Bases de données (avancées)

Cours: Fabrice.Jouanot@imag.fr
TDs/TPs: Clement Charpentier / Fabrice Jouanot

Support: http://imag-moodle.e.ujf-grenoble.fr
(code: M1INFO15)

Programme du cours

- Gestion des transactions
- Gestion des contraintes d'intégrités
  - BDs dynamiques & déclencheurs
  - Application à Oracle
- Optimisation de requêtes
  - Optimisation algébrique, optimisation du stockage et des structures d'accès
  - Application à Oracle: tuning
- Modèle de données avancées
  - BDs objets en bref
  - Relationnel Object (application à Oracle)
- Accès à une BD
  - JDBC
  - Framework de persistance: Hibernate ORM
- Bases de données distribuées
  - Introduction
  - conception

Références

- Database System concepts, H. F. korth
- Systèmes de bases de données, T. connolly & C. Begg
- « Bases de données » de Gardarin (PDF)
- Cours en ligne de S. Abiteboul
- etc.

1. Gestion des transactions

Comment garantir la cohérence et la fiabilité tout en favorisant les accès concurrents aux données ?

Introduction

- La propriété fondamentale d'un SGBD est de garantir l'intégrité des données.
- Mais autoriser les accès concurrents est nécessaire pour un SI.
- 3 fonctions importantes et liées faisant référence à la notion de transaction:
  - Garantir un état cohérent
  - Contrôler la concurrence
  - Reprise après panne.

Un exemple très simplifié

- On considère un système de gestion de stades basés sur trois tables :
  Stade(nomStade, NbPlaces),
  Match(Affiche, NomStade, Prix, PlacesPrises)
  Client(NoClient, Solde).
- et disposant d'une fonctionnalité de réservation de places pour un client.

Exemple: Réservation de places

```
Places (Client C, Match M, NbPlaces N)
begin
        Lire le match M (tuple de la table match)
        Lire le stade S (tuple de la table stade)

        if ( (S.nbplaces – M.PlacesPrises) >= N )
        begin
                Lire le client C (tuple de la table client)
                M.PlacesPrises += N
                C.solde -= N * M.Prix
                Ecrire le match M (tuple modifié de la table match)
                Ecrire le client C (tuple modifié de la table client)
        end
end
```


Exemple: scénario concurrent

- Soit l'histoire suivante représentant deux exécutions concurrentes de la fonction de réservation pour le même match et le même client :
  - T1=Places(C, M, 100)
  - T2=Places(C, M, 200)

T1	L(M)
T1	L(S)
T1	L(C)
T1	E(M)
T2	L(M)
T2	L(S)
T2	L(C)
T2	E(M)
T1	E(C)
T2	E(C)

- Si on suppose que le stade dispose de 1000 places, sans résa pour le match au début de l'exécution: Quel est l'état de la base à la fin ?

Trace de l'histoire

T1 L(M) => 0 places réservées
T1 L(S) => 1000 places dans le stade
T1 L(C) => soit X le solde du client
T1 E(M) => 100 places réservées par le client
T2 L(M) => 100 places réservées
T2 L(S) => 1000 places dans le stade
T2 L(C) => soit X le solde du client
T2 E(M) => 300 places réservées par le client
T1 E(C) => X=X – 100*prix
T2 E(C) => X=X – 200*prix

- 300 places ont été réservées, mais 200 ont été comptabilisées au client !

Notion de transaction

- **Une transaction est une opération ou une suite d'opérations qui lit ou met à jour le contenu de la base de données.**
  - C'est une unité logique de travail
  - Formé par un programme, une partie de programme ou une commande

Ex: une suite de commande SQL peut former une transaction composée de plusieurs opérations.

2 opérations	1 opération
Select count(*) from T1	
	Update T1
	Set attr=select count(*) from T2
Select count(*) from T1	

Fonctionnement d'une transaction

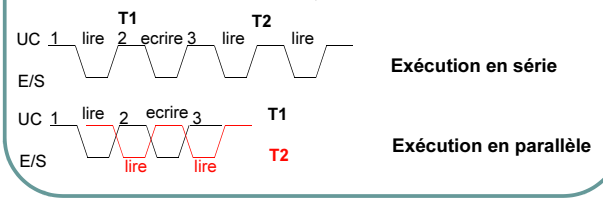
- Une transaction est constituée de 2 types d'opérations:
  - Lecture, notée lire(x) ou L(x) ou R(x)
  - Ecriture, notée ecrire(x) ou E(x) ou W(x)
- Une transaction possède 2 fins possibles
  - Achèvement avec succès, elle est validée ou confirmée (committed)
  - En échec, elle est annulée (aborted) car la BD doit rester dans un état cohérent et stable. L'annulation consiste en une opération de "roll back" pour remonter dans le dernier état cohérent.

Notion de concurrence

- La concurrence d'accès dans un environnement multi-utilisateurs permet une meilleure utilisation des ressources.
  - Simultanéité des accès
  - Mais compétition sur les données
- Les opérations d'E/S sont interfoliées pour assurer un accès concurrent aux données.

Exécution série vs. parallèle

T1:	T2:
| 1	|
Lire(x)	Lire(x)
| 2	|
Ecrire(x)	Lire(y)
| 3	|



Exécution en série

Exécution en parallèle

Problèmes liés à la concurrence

- L'entrelacement (interfoliage) des opérations de plusieurs transactions est un **ordonnancement** lorsque l'ordre des opérations de chaque transaction en concurrence est préservée.
- Des transactions a priori correctes peuvent produire des résultats incohérents en fonction de l'ordonnancement choisi.
- 3 problèmes peuvent apparaître:
  - Perte de mise à jour (écriture sale)
  - Lecture sale
  - Lecture non reproductible (et pare extension fantôme)

Perte de mise à jour (O1)

T1	T2
Début transaction	Début transaction
lire(solde)	lire(solde)
solde=solde-10	solde=solde+100
ecrire(solde)	ecrire(solde)
Validation	validation

- Une opération apparemment validée se trouve écrasée par une autre.
  - T1 et T2 démarrent presque au même temps
  - T1 et T2 lisent la même valeur de solde
  - La dernière écriture de T1 écrase celle de T2

solde=solde+100 est perdue

Lecture sale (O2)

T1	T2
	Début transaction
	lire(solde)
	solde=solde+100
Début transaction	ecrire(solde)
lire(solde)	…
solde=solde-10	Annulation
ecrire(solde)	
Validation	

- Une transaction lit les données écrites par une autre transaction non encore validée (O1 => O2)
  - T2 modifie la valeur de solde
  - T1 lit la valeur de solde modifiée et réalise sa tâche en se basant sur cette valeur
  - T2 annule entre-temps ces propres opérations.

solde=solde+100-10

Lecture non reproductible (O3)

T1	T2
Début transaction	Début transaction
lire(solde)	lire(solde)
solde=solde-10	solde=solde+100
…	ecrire(solde)
lire(solde)	validation
…	
Validation	

- Une transaction relit une valeur lu précédemment mais modifiée, entre-temps, par une autre transaction.
  - T1 lit la valeur de solde
  - T2 modifie la valeur de solde et valide
  - T1 relit la valeur de solde qui est différente de sa première lecture.

solde=100

Propriétés ACID assurées par un système de gestion des transactions

- **Atomicité**: "tout ou rien". Une transaction forme une entité indivisible, soit exécutée, soit annulée dans sa totalité.
- **Cohérence**: une transaction fait passer la BD d'un état cohérent à un autre état cohérent. La responsabilité de cohérence est cependant partagée:
  - Le SGBD vérifie les CI de schéma, et certaines CI métiers (déclencheurs des BD dynamiques)
  - Le développeur doit assurer la cohérence pour les CI d'application (créditer un compte erroné).

Propriétés ACID assurées par un système de gestion des transactions

- **Isolation**: les transactions s'exécutent de manière indépendante les unes des autres. Les effets des transactions incomplètes ne doivent pas être visibles par les autres transactions.
- **Durabilité**: les effets d'une transaction complètement achevée et validée sont inscrits de manière durable dans la BD. Ils ne peuvent être altérés par une défaillance.

Un SGBD possède donc un système de contrôle de concurrence et de reprise après panne capable de respecter ses propriétés + une gestion avancée des CI.

Gestion de transactions sous Oracle

- Valider une transaction: la commande **Commit** indique la fin d'une transaction et le début d'une autre. Une transaction est l'ensemble des opérations entre 2 commit.
- Annuler une transaction: la commande **RollBack** permet d'annuler toutes les opérations d'une transaction en remontant jusqu'au dernier point de sauvegarde. En l'absence de point de sauvegarde, la restauration remonte jusqu'au dernier commit.
  - savepoint Nom_du_point_de_sauvegarde
- Mode AutoCommit: Oracle est par défaut en mode **autocommit**, toutes les opérations sont systématiquement validées. Aucune transaction n'est possible dans ce mode (1 transaction = 1 opération).
  - Set autocommit on/off pour basculer d'un mode à l'aute

Ordonnancement et sérialisation

- Un ordonnancement est une séquence d'opérations d'un ensemble de transactions concurrentes qui préserve l'ordre des opérations dans chacune des transactions.
- **Un ordonnancement sériel** exécute les opérations de chaque transaction de manière consécutive, sans aucune opération interfoliée d'autres transactions.

T1	T2
	Début transaction
	lire(solde)
	solde=solde+100
	ecrire(solde)
	Validation
Début transaction	
lire(solde)	
solde=solde-10	
ecrire(solde)	
Validation	

Capacité de sérialisation

- Solution simple pour éviter les interférences (conflits) entre transactions: sérialiser les transactions.
  - Chaque transaction attend que la précédente ait validé pour commencer.
  - Cela contredit toute forme de simultanéité et de parallélisme.
- La capacité de sérialisation d'un ordonnancement reste donc intéressante
  - Si on trouve des ordonnancements non sériels dont le résultat est équivalent à une exécution sériel,
  - L'ordonnancement est dit cohérent.

Ordonnancement cohérent

- Les ordonnancements O1, O2 et O3 précédents ne sont pas sérialisables:
  - Les résultats produits ne sont équivalents ni à T1 puis T2, ni T2 puis T1.
  - Ces ordonnancements sont incohérents !

