gauche.

4. Si aucune des relations obtenues à l'étape précédente ne contient (ou ne permet pas d'obtenir) la clé de la relation initiale, on ajoute une relation composé des attributs de la clé.

fin algorithme

Exemple

Produit(prod_id, libellé, pu, dep_id, qte, adr, voulme)

Hypothèse: un produit est stocké dans un seul dépôt

f1: prod_id → libellé, pu

f2: prod_id, dep_id → qte

f3: prod_id → dep_id

f4: dep_id → adr, volume

f5: prod_id → adr

f6: prod_id → qte

Exemple

Produit(prod_id, libellé, pu, dep_id, qte, adr, voulme)

Hypothèse: un produit est stocké dans un seul dépôt

f1: prod_id → libellé, pu, f2: prod_id, dep_id → qte

f3: prod_id → dep_id , f4: dep_id → adr, volume

f5 prod_id → adr f6: prod_id → qte

Déterminer une couverture minimale (rendre DF élémentaires directes et canoniques)

1. **canonique** (f1, f4) par décomposition:

   f11: prod_id → libellé, f12: prod_id → pu

   f41: dep_id → adr, f42: dep_id → volume

2. Éliminer f2 car elle n'est pas **élémentaire** (à cause de notre hypothèse).

3. Eliminer les dépendances transitives:

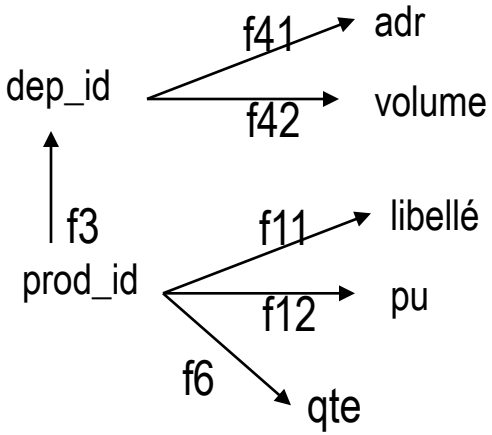
   (f3: prod_id →dep_id) et (f41: dep_id → adr) donne (f5: prod_id → adr)

supprimer f5

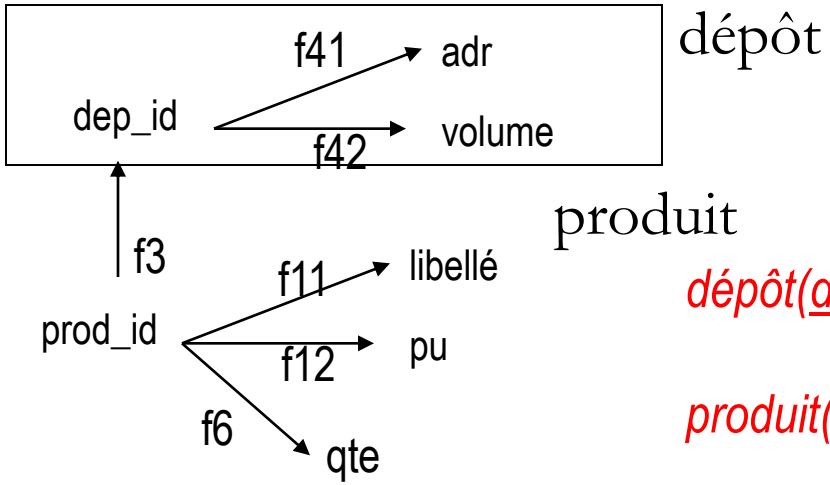
f11, f12, f3, f41, f42, f6 sont élémentaires directes et canoniques

Suite exemple

4. Traçons le graphe associé :



5. Grouper les attributs ayant la même source de DF et la source dans une relation



dépôt(<u>dep_id</u>, adr, volume)

produit(<u>prod_id</u>, libellé, pu, qte, dep_id)

Décomposition d'une relation en sous-relations

- **Théorème**
  - Toute relation a au moins une décomposition en 3FN qui est sans perte d'information et qui préserve les dépendances fonctionnelles.

- **Condition de décomposition**
  - si $X \rightarrow Y$ est vraie dans une relation R(X, Y, Z) alors R est décomposable en R1(X, Y) et R2(X, Z)

Exemple

- Avion(no_avion, type, constructeur, capacité, propriétaire)

  F= { f1: no_avion $\rightarrow$ type, f2: type $\rightarrow$ capacité ,

  f3: type $\rightarrow$ constructeur, f4: no_avion $\rightarrow$ propriétaire}

1ère décomposition (préserve les DF):

Modèle(type, constructeur, capacité)

Avion(no_avion, type, propriétaire)

2ème décomposition (ne préserve pas f4):

Avion1(no_avion, type)

Avion2(type, propriétaire)

Avion3(type, constructeur, capacité)

Pas toute décomposition est intéressante : il faut qu'elle soit sans perte d 'information et préserve les dépendances

Algorithme de décomposition

En entrée: R(U, F) /* relation universelle*/

En sortie: S = {Ri} 1 ≤ i≤n, schéma relationnel où chaque Ri est en 3NF ou en BCNF

début algorithme

S = R(U,F)