T1	T2
Début transaction	
lire(solde)	
solde=solde-10	
ecrire(solde)	
…	
lire(solde)	
Validation	
	Début transaction
	…
	lire(solde)
	solde=solde+100
	ecrire(solde)
	Validation

Conflits de sérialisation

- Le type et l'ordre des opérations influe directement sur la capacité de sérialisation:
  - 2 transactions qui lisent uniquement des données ne sont jamais en conflits
  - 2 transactions qui lisent et/ou écrivent des données différentes ne sont jamais en conflits
  - 2 transactions qui lisent et/ou écrivent des données dont un sous-ensemble est commun peuvent entrer en conflits.
- Une transaction Ti peut être en conflit avec Tj si:
  - Soit Lec(T) l'ensemble des objets lus par T et Ecr(T) l'ensemble des objets écrits par T, alors
  - $Ecr(Ti) \cap Lec(Tj) \neq \emptyset$
  - Ou $Ecr(Ti) \cap Ecr(Tj) \neq \emptyset$
  - Ou $Lec(Ti) \cap Ecr(Tj) \neq \emptyset$
- Comment identifier un conflit ?

Graphe de dépendance

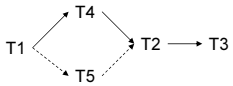
- Un graphe de dépendance permet de tester la capacité de sérialisation d'un ordonnancement: il décrit l'ordre des dépendances entre les opérations.
- Un graphe de dépendance est un graphe orienté (N,A) avec N un ensemble de nœuds et A un ensemble d'arcs dirigés:
  - Créer un nœud pour chaque transaction
  - Créer un arc Ti → Tj, si Tj lit la valeur d'un élément écrit par Ti
  - Créer un arc Ti → Tj, si Tj écrit une valeur d'un élément après qu'il a été lu par Ti
  - Créer un arc Ti → Tj, si Tj écrit une valeur d'un élément après qu'il a été écrit par Ti
  - Un arc est étiqueté avec la donnée lue/écrite (la cause de la dépendance).

  Aucun arc n'est ajouté si Tj lit la valeur d'un élément lu par Ti.

Analyser un graphe de dépendance

- Si une flèche Ti→Tj existe dans le graphe d'un ordonnancement O, alors dans tout ordonnancement sériel O' équivalent à O, Ti doit apparaître avant Tj.
- Un graphe de dépendance peut contenir plusieurs ordonnancements sériels O' équivalent à l'ordonnancement O de départ.

T1: Ecrire(A = 100)
Valider T1
T4 : Lire(A)
T5 : Lire(A)
Valider T5
Valider T4
T2 :Ecrire(A = 50)
Valider T2
T3 : Ecrire(A = 200)
Valider T3

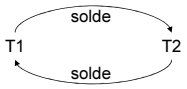


- Le graphe montre deux chemins pour aller de T1 à T2
  - Les positions T4 et T5 sont interchangeables
  - Équivalent à 2 ordonnancements sériels T1, T4, T5, T2, T3
    ou T1, T5, T4, T2, T3

Graphe de dépendance cyclique

- Si le graphe de dépendance montre la présence d'un cycle, l'ordonnancement n'a pas la capacité de sérialisation. Il est dit incohérent et des transactions sont en conflits.
- Si on considère O1 par exemple:

T1	T2
	Début transaction
Début transaction	
lire(solde)	
	lire(solde)
solde=solde-10	
	solde=solde+100
ecrire(solde)	
	ecrire(solde)
	validation
Validation	

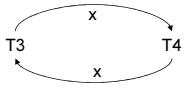


O1 n'est pas cohérent, il n'est équivalent à aucun ordonnancement sériel (ni T1 puis T2, ni T2 puis T1)

Problème des écritures non contraintes Limite du graphe de dépendance

- **Si une transaction écrit un objet sans l'avoir lu au préalable, il n'existe pas d'algorithme efficace pour déduire la capacité de sérialisation.**
- Prenons un exemple simple de deux transactions T3 et T4:

T3	T4
Lire(x)	
	Ecrire(x)
Ecrire(x)	



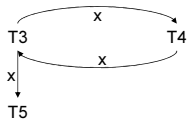
- Le graphe de dépendance est cyclique, il trouve le conflit.
- Le résultat n'est équivalent ni à T3 puis T4, ni T4 puis T3, donc l'ordonnancement n'est pas cohérent.

On croit que ça marche dans tous les cas…

Problème des écritures non contraintes

- Ajoutons une transaction T5:

T3	T4	T5
Lire(x)		
	Ecrire(x)	
Ecrire(x)		
		Ecrire(x)



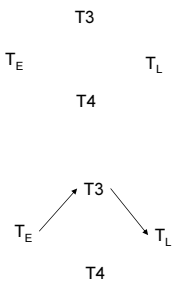
- Le graphe de dépendance est cyclique, donc l'ordonnancement n'a pas de capacité de sérialisation a priori.
- Cependant l'arc T4→T3 ne devrait pas être ajouté au graphe
  - Les écritures produites par T3 et T4 ne sont utilisées par aucune transaction: elles sont inutiles et remplacée par celle de T5
  - **L'ordonnancement sériel équivalent est T3, T4, T5**

Besoin d'un nouveau modèle de graphe

- Soit l'ordonnancement O et O' son équivalent sériel, avec Tj qui lit un objet précédemment écrit par Ti:
  - Si O est sérialisable alors Ti précède Tj dans O'
  - Si ∃Tk qui effectue une écriture dans O, alors Tk est avant Ti ou après Tj, sinon Tj ne lit pas la valeur écrite par Ti et ¬(O ≡ O')
- Il est impossible d'exprimer la possibilité d'ajouter ou non un arc avec un graphe de dépendance…
- Il faut l'étendre à un graphe de dépendance labellisé (ce label ne doit pas être confondue avec la donnée lue/écrite).

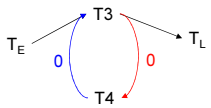
Graphe de dépendance labellisé 1/2

1. Créer un nœud pour chaque transaction
2. Créer un nœud T_E pour une transaction factice en début d'ordonnancement contenant une opération d'écriture pour chaque donnée accédée
3. Créer un nœud T_L pour une transaction factice en fin d'ordonnancement contenant une opération de lecture pour chaque donnée accédée
4. Créer un arc Ti $\rightarrow^0$ Tj si Tj lit la valeur d'une donnée écrite par Ti
5. Supprimer tous les arcs dirigés vers Tj pour lesquels il n'existe pas de chemin de Ti vers T_L



Graphe de dépendance labellisé 2/2

6. Pour chaque donnée lu par Tj, qui a été écrite par Ti, et que Tk écrit (Tk ≠ T_E):
   a) Si Ti=T_E et Tj ≠ T_L, alors Tj $\rightarrow^0$ Tk
   b) Si Ti ≠ T_E et Tj=T_L, alors Tk $\rightarrow^0$ Ti
   c) Si Ti ≠ TE et Tj ≠ TL, alors créer deux arcs Tk $\rightarrow^x$ Ti et Tj $\rightarrow^x$ Tk où x un entier positif encore non utilisé dans le graphe.



On identifie un cyle, donc a priori le graphe est incohérent ?

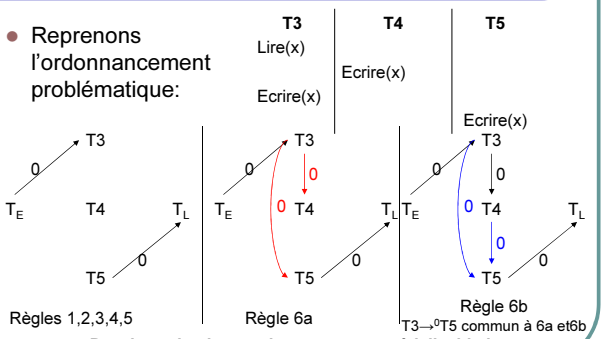
- Le cas 6.c généralise les règles précédentes: si Ti écrit une donnée que Tj lira, alors Tk qui écrit la même donnée doit soit précéder Ti, soit succéder Tj.

Analyse du graphe de dépendance labellisé

- Si le graphe ne contient aucun cycle, l'ordonnancement est sérialisable.
- Si le graphe possède un cycle, on ne peut conclure directement:
  - Si le graphe est étiqueté uniquement avec des arcs →0, alors l'ordonnancement n'est pas sérialisable
  - La génération d'une paire d'arcs par la règle 6c produit 2 graphes possibles. Si au moins l'un des graphes est acyclique: l'ordonnancement est sérialisable.
  - Si m paires d'arcs sont générées par la règle 6c alors il faut étudier 2^m graphes.

Exemple de graphe labellisé

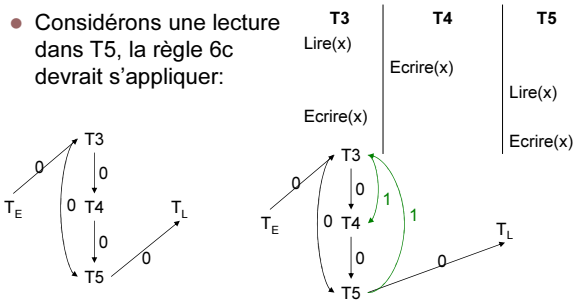
- Reprenons l'ordonnancement problématique:



Pas de cycle, donc ordonnancement sérialisable !

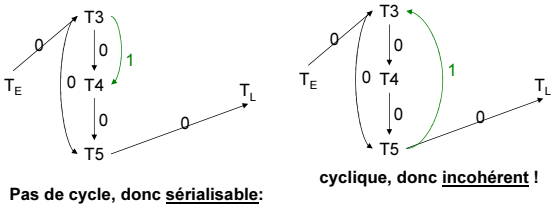
Exemple complexe de graphe labellisé 1/2

- Considérons une lecture dans T5, la règle 6c devrait s'appliquer:



Exemple complexe de graphe labellisé 1/2

- Nous obtenons un graphe pour chaque arc de la paire labellisé 1:



Pas de cycle, donc sérialisable:
T3, T4, T5

cyclique, donc incohérent !

- Au moins 1 graphe est sérialisable, donc l'ordonnancement est cohérent.

Objectif: produire des ordonnancements sérialisables

- Un SGBD ne teste pas la capacité de sérialisation d'un ordonnancement: trop coûteux !
- Des protocoles permettent d'assurer l'obtention d'ordonnancement sérialisable:
  - Protocoles de verrouillages
  - Protocoles d'estampillage
  - Méthodes optimistes

Méthodes de verrouillage

- Lorsqu'une transaction accède à une BD, un verrou est susceptible de bloquer l'accès à d'autres transactions.
- On distingue 2 types de verrous:
  - Partagé: si une transaction dispose d'un verrou partagé sur une donnée, elle peut la lire mais pas la modifier = verrou en lecture (VL).
    Plusieurs transactions peuvent se partager un même verrou en lecture.
  - Exclusif: si une transaction dispose d'un verrou exclusif sur une donnée, elle peut la lire et la modifier = verrou en écriture (VE).
    Une seule transaction peut obtenir un verrou Exclusif, les autres transactions doivent attendre la libération du verrou pour accéder à la donnée (en posant un nouveau verrou).

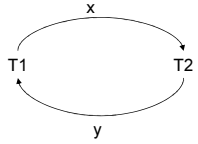
Protocole de verrouillage

- Toute transaction devant accéder à une donnée doit obtenir un verrou sur la donnée: soit VE(x) , soit VL(x)
- Si la donnée est déjà verrouillée par une autre transaction avec un verrou non compatible (seul VL est compatible avec VL) la transaction doit attendre la libération du verrou.
- Une transaction conserve un verrou tant qu'elle ne le libère pas:
  - Explicitement avec une commande de libération notée L(x).
  - Implicitement avec une terminaison de transaction: validation (commit) ou annulation (abort).
- **Insuffisant pour garantir la sérialisation des ordonnancements…**

Exemple d'ordonnancement incohérent par verrouillage

T1	T2
VE(x)	
Lire(x)	
x=x+100	
Ecrire(x)	
L(x)	
	VE(x)
	Lire(x)
	x=x*1.1
	Ecrire(x)
	L(x)
	VE(y)
	Lire(y)
	y=y*1.1
	Ecrire(y)
	L(y)
	Valider
VE(y)	
Lire(y)	
y=y-100	
Ecrire(y)	
L(y)	
Valider	

- Initialement x=100, y=400
  - T1, puis T2 donne x=220 et y=330
  - T2, puis T1 donne x=210 et y=340
- L'ordonnancement donne x=220 et y=340: donc **PAS SERIALISABLE**



Verrouillage à deux phases

- Pour garantir la sérialisation d'un ordonnancement on ajoute un protocole pour contrôler le verrouillage: V2P ou 2PL (2 phase-locking)
- Une transaction suit le protocole de verrouillage à deux phases si toutes les opérations de verrouillage précédent la première opération de déverrouillage de cette transaction.
  - Phase 1: la transaction acquière des verrous
  - Phase 2: la transaction libère les verrous.
  - Une transaction qui libère un verrou ne peut plus en acquérir.
- **Si dans un ordonnancement toutes les transactions respectent le protocole V2P alors cet ordonnancement est sérialisable.**

Résolution de la MAJ perdue

T1	T2
	Début transaction
Début transaction	VE(solde)
VE(solde)	lire(solde)
wait	solde=solde+100
wait	ecrire(solde)
wait	VL(prime)
wait	si (prime>500) …
wait	Validation (L(solde), L(prime))
lire(solde)	
solde=solde-10	
ecrire(solde)	
Validation (L(solde))	

- T1 et T2 en V2P
  - T1 attend que T2 relache son verrou sur la donnée
  - On retombe sur une exécution sérielle (T2 puis T1).

Résolution de la lecture sale

T1	T2
	Début transaction
	VE(solde)
	lire(solde)
Début transaction	solde=solde+100
VE(solde)	ecrire(solde)
wait	…
wait	Annulation (L(solde))
lire(solde)	
solde=solde-10	
ecrire(solde)	
Validation	

- T1 et T2 en V2P
  - T1 attend que T2 relache son verrou sur la donnée
  - On retombe sur l'exécution de T1.

Problème des annulations en cascade 1/2

T1	T2	T3
Début transaction		
VE(x)		
lire(x)		
VE(y)		
lire(y)		
x=y-x		
ecrire(x)		
L(x)	Début transaction	
…	VE(x)	
	lire(x)	
	x=x+100	
	ecrire(x)	
	L(x)	
Annulation	…	
		Début transaction
	Annulation	VE(x)
		lire(x)
		…
		Annulation

Problème des annulations en cascade 2/2

- Dépendances des transactions
  - T1 a écrit x avant de libérer son verrou,
  - T2 récupère la donnée de x modifié par T1, et écrit une valeur modifiée de x avant de libérer son verrou sur x,
  - T3 lit la valeur modifiée par T2 après avoir posé son verrou
- Si T1 est annulée
  - T2 dépend de T1 donc T2 doit être annulée
  - T3 dépend de T2 donc T3 doit être annulée
- Risque d'annulation en cascade: destruction important du travail accompli.
- Solution: repousser les libérations de verrous à la fin de la transaction
  - C'est le verrouillage rigoureux en deux phases
  - On peut aussi retarder uniquement les verrous exclusifs, c'est le verrouillage strict en deux phases.

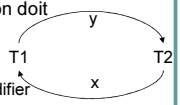
Problème des interblocages

T1	T2
Début transaction	
VE(x)	
lire(x)	
x=x-10	
ecrire(x)	
VE(y)	Début transaction
wait	VE(y)
wait	lire(y)
…	y=y+100
	ecrire(y)
	VE(x)
	wait
	wait
	…

- T1 attends que T2 libère y et T2 attends que T1 libère x
- **Stituation de blocage ou interblocage** (deadlock): impasse générée par deux transactions (ou plus) qui attendent mutuellement que des verrous se libèrent.
- Résolution simple: délai imparti pour l'acquisition d'un verrou
  - Une transaction a un délai maximum pour obtenir un verrou
  - Sinon la transaction est annulée.

Résolution de l'interblocage

- La solution générale est la construction d'un graphe d'attentes:
  - Un nœud par transaction
  - Un arc Ti → Tj est ajouté si la transaction Ti attend pour verrouiller un élément déjà verrouillé par Tj.
- Si un cycle est détecté: au moins une transaction doit être annulée
  - Soit la plus récente,
  - Soit celle ayant modifié le moins de données
  - Soit celle ayant encore le plus de données à modifier
- Un SGBD choisit la méthode la moins couteuse
  - En terme d'informations à conserver pour choisir sa victime
  - En terme d'annulation à réaliser
  - Il évite la famine, c'est-à-dire ne pas choisir toujours la même victime.



Stratégie de prévention d'interblocages

- Deux transactions Ta et Tb, on considère que Ta veut accéder à une donnée verrouillée par Tb:
  - Wait-die (attendre - mourir)
    - Si Ta plus vieille que Tb, alors Ta est mise en attente et Tb finit (commit ou rollback), sinon Ta est tuée (rollback).
    - Les transactions jeunes sont prioritaires tant qu'elles ne sont pas gênées par des plus vieilles.
    - Les transactions plus vieilles peuvent attendre longtemps
  - Wound-wait (blesser- attendre)
    - Si Ta plus vieille que Tb alors Ta tue Tb (rollback) et s'empare de sa ressource, sinon Ta attend.
    - Les transactions les plus vieilles sont exécutées rapidement (et les jeunes attendent).

Autre technique: Estampillage

- Verrouillage très employé dans les SGBD centralisés mais peu adaptés au SGBD répartis sur plusieurs sites (1 SGBD par sites qui forme globalement un système réparti).
- Les approches distribuées utilisent une technique à base d'**estampilles** pour gérer les transactions.
  - Une estampille est un identificateur temporel permettant d'ordonner des objets chronologiquement.
  - Le verrouillage assure qu'**un** ordonnancement soit équivalent à un ordonnancement sériel
  - L'estampillage assure qu'un ordonnancement soit équivalent à **l'**ordonnancement sériel qui correspond à l'ordre chronologique des estampilles des transactions.

Principe de l'estampillage

- Gestion de la concurrence
  - Toutes les transactions s'exécutent simultanément (pas de verrou)
  - Un conflit = une transaction demande à lire un élément qui a déjà été écrit par une transaction plus récente **ou** une transaction demande à écrire un élément qui a déjà été lu ou mis à jour par une transaction plus récente.
  - En cas de conflit: rejets des demandes conflictuelles sinon l'exécution simultanée ne serait plus équivalente à l'exécution chronologiques des transactions.
  - Solution: tuer la transaction "trop vieille" qui fait la demande et la relancer avec une nouvelle estampille.
- Tuer une transaction
  - Revient à défaire ou annuler une transaction, donc risque d'annulation en cascade.
  - Effectuer physiquement les écriture lors de la fin normale (COMMIT) de la transaction: rien à faire en cas d'annulation (ROLLBACK)

Deux types d'estampilles

- De transaction, noté e(T): identificateur unique créé par le SGBD et précisant le moment de démarrage d'une transaction (implémenté sous la forme d'un compteur logique)
- De donnée où chaque donnée possède deux estampilles:
  - LMAX qui est l'estampille de la transaction la plus jeune qui a lu cet élément (sans avoir été tuée lors de cette lecture),
  - EMAX qui est l'estampille de la transaction la plus jeune qui a écrit avec succès l'élément (sans avoir été tuée lors du COMMIT).
  - Le système ordonne l'exécution des transactions en comparant ces deux valeurs pour tout élément lu ou écrit avec l'estampille de la transaction.

Protocole de gestion des conflits avec estampilles

- soit une transaction T qui demande à lire un élément E

si e (T) ≥ EMAX(E)
  alors **/*OK, la lecture est effectuée */**
    LMAX(E):=Max(LMAX(E), e (T))
  sinon /*conflit */
    Relancer la transaction T avec une nouvelle estampille.

- soit une transaction T qui demande à écrire un élément E

si e(T) ≥ LMAX(E) et e(T) ≥ EMAX(E)
  alors **/*OK, la mise à jour est effectuée */**
    EMAX(E):=e (T)
  sinon **/*conflit */**
    Relancer la transaction T avec une nouvelle estampille.