Tant qu'il existe dans S un schéma de relation T(X, F_T) qui n 'est pas en 3NF ou BCNF

faire /* appliquer une étape de la décomposition*/

Choisir dans F_T une dépendance non triviale $W \rightarrow V$

Remplacer T par 2 relations de schéma (W V, F_{T1}) et (X\V, F_{T2})

où F_{T1} et F_{T2} sont de DF dérivées de la fermeture de F_T

fin tantque

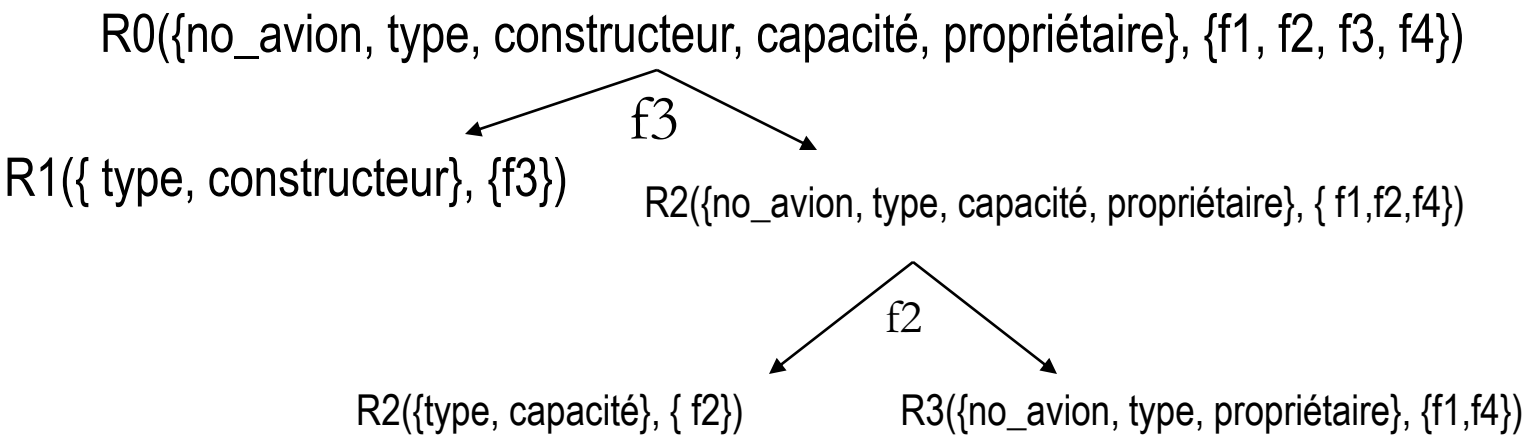
fin algorithme

Problème : la solution dépend du choix des DF selon lesquelles on choisit la décomposition.

Exemple 1 *Décomposition sans perte de DF* : Avion (U, F)

U= {<u>no_avion</u>, type, constructeur, capacité, propriétaire}

F= { f1: no_avion $\rightarrow$ type, f2: type $\rightarrow$ capacité ,
f3: type $\rightarrow$ constructeur, f4: no_avion $\rightarrow$ propriétaire}



3 relations en BCNF:

R1(<u>type</u>, constructeur)

R2(<u>type</u>, capacité)

R3(<u>no_avion</u>, propriétaire, type)

Exemple 2: Décomposition sans perte de DF

U= {no_avion, type, constructeur, capacité, propriétaire}

F= { f1: no_avion $\rightarrow$ type, f2: type $\rightarrow$ capacité ,
f3: type $\rightarrow$ constructeur, f4: no_avion $\rightarrow$ propriétaire}

Appliquer la règle de l'union à f2 et f3:

f23: type $\rightarrow$ capacité, constructeur

Utiliser f23 pour décomposer la relation initiale:

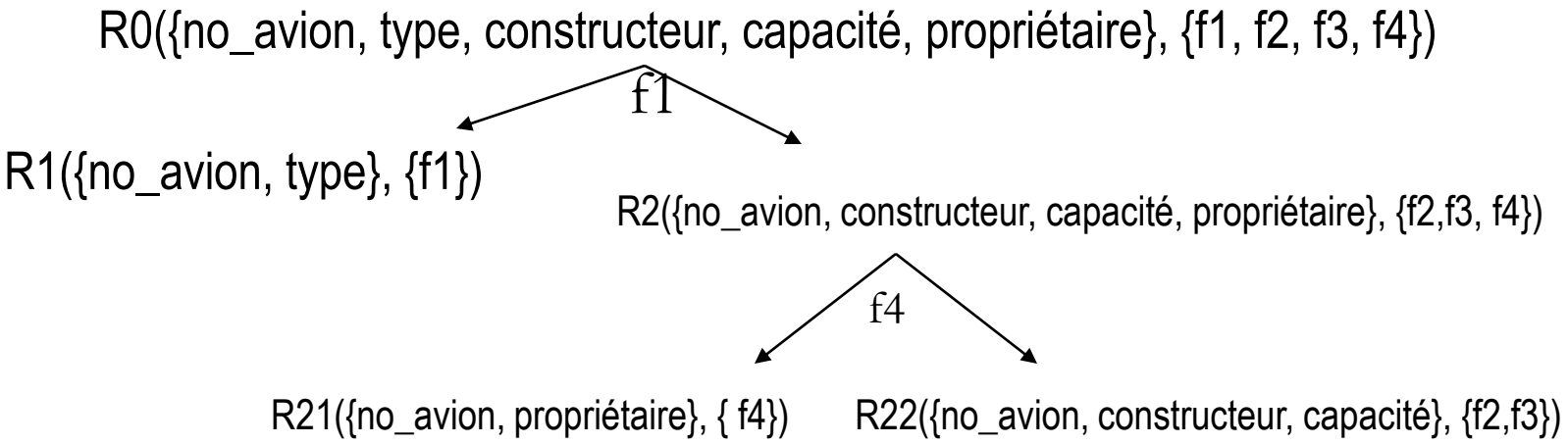
R1({type, constructeur, capacité},{f23})
R2({no_avion, propriétaire, type}, {f1, f4})