Exemple d'estampillage

T1	T2
Lire(B)	
	Lire(B)
	B=B-50
	Ecrire(B)
Lire(A)	
	Lire(A)
Afficher(A+B)	
	A=A+50
	Ecrire(A)
	Afficher(A+B)

- On suppose e(T1)<e(T2)
- Montrer que cet ordonnancement est valide…

Exemple d'estampillage

- Démonstration:
  e(T1)>=EMAX(B) [première écriture, EMAX(B)=0] donc LMAX(B):=Max(LMAX(B),e(T1)):=Max(0,e(T1)):=e(T1)
  e(T2)>=EMax(B) (première écriture, EMAX(B)=0) donc LMAX(B):=Max(LMAX(B), e(T2):=Max(e(T1), e(T2)):=e(T2)
  e(T2)>=LMAX(B) [e(T2)] et e(T2)>=EMAX(B) [0] donc EMAX(B):=e(T2)

  e(T1)>=EMAX(A) donc LMAX(A):=e(T1)
  e(T2)>=EMAX(A) donc LMAX(A):=e(T2)

  e(T2)>=LMAX(A) donc EMAX(A):=e(T2)

Approche Optimiste

- Réservé aux cas où les conflits sont rares: beaucoup de lecture et très peu d'écritures
  - Vérification d'un conflit lors de la validation d'une transaction
  - Annulation si conflit, donc peu être très coûteux si risque de conflits important !
- Protocole optimiste = 2 ou 3 phases
  1. Lecture (jusqu'à validation): les MAJ de données sont appliquées sur des copies locales
  2. Validation: détection des conflits (estampille par exemple)
  3. Écriture: copies locales recopiées sur la base.

Gestion de transactions dans un SGBD

- Elle repose sur 2 commandes
  - Validation = commande Commit
  - Annulation = commande Rollback
- Deux types de transaction
  - Read only
  - Read write
- Quatre niveaux d'isolation sont définies pour relâcher ou consolider les Pbs de lecture sale, lecture non reproductible et lecture fantôme
  - Chaque transaction peut choisir son niveau d'isolation
  - Le niveau d'isolation modifie la visibilité des écritures des autres et de ses propres écritures par les autres.

Niveaux d'isolation 1/2

- Serializable
  - La transaction suit le protocole 2PL strict
  - La transaction ne voit que les modifications validées au début de son exécution et ses propres modifications => synchronisation nécessaire à sa validation.
  - Une même requête (ré-exécutée) donnera le même résultat car les modifications sont bloquées pour les autres transactions.
- Repeatable read
  - Une même requête donnera le même résultat, mais les autres transactions peuvent modifier les mêmes données (non bloquées)
  - Fantômes possibles

Niveaux d'isolation 2/2

- Read commited
  - Chaque requête de la transaction ne voit que les modifications validées avant son exécution.
  - L'état de la base peut changer entre deux requêtes.
  - fantômes et lecture non reproductibles possibles
  - Ecriture sale possible
- Read uncommitted
  - La transaction peut lire des objets écrits par une autre transaction (pas de verrous en lecture)
  - lectures sales possibles
  - Proche du mode read only d'oracle.

Niveaux d'isolation: résumé

	Lecture sale	Lecture non reproductible	fantôme
Read Uncommited	Oui	Oui	Oui
Read commited	Non (mais ecriture sale)	Oui	Oui
Repeatable Read	Non	Non	Oui
Serializable	Non	Non	Non

Exemple d'isolation

- Voir Document Resa-spectacle
  - Description du schéma
  - Description des transactions (SQL)
- Trouver pour chaque niveau d'isolation un ordonnancement entre deux transactions qui démontrent une incohérence:
  - READ UNCOMMITTED => Lecture sale
  - READ COMMITTED => Ecriture sale
  - REPEATABLE READ => fantôme
  - SERIALIZABLE => Dead lock

Niveau d'isolation sous Oracle

- Deux modes pour Oracle:
  - `Set transaction isolation level read committed`, chaque instruction au sein de la transaction ne voit que les données validées.
  - `Set transaction isolation level serializable`, chaque instruction au sein de la transaction ne voit que les données validées avant le début de la transaction. ATTENTION, une transaction "sérialisable" peut être annulé par le système si elle ne peut être sérialisé !
  - Des verrous niveau enregistrements sont mis en œuvre.
- Un troisième mode: `set transaction read only`, définit une transaction en lecture seule qui voit les données validées avant le début de la transaction (et interdit les modifications)

Gestion des verrous sous Oracle

- Oracle dispose deux types de verrous
  - Logique sur les tables ou les lignes
  - Physique sur les segments, fichiers et pages.
- Au niveau logique, on trouve:
  - Verrous lignes: seules les lignes à modifier d'une table que modifie une transaction sont verrouillées.
  - Verrous tables: toutes les lignes de la table que modifie une transaction sont verrouillées.
  - Verrous dictionnaire: protègent la description des structures de données (schéma).
  - Verrous internes (loquets): protègent les objets systèmes.
  - Etc.
- Oracle intercepte les interblocages et annule l'ordre à la source de celui-ci (erreur !).

Manipulation des verrous sous Oracle

L'utilisateur a la possibilité de définir sa propre stratégie en conformité avec le verrouillage implicite géré par le SGBD.

`LOCK TABLE {nom_table|nom_vue IN`

- **`ROW SHARE`** verrou (RS), les lignes à modifier sont verrouillées et les autres transactions peuvent mettre à jour les données non concernées par le verrou.
- **`ROW EXCLUSIVE`** verrou (RX), aucune transaction ne peut demander de verrous (S) ou (X, SRX)
- **`SHARE UPDATE`** verrou (RS)
- SHARE verrou partagé (S), les autres transactions peuvent poser des verrous (S) sur la ressource mais pas exclusif (X ou SRX). Si plusieurs transactions ont posé un verrou S, aucune ne peut faire de modification.
- **`SHARE ROW EXCLUSIVE`** verrou (SRX), les autres transactions peuvent obtenir des verrous (RS)
- **`EXCLUSIVE`** verrou exclusif (X), toutes les lignes de la tables sont verrouillées, la consultation reste possible pour les autres transactions mais ne peuvent poser de verrous.

`MODE}`

`[NOWAIT]` empêche d'attendre si la ressource n'est pas disponible (rend la main à l'utilisateur).

Résumé de la compatibilité des verrous Oracle

Verrou obtenu	Verrou **impossibles** pour les autres	Verrou **possibles** pour les autres
RS	X	RX, RX, S, SRX
RX	S, SRX, X	RX, RS
S	RX, SRX, X	RS, S
SRX	RX, S, SRX, X	RS
X	RS, RX, S, SRX, X	-

Fiabilité d'un SGBD

Comment gérer la reprise après panne d'un SGBD pour garantir la cohérence des BDs ?

Panne et durabilité

- Pour chaque transaction un SGBD doit assurer
  - Que les effets d'une transaction validée persiste dans la base,
  - Qu'une transaction non validée n'ait d'effet ni sur la base ni sur les autres transactions.
- Un SGBD doit inclure des mécanismes pour éviter les erreurs et restaurer un état correct après une panne.
- Nombreuses causes d'erreurs: programmes, système, humaines, hardware, externe…

Définition de la fiabilité

- Capacité à restaurer la base de données dans un état connu comme correct après une erreur.
- Solution: la redondance d'informations
  - Backup périodique de la base,
  - Écriture dans un Journal des opérations sur la base
  - En cas de panne + base endommagée: restauration + utilisation du journal pour refaire les modifications depuis le backup
  - En cas de panne + base Ok: le contenu n'est plus fiable, on la restaure à l'aide du journal en défaisant les modifications.

Différentes catégories de panne

- Panne locale à une transaction
  - Détectée par le code de l'application: erreurs classiques suite à des conditions non vérifiées → Rollback
  - **Imprévue → Annulation par le système**
- **Panne du système impactant les transactions sans endommager la base → mise en œuvre de protocoles de reprise**
- Arrêt volontaire du système → attente de terminaison des transactions, puis arrêt.
- Panne endommageant la base
  - Perte de donnée → restauration + journal pour refaire les modifications.
  - Perte du journal → restauration ancien backup

Panne locale imprévue

- Le système doit défaire toutes les modifications de la transaction comme si elle n'avait jamais eu lieu.
- Utilisation du journal:
  - Écriture du début de transaction (BEGIN)
  - Écrire chaque modification, suppression, insertion avant la MAJ de la base (pourquoi ?)
  - Écriture de la fin de transaction (COMMIT/ROLLBACK)
- Procédure d'annulation: remonter le journal en arrière en replaçant les anciennes valeurs jusqu'au marqueur BEGIN !
- Défaire une transaction X fois = défaire 1 fois, sinon effet indésirable lors d'une panne pendant l'annulation.

Panne système → point de reprise

- Panne système = redémarrage
  - Perte de la mémoire (transactions en cours, buffer d'E/S)
  - La base n'est pas endommagée
- Comment connaître
  - les transactions validées
  - Les transactions qui auraient pu valider (mais écritures des buffers incertains ?)
  - Les transactions qui n'ont pas terminé avant la panne.
- On introduit **un point de reprise** pour ordonner les différentes transactions par rapport à un état cohérent.

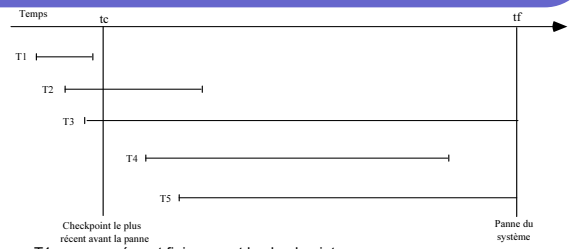
Définition du point de reprise

- Au point de reprise:
  - Les transactions sont suspendues
  - Écriture des buffers du journal
  - Écriture d'un point de reprise dans le journal (CHECKPOINT)
  - Écriture des buffers de données → Force-writing (les données en mémoire sont forcées sur le support persistant)
  - Écriture du point de reprise dans le fichier de redémarrage.
- Intervalle entre point de reprise
  - Soit périodique (tous les m minutes),
  - Soit en fonction du nombre de transactions validées (depuis le dernier CHECKPOINT)

Redémarrage avec point de reprise

- Le fichier de redémarrage indique le point de reprise le plus récent,
- Le journal est lu à partir du checkpoint
  - Les transactions à défaire → en remontant le journal jusqu'au checkpoint
  - Les transactions à refaire → en parcourant le journal à partir du checkpoint
- 5 cas de transactions sont considérés par le gestionnaire de reprise.

Différents cas de transactions



- T1 commencées et finies avant le checkpoint;
- T2 commencées avant le checkpoint et finies après (mais avant la panne);
- T3 commencées avant le checkpoint et toujours en cours lors de la panne;
- T4 commencées et finies après le checkpoint;
- T5 commencées après le checkpoint et toujours en cours lors de la panne.

Méthode générale de reprise

- Transaction non considérée: T1
  - Les modifications de T1 ont déjà été forcées sur la base.
- Transaction à défaire: T3 et T5
  - Les transactions ne sont pas terminées et la fin n'est pas connue
  - Elles ne doivent jamais avoir existé
- Transaction à refaire: T2 et T4
  - Elles ont validé après le checkpoint
  - Aucune assurance que les buffers ont été écrits sur la base
  - Pour T2, seuls les changements après le checkpoint sont à refaire.
  - Refaire une transaction X fois = refaire 1 fois !
- Mode d'écriture du journal *Write-ahead*:l'écriture sur la base se fait après l'écriture sur le journal !

Protocole Immediate Update

- C'est le protocole d'écriture considéré précédemment: le buffer d'une transaction est écrit dés qu'il est plein… sans attendre COMMIT.
- La méthode Write-ahead protège des MAJs impossible à défaire
- Problème : risque de défaire en cascade
  - On doit défaire T,
  - S lit une valeur X écrite par T → on doit aussi défaire S
  - R lit une valeur Y écrite par S → on doit aussi défaire R

Gestion du journal en mise à jour immédiate

1. Commencer à partir du dernier élément dans le journal et lire en arrière. Faire deux listes de transactions
   - LR celles qui ont fait COMMIT après le dernier checkpoint
   - LD1 celles qui n'ont pas fait COMMIT.
2. Faire une liste LD2 des transactions qui ont lu des valeurs écrites par les transactions de LD1.
   - Appliquer cette étape récursivement à toute transaction de LR
   - pour faire une liste LD2 des transactions qui doivent être défaites.
3. Défaire toutes les opérations WRITE des transactions de LD1 et LD2 dans l'ordre inverse du journal.
4. Refaire les opérations WRITE des transactions de LR dans l'ordre d'écriture dans le journal.

Application Immediate Update

T0: A=1000, B=2000	T1: C=700
Read(A)	Read(C)
A=A-50	C=C-100
Write(A)	Write(C)
Read(B)	
B=B+50	
Write(B)	

Exécution sérielle cohérent (T0, T1) sans panne

<T0, begin>	
<T0, A, 1000, 950>	
<checkpoint>	A=950
<T0, B, 2000, 2050>	
<T0, commit>	B=2050
<T1, begin>	
<T1, C, 700, 600>	
<T1, commit>	C=600

- Panne 1 après write(B)
  - A=1000, B=2000, C=700
- Panne 2 après write(C)
  - A=950, B=2050, C=700
- Panne 3 après <T1,commit>
  - A=950, B=2050, C=600

Protocole Deffered Update

- Les mises à jour sur la base de données sont effectuées physiquement seulement quand la transaction arrive au COMMIT.
- Les mises à jour sont mémorisées dans
  - le journal
  - l'espace de travail de la transaction.
- La transaction arrive au COMMIT
  - Le journal est forcé sur disque,
  - Les mises à jour sont mémorisées dans la base
- Avantage: si une transaction a une panne avant d'arriver au COMMIT → pas besoin de défaire les opérations, le journal est plus simple !

Gestion du journal en mise à jour différée

1. Remonter le journal jusqu'au checkpoint le plus récent. Faire deux listes de transactions:
   - LR celles qui ont fait COMMIT après le dernier checkpoint
   - LD celles qui n'ont pas fait COMMIT.
2. Refaire toutes les opérations WRITE des transactions LR dans l'ordre d'écriture dans le journal.
3. Les transactions de LD sont ignorées (rien à défaire).

- Inconvénient: cette méthode limite l'exécution concurrente car tous les éléments sont verrouillés jusqu'au COMMIT.

Application Deferred Update 1/2

T0: A=1000, B=2000	T1: C=700
Read(A)	Read(C)
A=A-50	C=C-100
Write(A)	Write(C)
Read(B)	
B=B+50	
Write(B)	

Exécution sérielle cohérent (T0, T1) sans panne

<T0, begin>	
<T0, A, 950>	
checkpoint	
<T0, B, 2050>	
<T0, commit>	A=950, B=2050
<T1, begin>	
<T1, C, 600>	
<T1, commit>	C=600

- Panne 1 après write(B)
  - A=1000, B=2000, C=700
- Panne 2 après write(C)
  - A=950, B=2050, C=700
- Panne 3 après <T1,commit>
  - A=950, B=2050, C=600
- Que se passe-t'il si un crache intervient pendant la reprise de la panne 2 ?
  - Rien, on peut refaire plusieurs fois = refaire 1 fois.

Connectivité d'une application à une BD (JDBC)

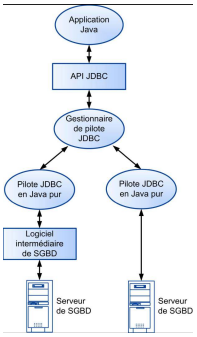
Comment interroger et modifier le contenu d'une BD dans un programme Java ?

Introduction JDBC

- JDBC (Java DataBase Connectivity) est une API Java utilisé pour se connecter à une BD
  - Le SGBD doit posséder une interface JDBC
  - La base doit être déclarée accessible (point de connexion)
- JDBC est composé de deux interfaces
  - Rédacteur d'applications
  - Rédacteur de pilotes
- C'est le cœur des API de plus haut niveau
  - + : accès standardisé au SGBD acceptant des fonctionnalités SQL
  - - : très proche du modèle relationnel, la persistance des objets d'applications n'est pas transparente → commandes SQL

Architecture JDBC

- Plusieurs types de pilotes
  - Pilote JDBC en Java pur + connexion directe (très performant)
  - Pilote JDBC en Java pur via middleware (déploiement sur SGBD distribués)
  - Pilote JDBC adHoc (conversion JDBC vers API client)
  - Pont JDBC-ODBC (peu performant)



Interfaces JDBC… de base

- DriverManager: gestion des pilotes disponibles
- Connection: établissement d'une connexion avec une base
- Statement: exécution d'une requête SQL à travers une connexion
- ResultSet: gestion des résultats d'une requête sous forme d'une liste d'éléments.
- SQLException: encapsule les erreurs lors des accès à la base.
- Canevas d'utilisation des interfaces:
  - importer le package java.sql (import java.sql.*)
  - Enregistrement du pilote JDBC correspondant,
  - Connexion au point d'entrée JDBC de la base,
  - Création d'un descripteur de requête,
  - Exécution d'une requête SQL,
  - Parcours de la liste des résultats et traitement par l'application,
  - Terminer (fermer) la connexion.

Exemple d'utilisation 1/5

- Soit la table Fichier(NumEtudiant, Nom, Prenom, Annee, Poursuite) :
  - NumEtudiant : numéro étudiant (clef primaire)
  - Nom : nom de l'étudiant (chaîne de caractère),
  - Prenom : prénom de l'étudiant (chaîne de caractère),
  - Annee : promotion (chaîne de caractère),
  - Poursuite : situation à la sortie de la formation (chaîne de caractère).
- La table est accessible sur la base ufrima de im2ag-oracle. Un spécialiste a déclaré cette base dans le SGBD via la chaine de connexion:

jdbc:oracle:thin:@im2ag-oracle.e.ujf-grenoble.fr:1521:ufrima

Exemple d'utilisation 2/5

- On commence par définir le pilote JDBC qui permet l'accès à la base.
  - On crée un objet du pilote correspondant ici Oracle.
  - On enregistre ensuite ce pilote dans le gestionnaire JDBC.

Ex : DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

- On peut ensuite se connecter à la BD :
  - On crée un descripteur sur la BD de type `Connection`,
  - On utilise la méthode `getConnection(String, String, String)`
    - 1er paramètre : chaine de connexion (à connaître),
    - 2eme paramètre : nom d'utilisateur
    - 3eme paramètre : mot de passe de l'utilisateur

```
Ex : Connection base = DriverManager.getConnection(
     "jdbc:oracle:thin:@im2ag-oracle.e.ujf-grenoble.fr:1521:ufrima", "jouanotf",
     "bd2009");
```


Exemple d'utilisation 3/5

- Pour poser une requête, il faut créer un descripteur de requête :
  - On crée un objet de type `Statement`,
  - On utilise la méthode `createStatement()` de l'objet de type `Connection`

Ex : Statement requete = base.createStatement();

- Pour envoyer une requête sur la base (sous la forme d'une chaine de caractère) et récupérer les résultats, on utilise la méthode `executeQuery(String )` de l'objet `Statement`

```
Ex : ResultSet resultat = requete.executeQuery(
          "SELECT Nom, Prenom, Poursuite " +
          "FROM fichier " +
          "WHERE Annee = '1997'");
```


Exemple d'utilisation 4/5

- Exploitation des résultats d'une requête
  - Les méthodes `next()` et `previous()` appelées sur un objet de type ResultSet permettent de parcourir les éléments de la liste.
  - La méthode `getString(String )` retourne la valeur de l'attribut dont le nom est passé en paramètre

```
Ex : while(resultat.next()) {
          System.out.println("Nom = " + resultat.getString("Nom")
          + ", Prenom = " + resultat.getString("Prenom")
          + ", Travail = " + resultat.getString("Poursuite"));
     }
```

- Libération des ressources à l'aide de la méthode `close()`.

Exemple d'utilisation 5/5

```
import java.sql.*;

public class Travail {
   public static void main(String[] arg) {
      try {
         DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver()); // Chargement du pilote
         Connection base = DriverManager.getConnection ("jdbc:oracle:thin:@hopper.e.ujf-grenoble.fr:1521:ufrima",
      "jouanotf", "bd2009"); // connexion
         Statement requete = base.createStatement(); // création du descripteur de requête
         ResultSet resultat = requete.executeQuery( // exécution d'une requête
                  "SELECT Nom, Prenom, Poursuite " +
                  "FROM fichier " +
                  "WHERE Annee = '1997'");
         while(resultat.next()) { // récupération des résultats
            System.out.println("Nom = " + resultat.getString("Nom")
                  + ", Prenom = " + resultat.getString("Prenom")
                  + ", Travail = " + resultat.getString("Poursuite"));
         }
         requete.close();
         resultat.close();
         base.close(); // fermeture de la connexion
      } catch (Exception err) { System.out.println("Une erreur, Oh Oh!"); } // Attention il faut capturer les
      exceptions !
   }
}
```


Gestion des transactions en JDBC

- Changer la gestion de l'auto-commit
  - Méthode `setAutoCommit(true | false)` de la classe `Connection`.
- Définir une transaction
  - Méthode `commit()` de `Connection`
  - Méthode `rollback()` de `Connection`
- Niveaux d'isolation
  - Méthode `setTransactionIsolation(READ_COMMITED | SERIALIZABLE)` de Connection
  - Méthode `getTransactionIsolation()` retourne le niveau d'isolation courant.