R1 et R2 en BCNF et préservant DF

Exemple3: *Décomposition avec perte de DF*: Avion (U, F)

U= {no_avion, type, constructeur, capacité, propriétaire}

F= { f1: no_avion → type,f2: type → capacité ,
f3: type → constructeur, f4: no_avion → propriétaire}



3 relations en BCNF :

R1(<u>no_avion</u>, type)

R21(<u>no_avion</u>, propriétaire)

R22(<u>no_avion</u>, constructeur, capacité)

Perte d'information

Relation Avion

no_avion	constructeur	type	capacité	propriétaire
AH32	Boeing	B747	C1	AirFrance
FM34	Airbus	A320	C2	BritishAirwyas
BA45	Boeing	B747	C1	EygptAir

- R1(no_avion, type) : (AH32, B747)
- R22(no_avion, constructeur, capacité) : (AH32, AirBus, C1)

J'ai perdu « type → constructeur , capacité ».

Conséquence : j'ai perdu également « no_avion → constructeur, capacité »

Conclusion

- Algorithme de décomposition conduit à des relations 3NF ou BCNF
- Inconvénients:
  - le résultat dépend de l'ordre d'application de DF…
  - DF ne sont pas toujours conservées

PL/SQL

Présentation

Pourquoi PL/SQL ?

- SQL est un langage non procédural
- Les traitements complexes sont parfois difficiles à écrire si l'on veut utiliser des variables et les structures de programmation comme les boucles et les alternatives
- On ressent vite le besoin d'un langage procédural pour lier plusieurs requêtes SQL avec des variables et dans les structures de programmation habituelles

Caractéristiques de PL/SQL

- Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
- Un programme est constitué de procédures et de fonctions
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

Utilisation de PL/SQL

- PL/SQL peut être utilisé pour l'écriture des procédures stockées et des triggers (Oracle accepte aussi le langage Java)
- Il convient aussi pour écrire des fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)

Structure d'un programme PL/SQL

```
DECLARE
   -- définition des variables
BEGIN
   -- code du programme
EXCEPTION
   -- code de gestion des erreurs
END;
```

Déclaration, initialisation des variables

- Déclaration et initialisation

  Nom_variable type_variable := valeur;
- Initialisation

  Nom_variable := valeur;

Le type de variables

- VARCHAR2
  - Longueur maximale : 32767 octets
  - Syntaxe:

    Nom_variable VARCHAR2(30);

    Exemple: `name VARCHAR2(30);`

    `name VARCHAR2(30) := 'toto';`

- NUMBER

  Nom_variable NUMBER(long,dec);

  avec Long : longueur maximale

  Dec : longueur de la partie décimale

  Exemple: `num_tel number(10);`

  `toto number(5,2)=142.12;`

Le type de variables (2)

- DATE

  Nom_variable DATE;

  - Par défaut DD-MON-YY (18-DEC-02)
  - Fonction TO_DATE

    Exemple :

```
start_date := to_date('29-SEP-2003','DD-MON-YYYY');
start_date := to_date('29-SEP-2003:13:01','DD-MON-
                      YYYY:HH24:MI');
```

- BOOLEAN
  - TRUE, FALSE ou NULL

PL/SQL

Les principales commandes

Test conditionnel

- IF-THEN
  - IF l_date > ’11-APR-03’ THEN
    l_salaire := l_salaire * 1.15;
    END IF;
- IF-THEN-ELSE
  - IF l_date > ’11-APR-03’ THEN
    l_salaire := l_salaire * 1.15;
    ELSE l_salaire := l_salaire * 1.05;
    END IF;

Test conditionnel

- IF-THEN-ELSIF
  - IF l_nom = ’PAKER’ THEN
    l_salaire := l_salaire * 1.15;
    ELSIF l_nom = ‘ASTROFF’ THEN
    l_salaire := l_salaire * 1.05;
    END IF;
- CASE
  - CASE sélecteur
    WHEN expression1 THEN résultat1
    WHEN expression2 THEN résultat2
    ELSE résultat3
    END;

Test conditionnel

Exemple :

```
val := CASE city
            WHEN ‘TORONTO’ THEN ‘RAPTORS’
            WHEN ‘LOS ANGELES’ THEN ‘LAKERS’
            ELSE ‘NO TEAM’
       END;
```

Les boucles

- LOOP

  instructions exécutables;

  END LOOP;

  Obligation d'utiliser la commande EXIT
- WHILE condition LOOP

  instructions exécutables;

  END LOOP;

Les boucles

- FOR variable IN val_deb..val_fin
  LOOP
      instructions;
  END LOOP;

Affichage

- Activer le retour écran
  - set serveroutput on
- Affichage
  - dbms_output.put_line(chaîne);
  - Utilise || pour faire une concaténation

Exemple n°1

```
DECLARE
   i number(2);
BEGIN
   FOR i IN 1..5 LOOP
   dbms_output.put_line('Nombre : ' || i );
   END LOOP;
END;
```