BD dynamique / Active

Laisser au SGBD la gestion des contraintes ou comment programmer des contraintes…

Plan

- Définitions
- Déclencheurs et règles
- Utilisation des règles
  - intégrité
  - vues
- Définir des règles sous Oracle
  - Introduction
  - Programmation en PL/SQL
  - Développement de triggers (déclencheurs)
  - Use case

Définitions

- SGBD passif
  - stockage de l'information
  - restitution de l'information
  - les activités de contrôle sont réalisées dans les programmes d'application
- SGBD actif
  - stockage et restitution de l'information
  - description des événements pertinents et des réactions à activer lorsque ces événements se produisent

Limitation SQL

- Gestion de contraintes simples dans la définition du schéma:
  - valeur nulle impossible (NOT NULL)
  - unicité de l'attribut (UNIQUE)
  - contrainte référentielle (FOREIGN KEY ... REFERENCES ...)
  - contrainte générale (CHECK)
- Réactions prédéfinies aux opérations de suppression et de mise à jour (ON DELETE, ON UPDATE) liées aux contraintes référentielles (NO ACTION, CASCADE, SET DEFAULT, SET NULL)

Création de schéma: CREATE TABLE

```
CREATE TABLE  nomTable (
    colonne1 type1 [DEFAULT valeur1] [NOT NULL],
    colonne2 type2 [DEFAULT valeur2] [NOT NULL],
    [CONSTRAINT nomContrainte1 typeContrainte1 paramContrainte1] …
    );
```

- NomTable: le nom de la relation comportant des lettres, des chiffres et les symboles classiques (_,$,#,etc.).
- Colonnei: le nom d'un attribut de la relation,
- Typei: un type (Oracle) qui définit le domaine d'un attribut.
- CONSTRAINT …: une contrainte pour cette relation (détaillé plus loin),
- DEFAULT: permet de préciser une valeur par défaut (si aucune n'est précisée lors de l'insertion d'un tuple)
- NOT NULL: précise que la valeur NULL (absence de valeur) est interdite pour un attribut.

Types (Oracle) disponibles

- Type chaîne de caractères
  - de longueur fixe : **`CHAR(longueur)`**
    Par défaut la valeur est d'1 caractère et au maximum de 2000 caractères.
  - de longueur variable : **`VARCHAR2(longueur)`**
    Par défaut 1 caractère et au maximum 4000 caractères.
    Le stockage dépend de la taille réelle de la chaîne.
    *La norme définit le type **char(longueur)**.*
  - de flot de caractères: **`CLOB(longueur) jusqu'à 4Go.`**
  - de caractères Unicode: **`NCHAR(l), NVARCHAR2(l), NCLOB(l)`**

Types (Oracle) disponibles

- Type numérique décimale :
  `NUMBER(précision, échelle)`
  - `précision` : nombre entier de chiffres significatifs (1 à 38).
  - `échelle` : nombre de chiffre à droite de la virgule (-84 à 127). Une valeur négative arrondit le nombre.

  ex: VAL NUMBER(8,2) définit la donnée VAL avec 8 chiffres significatif dont deux après la virgule (-999999,99 à +999999,99)
  VAL NUMBER(8,-2) arrondit la donnée à la centaine.
  - *La norme définit les types **decimal(precision, echelle)** et **integer**.*

Types (Oracle) disponibles

- Type date/heure:
  - **`DATE`**: le format standard d'une date qui permet de stocker le siècle, l'année, le mois, le jour, l'heure, les minutes et les secondes. Le format par défaut dépend de la langue (le stockage est indépendant).
  - **`TIMESTAMP`**(précision): date et heure incluant des fractions de secondes (précision de 0 à 9).
  - **`INTERVAL YEAR`** (précision) **`TO MONTH`**: permet d'extraire une différence entre deux moments avec une précision mois/année (précision)
  - **`INTERVAL DAY`** (précision1) **`TO SECOND`** (précision2): différence plus précise entre deux moments (précision1 pour les jours et précision2 pour les fractions de secondes).
- Type binaires:
  - **`BLOB`**: données non structurées jusqu'à 4Go.
  - **`BFILE`**: données stockées dans un fichier externe à la base.

CREATE TABLE (exemple)

```
/* Description des Personnes */
CREATE TABLE Personne (nP CHAR(10), nom VACHAR2(15), adr
    VARCHAR2(50));
/* Description des prénoms
    d'une personne */
CREATE TABLE PersonnePrénoms(
    num CHAR(10),
    prénom VARCHAR2(15) );
-- Description des étudiants
-- qui sont par ailleurs des personnes.
CREATE TABLE Etudiant(nP CHAR(10), nE CHAR(10), dateN DATE);
/* Description des études réalisées par un étudiant */
CREATE TABLE EtudiantEtudes(nE CHAR(10), année NUMBER(1), diplôme
    VARCHAR2(30));
-- Fin du script SQL pour Oracle.
```


Gestion des contraintes

- Les contraintes sont déclarées de deux manières:
  - **mode *inline***: c'est-à-dire en même temps que l'attribut qu'elle contraint (contrainte monocolonne).
  - **mode *out-of-line***: la définition apparaît à la suite des définitions d'attributs, introduit avec le mot clé CONSTRAINT. Dans ce cas la contrainte peut être nommée (plus flexible) et peut porter sur plusieurs attributs (ou colonnes).
- On distingue 4 types de contraintes:
  - **UNIQUE** : impose une valeur distinct d'un attribut pour chacun des tuples.
  - **PRIMARY KEY** : déclare l'identifiant de la relation (ou clé de la table), la contrainte UNIQUE et l'option NOT NULL est sous-entendu pour les attributs concernés.
  - **FOREIGN KEY** : déclare un identifiant externe (ou clé étrangère) ainsi que la méthode employée pour assurer l'intégrité référentielle.
  - **CHECK** : permet de spécifier une condition simple que doit suivre le ou les attributs.

Exemple de schéma (amélioré)

```
/* Description des Personnes */
CREATE TABLE Personne (
    nP CHAR(10) NOT NULL UNIQUE, -- contrainte UNIQUE inline
    nom VACHAR2(15) NOT NULL,
    adr VARCHAR2(50));
/* Description des prénoms
    d'une personne */
CREATE TABLE PersonnePrénoms(
    num CHAR(10) NOT NULL,
    prénom VARCHAR2(15) NOT NULL,
    -- contrainte out-of-line nommée
    CONSTRAINT une_fois_prenom UNIQUE (num,prénom) );
```

On peut faire beaucoup mieux en introduisant les identifiants !

Contrainte PRIMARY KEY

```
/* Description des Personnes */
CREATE TABLE Personne (
    nP CHAR(10) PRIMARY KEY, -- contrainte inline
    nom VACHAR2(15) NOT NULL,
    adr VARCHAR2(50));
/* Description des prénoms
    d'une personne */
CREATE TABLE PersonnePrénoms(
    num CHAR(10),
    prénom VARCHAR2(15),
    -- contrainte out-of-line nommée
    CONSTRAINT pk_PersonnePrénoms PRIMARY KEY (num,prénom) );
```

- Dans la première table on peut utiliser une contrainte *inline* où *out-of-line*.
- Dans la seconde table, la contrainte *out-of-line* est obligatoire: elle porte sur deux colonnes, l'identifiant étant composé de deux attributs.

Contrainte Foreign Key

- Cette contrainte est le noyau de la gestion de cohérence d'une base de données relationnelle: l'intégrité référentielle.
  - basée sur une relation entre identifiant (clé primaire/candidate) et identifiant externe (clé étrangère): "Un étudiant est une personne qui doit exister dans la base".
  - on parle souvent de tables:
    - **père**: cette table possède la clé primaire référencée (ex: Personne).
    - **fils**: cette table possède une clé étrangère qui référence la clé primaire/candidate d'une table père (ex: Etudiant).
- Les problèmes résolus/évités par un SGBD en activant cette cohérence:
  - **fils vers père**: impossible d'ajouter un tuple fils qui ne puisse pas se rattacher à un tuple père (ex: pas d'étudiant sans être une personne),
  - **père vers fils**: impossible de supprimer un tuple père ayant encore des tuples fils qui lui sont rattachés (ex: plus d'étudiants si plus de personnes).

Contrainte Foreign Key (suite)

- Il faut donc gérer cette contrainte qui assure l'intégrité référentielle du coté père et du coté fils:
  - du coté père: on s'assure d'avoir une clé primaire (PRIMARY KEY) pour pouvoir référencer les tuples de la relation père.
  - du coté fils: on définit la clé étrangère à l'aide d'une contrainte FOREIGN KEY (qui précise le ou les attributs du fils qui référencent la clé primaire/candidate du père).
- CONSTRAINT contrainte FOREIGN KEY(f_attr1, f_attr2) REFERENCES p_table(p_attr1,p_attr2)
  - contrainte: nom de la contrainte pour faciliter sa gestion,
  - f_attr_i: nom des attributs de la table fils composant la clé étrangère,
  - p_table: nom de la table père référencée,
  - p_attr_i: nom des attributs de la table père p_table composant sa clé primaire/candidate qui sont référencés par f_attr_i.