Exemple n°2

```
DECLARE
   compteur number(3);
   i number(3);
BEGIN
   select count(*) into compteur from clients;
   FOR i IN 1..compteur LOOP
        dbms_output.put_line('Nombre : ' || i );
   END LOOP;
END;
```

Les curseurs

```
DECLARE
    compteur number(3);
    i number(3);
    cursor get_nb_clients IS
    select count(*) from clients;
BEGIN
    OPEN get_nb_clients;
    FETCH get_nb_clients INTO compteur;
    FOR i IN 1..compteur LOOP
            dbms_output.put_line('Nombre : ' || i );
    END LOOP;
    CLOSE get_nb_clients;
END;
```

Les curseurs (2)

```
DECLARE
   nom varchar2(30);
   CURSOR get_nom_clients IS
        SELECT nom,adresse  FROM clients;
BEGIN
   FOR toto IN get_nom_clients
   LOOP
        dbms_output.put_line('Employé : ' ||
        UPPER(toto.nom) ||' Ville : '|| toto.adresse);
   END LOOP;
END;
```

PL/SQL

Procédures et fonctions

Les procédures et fonctions stockées

- Sous-programme stocké dans une base de données et peut être lancé à partir de divers programmes applicatifs
- Avantages :
  - Efficacité dans l'architecture client/serveur
    - Le code est stocké sous forme compilé et n'a pas besoin de transiter sur le réseau. Seul le résultat est renvoyé à l'utilisateur.
  - Réutilisation
    - Le même code peut être utilisé à partir de diverses applications.
  - Sécurité/intégrité
    - Certains tables ne seront accessibles que via les fonctions et les procédures stockées.

Les procédures

```
create or replace procedure list_nom_clients
IS
BEGIN
    DECLARE
        CURSOR get_nom_clients IS
                select nom, adresse  from clients;
    BEGIN
        FOR toto IN get_nom_clients LOOP
        dbms_output.put_line('Employé : ' || UPPER(toto.nom) ||'
    Ville : '|| toto.adresse);
        END LOOP;
    END;
END;
```

Les procédures

```
create or replace procedure list_nom_clients
(ville     IN        varchar2,
result    OUT     number)
IS
BEGIN
DECLARE
CURSOR get_nb_clients IS
   select count(*) from clients where adresse=ville;
BEGIN
  open get_nb_clients;
  fetch get_nb_clients INTO result;
end;
end;
```

Récupération des résultats

- Déclarer une variable

  SQL> variable nb number;
- Exécuter la fonction

  SQL> execute list_nom_clients('paris',:nb)
- Visualisation du résultat

  SQL> print
- Description des paramètres

  SQL> desc nom_procedure

Les fonctions

```
create or replace function nombre_clients
return number
IS
BEGIN
DECLARE
      i number;
CURSOR get_nb_clients IS
   select count(*) from clients;
BEGIN
  open get_nb_clients;
  fetch get_nb_clients INTO i;
      return i;
end;
end;
```

- Exécution

```
execute :v :=  nombre_clients();
```

Procédures et fonctions

- Suppression de procédures ou fonctions
  - DROP PROCEDURE nom_procedure
  - DROP FUNCTION nom_fonction
- Table système contenant les procédures et fonctions : user_objects

Exercices

- Réalisez une procédure list_tables qui donne le nom de toutes vos tables
- Réalisez une procédure UPDATENOM qui remplit correctement la colonne NOM_PIECE de clients par rapport à la table fournisseurs
- Réalisez une procédure UPDATEPRIX qui met à jour tous les prix de la table clients

Procédure LIST_TABLES

```
create or replace procedure list_tables
IS
BEGIN
 DECLARE
  CURSOR get_nom IS
   select table_name from user_tables;
 BEGIN
  FOR toto IN get_nom
    LOOP
    dbms_output.put_line('Nom de la table : ' || toto.table_name);
    END LOOP;
 END;
END;
```

Procédure UPDATENOM

```
CREATE OR REPLACE PROCEDURE updatenom
IS
 BEGIN
  DECLARE
    nompiece varchar2(30);
    cursor toto IS
          SELECT  distinct fournisseurs.reference, fournisseurs.nom_piece
          FROM fournisseurs;
  BEGIN
     FOR nompiece IN toto LOOP
              UPDATE clients
              SET clients.nom_piece=nompiece.nom_piece
              WHERE clients.reference=nompiece.reference;
     END LOOP;
   END;
 END;
```

Procédure UPDATEPRIX

```
CREATE OR REPLACE PROCEDURE updateprix
 IS
BEGIN
 DECLARE
    prixunit number(5);
    CURSOR toto IS
          SELECT clients.nom, clients.adresse, fournisseurs.prix_piece_unite
          FROM fournisseurs,clients
          WHERE fournisseurs.reference=clients.reference;
 BEGIN
   FOR prixunit IN toto LOOP
     UPDATE clients
     SET clients.prix=clients.quantite*prixunit.prix_piece_unite
     WHERE prixunit.nom=clients.nom and prixunit.adresse=clients.adresse;
   END LOOP;
 END;
END;
```

Les déclencheurs (trigger)

- Automatiser des actions lors de certains événements du type :

  AFTER ou BEFORE et

  INSERT, DELETE ou UPDATE

- Syntaxe :

  CREATE OR REPLACE TRIGGER nom_trigger
  Evénement [OF liste colonne] ON nom_table
  WHEN (condition) [FOR EACH ROW]
  Instructions PL/SQL ou SQL

Exemple de création

SQL> CREATE OR REPLACE TRIGGER pasDeDeleteDansClient
2 BEFORE DELETE ON CLIENT
3 BEGIN
4 RAISE_APPLICATION_ERROR(-20555, ’Stop ...');
5 END;
6 / Déclencheur créé.