Contrainte Foreign Key (exemple)

```
/* Description des Personnes: table père */
CREATE TABLE Personne (
    nP CHAR(10) PRIMARY KEY, -- définition de l'identifiant chez le père
    nom VACHAR2(15) NOT NULL,
    adr VARCHAR2(50));
/* Description des Etudiants: table fils */
CREATE TABLE Etudiant(nE CHAR(10) PRIMARY KEY,
    nP CHAR(10),
    dateN DATE,
    -- définition de l'identifiant externe
    CONSTRAINT fk_Etudiant_Personne FOREIGN KEY(nP)
        REFERENCES Personne(nP) );
```


Contrainte Foreign Key (gestion de la cohérence père -> fils)

- La suppression d'un tuple du père donnent trois alternatives:
  - on interdit la suppression de ce tuple s'il est encore référencé par des tuples des fils: il faut supprimer tous les tuples fils avant. On ne doit pas survivre à ces enfants!
  - on accepte la suppression de ce tuple mais cette suppression est automatiquement propager aux tuples des fils. Ainsi la cohérence est respectée. La clause optionnelle ON DELETE CASCADE doit être ajoutée.
  - on accepte cette suppression de tuple mais sans propagation aux tuples des fils. Les composants de la clé étrangère des tuples des fils prennent la valeur NULL. La clause optionnelle ON DELETE SET NULL doit être ajoutée (et ne pas contredire les contraintes NOT NULL).
- Exemple:

```
CREATE TABLE Etudiant(nE CHAR(10) PRIMARY KEY,
    nP CHAR(10),DateN DATE,
    -- définition de l'identifiant externe
    CONSTRAINT fk_Etudiant_Personne FOREIGN KEY(nP)
        REFERENCES Personne(nP) ON DELETE CASCADE);
```


Contrainte Check(condition)

- Cette contrainte contrôle l'insertion ou la modification d'un tuple en fonction d'une condition explicite:
  - impose un domaine de valeurs (restriction plus précise qu'un typage),
  - Les contraintes plus complexes se programment via des "déclencheurs" .
- Exemple:

```
CREATE TABLE Etudiant(nE CHAR(10) PRIMARY KEY, nP CHAR(10), DateN DATE,
    CONSTRAINT ck_DateN CHECK ( (SYSDATE-DateN)/365>18) );
```

- La condition accepte divers opérateurs:
  - de comparaison (>,=,<,>=,<=,<>)
  - logiques (NOT, AND, OR)
  - mathématiques (+,-,*,/)
  - des fonctions (cf. fin de ce cours)
  - des opérateurs intégrés (BETWEEN, IN, LIKE, IS NULL)

Contrainte CHECK (Suite)

- Les opérateurs intégrés (comme le format de condition) sont très utilisés en SQL (en mode interrogation).

```
CREATE TABLE Personne (
    nP CHAR(10) PRIMARY KEY,
    nom VACHAR2(15),
    adr VARCHAR2(50)
```

  - Opérateur BETWEEN … AND … teste l'appartenance à un intervalle de valeurs, renvoie un booléen (VRAI/FAUX).

```
CONSTRAINT ck_nP1 CHECK (nP BETWEEN 1 and 10),
```

  - Opérateur IN (…) teste l'appartenance à un ensemble de valeurs.

```
CONSTRAINT ck_nP2 CHECK (nP IN (1,2,3,4,5,6,7,8,9,10)),
```

  - Opérateur LIKE … compare une expression à une chaîne de caractère type (% pour un ou plusieurs caractères, _ pour un seul caractère).

```
CONSTRAINT ck_adr CHECK (adr LIKE '% Grenoble %'),
```

  - Opérateur IS NULL teste si une expression correspond à la valeur NULL.

```
CONSTRAINT ck_nom CHECK (nom NOT IS NULL) );
```


Évolution de schéma

- La commande DROP (pour supprimer une table)
  - DROP TABLE nom_table [CASCADE CONSTRAINTS];
  - CASCADE CONSTRAINTS
    - Supprime toutes les contraintes référençant une clé primaire (primary key) ou une clé unique (UNIQUE) (permet de supprimer une table sans être gêné par ses fils)
    - Si on cherche à détruire une table dont certains attributs sont référencés sans spécifier CASCADE CONSTRAINT, Oracle retourne une erreur.
  - Avant de créer un schéma il est toujours préférable d'avoir un script qui le détruit (pour éviter les erreurs de cohérence):
    - La destruction suit un chemin précis: d'abord les fils, ensuite les relations pères (inverse de la création).

```
DROP TABLE EtudiantEtudes;
DROP TABLE Etudiant;
DROP TABLE PersonnePrénoms
DROP TABLE Personne;
```


Manipulation de données

- Après la création d'un schéma, il faut peupler les tables, c'est-à-dire insérer des tuples.
- La commande INSERT INTO permet d'insérer ligne par ligne de nouveaux tuples:

INSERT INTO table VALUES …

  - table: le nom de la table où insérer un tuple.
  - on peut renseigner ensuite toutes les valeurs d'un tuple:

```
INSERT INTO Personne VALUES (1,'Dupont', 'Grenoble');
```

  - ou bien ne renseigner que certaines valeurs:

```
INSERT INTO Personne(nP, nom) VALUES (2,'Aubry');
```

La valeur de l'attribut Adr est alors implicitement NULL.

- Problème avec la date:
  - les formats '01/09/04', '01-09-04' et '01-09-2004' sont équivalents
  - sinon il faut utiliser la fonction TO_DATE(texte,format) qui transforme une chaîne de caractères en une date en fonction d'une expression de formatage:

```
INSERT INTO Etudiant VALUES (1,1001,'22-01-71');
ou INSERT INTO Etudiant VALUES(1,1001,TO_DATE('22 Janvier 71','DD MONTH YY');
```


Limitation SQL

- Prémisse d'une gestion active: la commande CREATE ASSERTION permet de tester des conditions avant ou après certains événements.

```
CREATE ASSERTION <nom contrainte>
    [BEFORE COMMIT |
    AFTER {INSERT|DELETE|UPDATE[OF (Attribut+)]} ON <Relation>]
    CHECK <Condition>
    [FOR [EACH ROW OF] <Relation>]
```

Exemple :

```
CREATE ASSERTION Mindegre
    BEFORE COMMIT
    CHECK (SELECT MIN(Degre) FROM Vins >10)
    FOR Vins;
```

! Implémentation réduite dans les SGBD !

Règle active

règles ECA

- Event: LORSQUE l'événement *E* se produit
- Condition: SI la condition *C* est satisfaite
- Action: ALORS exécuter l'action *A*

Les événements

L'événement spécifie la cause qui déclenche une règle.

- Modification des données
  insert, delete, update, méthode (BDOO uniquement)
- Interrogation
  select, méthode (BDOO)
- Temporel
  le 1er janvier 2000 / tous les jours à 12h00 / toutes les 10 minutes / ....
- Définis par l'application (température trop élevée, login-utilisateur, ...) : cet événement doit être correctement déclaré et son monitoring bien défini.

Événements composés

- opérateurs logiques
  - AND, OR, NOT
- séquences logiques
  - avant, après, ...
- séquences temporelles
  - 5 secondes après e1
  - 10 fois toutes les 30' après e2

Conditions

La condition définie une condition supplémentaire pour valider l'action après déclenchement d'une règle.

- prédicats BD
  - WHERE SQL
- requête SQL
  - résultat vide / non vide
- fonctions d'agrégation, ...
- procédures ad-hoc (retourne Vrai / Faux)

Actions

Action exécutée quand une règle est déclenchée et sa condition OK.

- opérations de modification des données
  - insert, delete, update
  - appel de méthodes
- interrogation de la base de données
  - select
- autres
  - rollback, commit, abort, grant /revoke privilèges, ...
- procédures ad-hoc
- déclenchement d'autres règles

Exécution de l'action

- immédiate
- à la fin de la transaction
- check point
- autre moment explicitement défini

Algorithme de déclenchement

```
TANT QUE il y a une règle à déclencher

   PRENDRE une règle déclenchable
      EVALUER la condition
      SI la condition est vérifiée ALORS
            EXECUTER l'action
FIN TANT QUE
```


Organisation des règles

- le même événement peut déclencher plusieurs règles
  - priorités entre règles
  - priorités statiques: 1ère, 2ème, ...
  - priorités relatives: celle-ci avant celle-là
- structuration de l'ensemble des règles (create / delete / modify / enable / disable, ...)

Quand examiner les règles ?

- dès qu'un événement survient
- à la fin de l'exécution d'une requête SQL
- à la fin de la transaction
- périodiquement
- à la demande

Quand examiner les règles ? (Oracle)

- Mode classique: gestion **immédiate**, par défaut les contraintes sur les schémas sont testées pour chaque opération.
  - *Set constraint Nom_contrainte immediate*
- Mode relâchée: gestion **différée**, les contraintes ne sont testées qu'au moment de la validation d'une transaction.
  - *Set constraint Nom_contrainte deffered* si la contrainte a été définie ***defferable***.

Séquentiel versus parallèle

- Exécution séquentielle
  - choisir une règle, l'exécuter
  - choisir une autres règle, l'exécuter
- Exécution parallèle
  - choisir une règle, l'exécuter
  - pendant l'exécution, choisir une autres règle et l'exécuter
- Exécution groupée
  - toutes les règles déclenchables sont exécutées en parallèle
  - **Choix de l'ordre de déclenchement et exécution ?**

Terminaison

- L'algorithme de déclenchement peut ne pas terminer (boucle dans le graphe de déclenchement)
- solutions :
  - laisser faire
  - admettre un nombre maximal de règles exécutées après un événement
  - interdire l'activation d'une règle par une autre règle
  - restreindre les activations en chaîne au sein d'un groupe de règles

Implémentation

- mémorisation et gestion des règles (création, suppression, modification, activation, persistance)
- Concurrence (sur les règles, les événements, les conditions et les actions)
- Fiabilité (en cas de déclenchement d'une règle pendant un crash)
- Autorisations d'accès / privilèges sur les règles
- Gestion d'erreurs à différents niveaux
- Tracer les règles pour le debugging
- Performances des règles
- ....

Utilisations des règles

- vérification de contraintes d'intégrité => Gestion Intégrité
  - avantage: on ne vérifie que les données en cause
  - possibilité d'action corrective
  - contraintes statiques: facile
  - contraintes dynamiques : événements composés, ...
- maintien de la cohérence des vues (virtuelles, matérialisées, instantanées)
  => Gestion de données dérivées
- gestion de versions => Gestion de réplication
  - propagation des modifications de version
  - règle d'évolution des versions
- évolution de schéma

Intégrité: Génération de règles

- Peut être semi-automatique
  - (1) formule déclarative de la CI ⇒ condition
  - (2) causes possibles de la violation ⇒ événements
  - (3) action réparatrices ⇒ actions
- Approche simple: CI encodées comme règles d'avortement
  - Activées par tout événement qui pourrait violer la contrainte
  - Précondition teste l'occurrence des violations
  - Si violation: commande ROLLBACK
  - Inconvénient: cause beaucoup de ROLLBACK

Intégrité: Génération de règles, cont.