SQL> DELETE **FROM** CLIENT;
DELETE **FROM** CLIENT * ERREUR à la ligne 1 :
ORA-20555: Stop ...
ORA-06512: à *”PASDEDELETEDANSCLIENT"*,
ligne 2 ORA-04088: erreur lors d exécution du déclencheur *’PASDEDELETEDANSCLIENT'*

Combinaison d'événements

```
CREATE OR REPLACE TRIGGER afficheEvenement
BEFORE INSERT OR UPDATE OR DELETE ON CLIENT
FOR EACH ROW
BEGIN
    IF INSERTING THEN DBMS_OUTPUT.PUT_LINE('Insertion dans CLIENT');
    ELSIF UPDATING THEN DBMS_OUTPUT.PUT_LINE('Mise a jour dans
CLIENT');
    ELSE DBMS_OUTPUT.PUT_LINE('Suppression dans CLIENT');
    END IF;
END;
```

Accès aux valeurs modifiées

- Utilisation de new et old
- Si nous ajoutons un client dont le nom est toto

  alors nous récupérons ce nom grâce à la variable new.nom
- Dans le cas de suppression ou modification, les anciennes valeurs sont dans la variable old.nom

Exemple

- Archiver le nom de l'utilisateur, la date et l'action effectuée (toutes les informations) dans une table LOG_CLIENTS lors de l'ajout d'un client dans la table CLIENTS
  - Créer la table LOG_CLIENTS avec la même structure que CLIENTS
  - Ajouter 3 colonnes USERNAME, DATEMODIF, TYPEMODIF

Exemple

```
create or replace trigger logadd
after insert on clients
for each row
begin
insert into log_clients values
   (:new.nom,:new.adresse,:new.reference,:new.nom_
   piece, :new.quantite,:new.prix,:new.echeance,
   USER,SYSDATE,'INSERT');
end;
```

Accès aux lignes en modification

```
CREATE OR REPLACE TRIGGER pasDeBaisseDeSalaire
BEFORE UPDATE ON EMP
FOR EACH ROW
BEGIN
IF (:old.sal > :new.sal) THEN
        RAISE_APPLICATION_ERROR(-20567, 'Pas de baisse de salaire !');
END IF;
END;
```

Les exceptions et erreurs

- NO_DATA_FOUND
  - Quand Select into ne retourne aucune ligne
- TOO_MANY_ROWS
  - Quand Select into retourne plusieurs lignes
- OTHERS
  - Toutes erreurs non interceptées
- RAISE_APPLICATION_ERROR
  - Pour retourner une message d'erreur
  - Numéro d'erreur entre -20000 et -20999

TRANSACTIONS

Transaction : Définition

- Une transaction est un programme composé d'une ou plusieurs opérations d'accès aux objets de la BD.
- Une transaction est une unité logique de traitement.
- Une transaction fait passer la BD d'un état cohérent E1 à un autre état cohérent E2.

Transaction : Problèmes

Deux problèmes à prendre en compte :

- Tolérance aux pannes
  - Restaurer la BD dans l'état avant l'incident
  - Solution : tenue d'un journal et reprise après panne
- Exécution concurrente des transactions
  - Eviter l'incohérence des données
  - Solution : verrouillage et gestion des interblocages

Transaction : Exemple

Transfert d'argent entre les comptes

1. **lire (A)**
2. **A := A -100**
3. **écrire (A)**
4. **lire (B)**
5. **B := B+100**
6. **écrire (B)**

Transaction : Propriétés ACID

- Atomicité : *tout ou rien*
  - Exemple : s'il y a échec après l'étape 3 et avant l'étape 6, le système doit assurer que A et B n'ont pas changé.
- Cohérence : *état cohérent après exécution*
  - Exemple : Somme des soldes de A et B est inchangé après l'exécution de la transaction.

Transaction : Propriétés ACID

- Isolation : *pas d'incohérence liée à la concurrence*
  - Exemple : si une autre transaction essaie d'accéder à la BD entre les étapes 3 et 6, elle ne verra pas les modifications faites aux étapes 1 et 3.
  - Exécution des transactions en séquence.
- Durabilité : *effet persistant après exécution*
  - Données durables même après les défaillances logicielles et matérielles.

Transaction : Etats

- Active
  - état initial pour démarrer l'exécution
- Partiellement validée
  - Lorsque la dernière opération a été exécutée
- Echouée
  - Dès que l'exécution ne peut plus continuer normalement

Transaction : Etats

- Annulée
  - L'effet de la transaction est invalidé
  - Soit on redémarre la transaction, soit on la tue
- Validée
  - L'effet de la transaction est entériné après une exécution totalement terminée

Transaction : Commit et Rollback

- Commit
  - Valider une transaction (achever la transaction)
  - Modifications demeurent permanentes
- Rollback
  - Annuler une transaction (achever la transaction)
  - Aucun effet sur la BD (on revient à l'état initial)

Transaction : Autre exemple

On suppose la relation :

MagAFruits(Magasin, fruit, prix)

Supposons que le magasin *Exotique* vend :

la mangue (3,00 €) et le kiwi (2,50 €).

Paul interroge la BD pour connaître le prix maximum et le prix minimum du magasin Exotique.

Entre temps, le magasin Exotique décide de cesser de vendre la mangue et le kiwi pour ne plus vendre que de l'avocat (3,50 €).

Transaction : Autre exemple

Paul exécute les requêtes SQL suivantes :

(max) select max(prix) from MagAFruits
where magasin = 'Exotique';

(min) select min(prix) from MagAFruits
where magasin = 'Exotique';

Transaction : Autre exemple

Pendant ce temps, le magasin *Exotique* exécute les requêtes suivantes :

(del) delete from MagAFruits
where magasin = 'Exotique';

(ins) insert into MagAFruits
values('Exotique', 'Avocat', 3.50);

Transaction : Autre exemple

Contraintes : (max) avant (min)

(del) avant (ins)

Prix	2,50 | 3,00	2,50 | 3,00		3,50
Instruction	(max)	(del)	(ins)	(min)
Résultat	3,00			3,50

Transaction : Autre exemple

Contraintes : (max) avant (min)

(del) avant (ins)

Prix	2,50 | 3,00	2,50 | 3,00		3,50
Instruction	(max)	(del)	(ins)	(min)
Résultat	3,00			3,50

Paul verra que max < min !

Transaction : Oracle

Une transaction débute avec la première instruction SQL et se termine :

- lors de l'exécution de commit ou rollback
- lors de la déconnexion de l'utilisateur (commit)
- lors d'un arrêt anormal du processus utilisateur (rollback)

Tolérance aux Pannes (TP)

Plusieurs incidents peuvent survenir sur une BD :

- Échec d'une transaction
  - Erreur logique ou système (e.g. Insertion incorrecte)
- Crash système
  - Perte des données en mémoire centrale
- Panne disque
  - Perte physique des données

TP : Récupération

- Les algorithmes de récupération (recovery) sont des techniques qui permettent aux SGBD d'assurer l'*atomicité*, la *cohérence* et la *durabilité* malgré les incidents.
- Faire revenir la BD vers l'état initial et cohérent avant l'exécution des transactions.

TP : Récupération

Un journal contient toutes les actions exécutées.

- Démarrage : enregistrer $<$*début transaction*, $T_i>$
- Écriture : enregistrer $<T_i, X, V_1, V_2>$
- Lecture : enregistrer $<$*lecture*, $T_i, X>$
- Validation : enregistrer $<$*valider*, $T_i>$
- Annulation : enregistrer $<$*annuler*, $T_i>$

TP : Récupération

Oracle utilise deux fichiers :

- Fichier journal redo log
  - Il renregistre toutes les modifications faites (insert, update, delete).
- Fichier rollback segment
  - Il enregistre les anciennes valeurs des lignes des tables modifiées par une transaction.

TP : Récupération

Lors d'un commit d'une transaction, il faut :

- Enregistrer le commit dans le fichier redo log
- Enregistrer l'effet de la transaction dans le fichier rollback segment
- Débloquer les lignes verrouillées par la transaction

TP : Récupération

Lors d'un rollback d'une transaction, il faut :

- Redonner les anciennes valeurs à toutes les lignes des tables modifiées par la transaction à partir de rollback segment
- Débloquer les lignes verrouillées par la transaction

TP : Reprise après une panne

Le mécanisme de reprise est le suivant :

- Pour toutes les transactions sans commit on remet les lignes des tables concernées aux anciennes valeurs (rollback)
- Appliquer toutes les modifications enregistrées dans le fichier redo log à la BD (rollforward)

TP : Reprise après une panne

On considère que les modifications faites après un point de reprise (ou point de contrôle).

Un point de reprise consiste à :

1) écrire le fichier redo log sur disque

2) écrire la BD sur disque

3) écrire un article point de reprise dans le fichier redo log

TP : Reprise après une panne

SQL	Résultats
SAVEPOINT A;	Premier point de reprise de la transaction
DELETE …;	Modification de la base
SAVEPOINT B;	Deuxième point de reprise de la transaction
INSERT INTO …;	Modification de la base
ROLLBACK TO B;	L'ordre INSERT est annulé, le point B reste défini
ROLLBACK TO A;	L'ordre DELETE est annulé, le point B est perdu, le point A reste défini
UPDATE …;	Modification de la base
COMMIT;	Valide l'action effectuée par le UPDATE (dernière modification) . Toutes les autres modifications ont été annulées avant le COMMIT.

Gestion de la Concurrence

- Deux transactions sont concurrentes si elles accèdent simultanément aux mêmes données.
- Un SGBD doit disposer d'un gestionnaire de concurrence pour éviter que la mise-à-jour simultanée des mêmes données par plusieurs utilisateurs n'introduise des incohérences.

Gestion de la Concurrence : Problèmes

- Perte des mise à jour

Temps	Transaction T1	Etat de la base	Transaction T2
t1	Lire (A)	A = 10	…
t2	…		Lire (A)
t3	A = A + 10		…
t4	…		…
t5	…		A = A + 50
t6	Ecrire (A)	A = 20	…
t7	…	A = 60	Ecrire (A)

On a perdu la première mise à jour

Gestion de la Concurrence : Problèmes

- Lecture impropres

Temps	Transaction T1	Etat de la base	Transaction T2
t1	Lire (A)	A = 10	…
t2	A = A + 20		…
t3	Ecrire (A)	A = 30	…
t4	…		Lire (A)
t5	Rollback	A = 10	…

La valeur lue et traitée par T2 est alors incorrecte

Gestion de la Concurrence : Problèmes

- Lectures non reproductibles

Temps	Transaction T1	Etat de la base	Transaction T2
t1	…	A = 10	Lire(A)
t2	Lire (A)		…
t3	A = A + 10		…
t4	Ecrire (A)…	A = 20	…
t5	…		Lire (A)

T2 lit la même donnée A à 2 instants différents et n'obtient pas la même valeur

Gestion de la Concurrence : Verrouillage

Principe

- Un verrou est une variable binaire associée à une donnée qui décrit si elle disponible ou non
- Verrou V = 0 sur X alors X est disponible
- Verrou V = 1 sur X alors X n'est pas accessible.
- Un verrou binaire applique une exclusion mutuelle sur la donnée ==> Trop restrictif
- Deux types de verrous : verrous partagés et verrous exclusifs

Gestion de la Concurrence : Verrouillage

Exemple

T1	**T2**
Lock(c1)	…
Read(c1) for update	T2 attend la libération de c1
c1 := c1 - m1	(demande accès pour update de c1)
Écrire(c1)	…
Unlock(c1)	Lock(c1)
	Read(c1) for update
	c1 := c1 - m2
	Écrire(c1)
	Unlock(c1)

Gestion de la Concurrence : Verrouillage

Interblocage (Deadlock)

Il survient lorsque ***chaque*** transaction T d'un ensemble de ***n (n≥2) transactions*** attend une donnée verrouillée par une autre transaction T' de l'ensemble.

T1	T2
Lock(c1)	…
Read(c1)	…
c1 := c1 + m1	…
Écrire(c1)	Lock(c2)
« T1 attend libération de c2 »	Read(c2)
	c2 := c2 + n1
	Ecrire(c2)
	« T2 attend libération de c1 »

Gestion de la Concurrence : Verrouillage

Résolution de l'interblocage

Deux solutions sont utilisées :

1) Si le système détecte l'interblocage, il tue une (ou plusieurs) des transactions impliquées. Le choix peut se faire selon la plus récente, celle qui a fait le moins de mises à jour, …

2) Le système utilise des mises hors délais (timeouts) pour arrêter les attentes des transactions.

Transaction : Exemple

Paul exécute les requêtes SQL suivantes :

(max) select max(prix) from MagAFruits
where magasin = 'Exotique';

(min) select min(prix) from MagAFruits
where magasin = 'Exotique';

Pendant ce temps, Jo du magasin *Exotique* exécute les requêtes suivantes :

(del) delete from MagAFruits
where magasin = 'Exotique';

(ins) insert into MagAFruits
values('Exotique', 'Avocat', 3.50);

Gestion de la Concurrence : Verrouillage

Niveaux d'isolations

SET TRANSACTION ISOLATION LEVEL X

X = **SERIALIZABLE** : c'est le mode le plus contraignant, les transactions sont exécutées dans un ordre séquentiel

Exemple : On suppose l'exemple précédent Paul = (max)(min) et Jo = (del)(ins)

Si Paul choisi le mode SERIALIZABLE alors il verra la base dans l'état soit avant les modifications de Jo, soit après, mais pas entre les deux opérations de Jo.

Gestion de la Concurrence : Verrouillage

Niveaux d'isolations

SET TRANSACTION ISOLATION LEVEL X

X = **REPEATED READ** : on ne voit que des données modifiées par des transactions validées et les lectures multiples donnent le même résultat, si ce n'est que de nouveaux tuples peuvent apparaître entre deux lectures.

Exemple : Si Paul a choisi ce niveau et que l'ordre d'exécution est :

(max)(del)(ins)(min), alors max verra les prix 2.50 et 3.00 et min verra 3.50 mais également 2.50 et 3.00.

Gestion de la Concurrence : Verrouillage

Niveaux d'isolations

SET TRANSACTION ISOLATION LEVEL X

X = **READ COMMITTED** : on ne lit que des données modifiées par des modifications validées, mais des lectures multiples des mêmes données peuvent produire un résultat différent.

Exemple : Si Paul a choisi le niveau READ COMMITTED alors il peut voir les valeurs validées mais pas nécessairement les mêmes à chaque requête. L'entrelacement (max)(del)(ins)(min) est autorisé aussi longtemps que Jo valide (mais Paul peut voir max < min).

Gestion de la Concurrence : Verrouillage

Niveaux d'isolations

SET TRANSACTION ISOLATION LEVEL X

X = **READ UNCOMMITED** : la lecture de données modifiées par d'autres transactions qui ne sont pas encore validées est possible.

Exemple : Si Paul a choisi ce niveau, il peut voir le prix 3.50 même si Jo abandonne la transaction.

Gestion de la Concurrence : Verrouillage

Verrouillage et niveaux de granularité

- La performance du contrôle de la concurrence dépend de la granularité du verrouillage : attribut, tuple, table ou toute la base.
- Plus la taille des éléments à verrouiller est importante plus le degré de concurrence autorisé est faible.
- Si on verrouille des éléments de taille plus petite, il peut y avoir beaucoup d'éléments à verrouiller (beaucoup d'opérations de verrouillage/déverrouilage).

Verrouillage sous Oracle

Principe

- Verrouillage au niveau de la ligne
- On ne voit que les données validées au moment du SELECT
- La première transaction qui accède à une donnée positionne un verrou sur les données accédées (tuples ou tables), les autres transactions souhaitant accéder les mêmes données sont mises en attente.
- Ce mécanisme ne prend pas compte les lectures non reproductibles
- Transaction à lecture seulement (Read Only) pour prendre en compte les lectures non reproductibles.

Verrouillage sous Oracle

Niveaux d'isolation

SET TRANSACTION ISOLATION LEVEL {SERALIZABLE | READ COMMITTED}

- READ COMMITED : c'est le mode par défaut d'Oracle qui empêche les principaux problèmes de concurrence mais avec des les lectures non reproductibles.

- SERIALIZABLE : ce mode est coûteux puisqu'il limite le fonctionnement en parallèle des transactions (les transactions s'exécutent concurremment comme si elles s'exécutaient les unes après les autres).

Verrouillage sous Oracle

Verrouillage de niveau tuple

Lorsqu'une transaction pose un verrou sur un tuple, Oracle effectue les opérations suivantes :

- Un verrou DML (Data Manipulation Language) est posé. Ce verrou évite que d'autres transactions verrouillent ce tuple. Ce verrou ne sera relâché que lorsque la transaction qui a posé le verrou aura été validée ou annulée.

- Un verrou DDL (Data Dictionnary Language) est posé sur la table afin d'éviter les modifications structurelles de la table (suppression de la table, retrait ou ajout d'un attribut, …). Ce verrou ne sera relâché que lorsque la transaction qui a posé le verrou aura été validée ou annulée.

Verrouillage sous Oracle

Deux modes de verrouillage

- Mode exclusif (X)
  Empêche les ressources verrouillées d'être partagées. C'est le mode privilégié pour mettre à jour des données.
- Mode partagé (S)
  Permet à une ressource d'être partagée. Plusieurs transactions peuvent avoir un verrou partagé et peuvent ainsi éviter qu'une autre transaction obtienne un verrou exclusif.

Oracle gère les types de verrous suivants :

row share (RS), row exclusive (RX), share (S), share row exclusive (SRX) et exclusive (X)

Verrouillage sous Oracle

Row Share (RS)

- Principe
  La transaction pose un verrou sur les tuples d'une table dans l'intention de les modifier.
- Opérations autorisées
  Les autres transactions peuvent *lire* tous les tuples de la table (SELECT) et peuvent *insérer, supprimer et mettre à jour* les tuples de la table (**sauf ceux verrouillés**).
  Ils peuvent *obtenir simultanément* les verrous de type RS, RX, S ou SRX.
- Opérations non autorisées
  Il empêche les autres transactions d'avoir un accès exclusif (X) en écriture sur la table.

Verrouillage sous Oracle

Exemples sur Row Share (RS)

SESSION 1	SESSION 2
select job from emp where job = 'CLERK’ for update of empno; OK	select job from emp where job = 'CLERK’ for update of empno; Waiting .... select job from emp where job = 'MANAGER’ for update of empno; OK

Verrouillage sous Oracle

Row Exclusive (RX)

- Principe
  La transaction qui possède le verrou a effectué des m-à-j sur les tuples de la table.
- Opérations autorisées
  Les autres transactions peuvent *lire* **(sauf ceux verrouillés)** les tuples de la table (SELECT) et peuvent *insérer, supprimer et mettre à jour* les tuples de la table (**sauf ceux verrouillés**).
  Ils peuvent *obtenir simultanément* les verrous de type RS ou RX.
- Opérations non autorisées
  Il empêche les autres transactions d'avoir un accès exclusif (X) en lecture et en écriture sur la table (**pas de S et X**).

Verrouillage sous Oracle

Exemples sur Row Exclusive (RX)

SESSION 1	SESSION 2
update emp set ename = 'Zahn‘ where empno = 7900; commit; OK	lock table emp in exclusive mode; Waiting ....

Verrouillage sous Oracle

Share (S)

- Principe

  Acquis par : **LOCK TABLE tab IN SHARE MODE**

- Opérations autorisées

  Permet aux autres transactions les opérations suivantes : SELECT, rechercher des lignes spécifiques avec SELECT … FOR UPDATE ou d'avoir un verrou S.

- Opérations non autorisées

  Il empêche les autres transactions de modifier la même table et de poser les verrous SRX, X, RX.

Verrouillage sous Oracle

Exemples sur Share (S)

SESSION 1	SESSION 2
lock table emp in share mode; OK update emp set ename = 'Zahn' where empno = 7900; commit; OK update emp set ename = 'Müller' where empno = 7900; Waiting	select * from emp; OK lock table emp in share mode; OK

Verrouillage sous Oracle

Share Row Exclusive (SRX)

- Principe

  Acquis par : **LOCK TABLE tab IN SHARE ROW EXCLUSIVE MODE**

- Opérations autorisées

  Une seule transaction pose SRX à la fois. Les autres ne peuvent que consulter la table, ou poser un verrou RS (mais elles ne peuvent pas modifier).

- Opérations non autorisées

  Empêche les autres transactions de modifier la table, ou d'obtenir en même temps les verrous S, SRX et X.

Verrouillage sous Oracle

Exclusive (X)

- Principe

  La transaction a un accès en écriture exclusive. Il est acquis par :
  LOCK TABLE tab IN EXCLUSIVE MODE

- Opérations autorisées

  Une seule transaction pose X à la fois. Les autres ne peuvent que consulter la table.

- Opérations non autorisées

  Empêche les autres transactions de modifier la table, ou d'obtenir n'importe quel autre type de verrou.