- Autre approche: CI encodées comme règles de réparation

- CI spécifiées avec actions à réaliser pour restaurer l'intégrité
- Ex: intégrité référentielle SQL-92
* même pré-condition et événements que règles d 'avortement
* actions contiennent des manipulations de la BD
- Ex: actions de réparation pour l'intégrité référentielle reflètent la sémantique de CASCADE, RESTRICT, SET NULL, SET DEFAULT

Intégrité : Analyse des contraintes

- Les CI conservent la base dans un état cohérent
- Mais les CI sont elles correctes:
  - sont-elles cohérentes entre elles ?

```
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre>12;
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre<11;
```

  - Ne sont-elles pas redondantes ?

```
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre>12;
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre>11;
```

- sont-elles minimales ou peut on les simplifier ?

Les règles dans les systèmes existants

- SQL 3 : standardisation de la syntaxe des déclencheurs (règles)
- Oracle:
  - les règles ne peuvent pas manipuler la relation en cause
  - gestion de la règle au niveau table ou au niveau ligne.

Oracle et les triggers

- Un trigger est un « programme résident » dans une BD
  - Associé à un événement (insertion, modification, suppression)
  - qui porte sur une table
  - Déclenché automatiquement
  - Et programmé en PL/SQL
- Un trigger permet par exemple
  - De définir une politique de sécurité complexe
  - De modifier la valeur d'un attribut en fonction de la valeur d'autres attributs de différentes tables
  - De valider des valeurs d'attribut en fonction d'un calcul complexe relatif aux valeurs d'autres attributs
  - D'archiver des informations automatiquement.

Trigger

- De manière générale un trigger assure:
  - L'implémentation des règles de gestion complexes,
  - La déportation des contraintes au niveau du serveur de données,
  - La programmation de l'intégrité référentielle
  - L'archivage de données
- Cycle de vie d'un trigger
  - Spécifier et analyser les règles de gestion
  - Coder en PL/SQL les règles à implémenter au niveau du serveur de données
  - Compiler chaque trigger (stocké dans la base)
  - Activation/Désactivation des triggers

Les bases de PL/SQL

- Oracle fournit une extension à SQL, sous la forme d'un langage procédural mêlé à des commandes SQL
  - Script PL/SQL pour éteindre les possibilités du SQL
  - Procédure stockée
  - Déclencheur pour la gestion de règles
- Avantages de PL/SQL:
  - Blocs d'instructions très modulaire
  - Indépendance de l'OS et du langage de l'application
  - Intégration à SQL

Structure d'un programme PL/SQL

```
DECLARE
    -- Déclarations des variables, types, exceptions
    …
BEGIN
    -- Code PL/SQL avec (ou pas) des directives SQL
    …
EXCEPTION
    /* Traitement des erreurs du SGBD */
    …
END;
/ -- (symbole nécessaire pour interpréter un bloc en tant que bloc PL/SQL)
```

→

```
DECLARE
    Déclarations…
BEGIN
    Code PL/SQL…
EXCEPTION
    Gestion des erreurs…
END;
```


Variables et typage en PL/SQL

- Variables scalaires
  - D'un type de base (number, char, boolean, varchar2, date, timestamp, interval, blob, etc.)
  - D'un type prédéfini:
    - Binary_integer (-2^31 à 2^31), natural (0 à 2^31), positive (1 à 2^31), pls_integer
    - Decimal, integer, float

NomVariable [CONSTANT] type [NOT NULL] [:=|DEFAULT expression];

Ex: dateNaiss DATE:=sys.date();

Variables et typage en PL/SQL

- Exemple très simple:

```
SQL>create table etudiant(    num integer,
                              nom varchar2(20),
                              prenom varchar2(20),
                              dateNaiss DATE,
                              dateInscr DATE);
SQL>declare
    i int;
Begin
    for i in 1..100 loop
        insert into etudiant(num,dateInscr)
        values (i, sysdate);
    end loop;
    Commit;
End;
/
```


Variables et typage en PL/SQL

- Type indirect (%TYPE)
  - Le type d'une variable est déclarée comme le type d'un attribut d'une table existante.

NomVariable nomTable.nomColonne%TYPE

Ex: name etudiant.nom%type;

- Type complexe implicite (%ROWTYPE)
  - Défini une variable comme une structure dont le schéma est celui d'une table.

Ex: MonEtudiant etudiant%ROWTYPE;

Variables et typage en PL/SQL

- Exemple de déclaration indirect/implicite:

```
Declare
    lastname etudiant.nom%type='Doe';
    firstname etudiant.prenom%type='John';
    MonEtudiant etudiant%ROWTYPE;
    i int;
Begin
    update etudiant
    set nom=lastname, prenom=firstname
    where nom is NULL;

    MonEtudiant.nom='Smith'; MonEtudiant.prenom='John';
    MonEtudiant.DateNaiss=NULL;
    for i in 1..100 loop
        MonEtudiant.num=i;MonEtudiant.DateInscr=sysdate;
        insert into etudiant values MonEtudiant;
    end loop;
    Commit;
End;
/
```


Variables et typage en PL/SQL

- Type structurée RECORD
  - Une variable peut être structurée et explicitement définie à partir d'un type complexe.

Ex:
```
TYPE rec_etudiant IS RECORD (num integer,nom varchar2(20),prenom
varchar2(20),dateNaiss DATE,dateInscr DATE);
MonEtudiant rec_etudiant;
```

- Type tableau TABLE
  - Une variable peut être un tableau dynamique (pas de dimensionnement initial). Un tableau est une table (relationnelle) composé d'un identifiant et d'un contenu.
  - Un ensemble de fonctions permettent de gérer les tableaux.

Ex:
```
TYPE tab_etudiant IS TABLE OF rec_etudiant INDEX BY
BINARY_INTEGER;
LesEtudiants tad_etudiant;
…
LesEtudiants(-1)=MonEtudiant;
LesEtudiants(50)=MonEtudiant;
…
```


Variables et typage en PL/SQL

- Fonctions de gestion des tableaux
  - EXISTS(x) : retourne TRUE si le xieme élément existe
  - COUNT : retourne le nombre d'éléments du tableau
  - FIRST/LAST : retourne le premier/dernier indice du tableau
  - PRIOR(x)/NEXT(x) : retourne l'élément suivant/précédent le xième élément
  - DELETE, DELETE(x), DELETE(x,y) : supprime un ou plusieurs éléments d'un tableau.
  - Fonctions non utilisables dans une commande SQL !
- Opérateurs utiles sur les variables:
  - :=, IS NULL, IS NOT NULL
- Variables de session (= variable globale/statique)
  - Mot clé VARIABLE au début d'un bloc SQL/PLUS
  - Affichage par PRINT

Ex:
```
Variable compteur number;
Begin   :compteur:=:compteur+1; End;/
```


Structure de contrôle PL/SQL

- Structures conditionnelles:

```
If cond then
  instructions;
End if;
```

```
If cond then
  instructions;
Else
  instructions;
End if;
```

```
If cond1 then
  instructions;
Elsif cond2 then
  instructions;
Else
  instructions;
End if;
```

```
Case
  when cond1 then instructions1;
  when cond2 then instructions2;
  …
  else instructions;
End case;
```


Structure de contrôle PL/SQL

- Structures répétitives:

```
While cond loop
  instructions;
End loop;
```

```
Loop
  instructions;
  exit when cond;
End loop;
```

```
For compt in [reverse] Vinf..VSup loop
  instructions;
End loop;
```


Manipulation de données en PL/SQL

- PL/SQL permet de récupérer les résultats d'une requête dans des variables.
  - Requête SELECT FROM WHERE classique
  - Mot clé INTO pour transférer les valeurs
  - La requête ne doit renvoyer qu'un seul tuple (résultat mono-valué)
  - Une requête SELECT est obligatoirement associé à INTO

Ex:
```
declare
    nb integer;
Begin
    select count(*) INTO nb from etudiant;
    …
End;
/
```

- Pour gérer des résultats multi-valués, on utilise des curseurs (cf. cours IBD)

Manipulation de données en PL/SQL

- Une requête SELECT qui ne renvoie aucun résultat retourne une exception:
  - Erreur NO_DATA_FOUND (ORA-01403)
  - Il faut capturer l'exception
- Insertion,Modification,Suppression
  - Des variables peuvent être utilisées dans les clauses WHERE de la requête
  - Une requête de MAJ ne produit aucune erreur si aucun tuple modifié/supprimé
  - Variables curseurs "implicites" utiles:
    - SLQ%ROWCOUNT = nombre de tuples modifiés/supprimés
    - SQL%FOUND = TRUE si au moins un tuple modifié/supprimé
    - SQL%NOTFOUND = TRUE si contraire SQL%FOUND

Notion de curseur

- Gestion d'une requête multivaluée
  - déclaration du curseur: association à une requête SQL (CURSOR)
  - Ouverture du curseur: préparation du résultat (OPEN)
  - Parcours des tuples résultats (FETCH)
  - Libération des ressources du curseur (CLOSE)
- Exemple: parcourir la table étudiant limitée aux étudiants de plus de 30ans
  - tuple = (nom, prenom)
  - affichage (nom,prenom) de chaque étudiant

Exemple curseur

```
DECLARE
        CURSOR c1 IS SELECT nom, prenom FROM etudiant WHERE age>30;
        UnTuple c1%ROWTYPE;
        BEGIN
                OPEN c1;
                FETCH c1 INTO UnTuple;
                WHILE (c1%FOUND) LOOP
                        DBMS_OUTPUT.PUT_LINE('identité: '||c1.nom||c1.prenom);
                        FETCH c1 INTO UnTuple;
        END LOOP;
        CLOSE c1;
        END;
/
```

- **Idem avec une structure de répétition FOR**

```
BEGIN
        FOR UnTuple IN c1 LOOP
                DBMS_OUTPUT.PUT_LINE('identité: '||c1.nom||c1.prenom);
        END LOOP:
END;
/
```


Curseur paramétré

- Un curseur peut spécifier des paramètres :
  - permet de paramétrer la requête
  - instancié au moment de l'ouverture du curseur

```
DECLARE
        CURSOR c1(p_age IN INTEGER) IS SELECT nom, prenom FROM etudiant WHERE age>p_age;
        UnTuple c1%ROWTYPE;
BEGIN
        OPEN c1(30);
        FETCH c1 INTO UnTuple;
        WHILE (c1%FOUND) LOOP
                DBMS_OUTPUT.PUT_LINE('identité: '||UnTuple.nom||UnTuple.prenom);
                FETCH c1 INTO UnTuple;
        END LOOP;
        CLOSE c1;
        FOR UnTuple IN c1(25) LOOP DBMS_OUTPUT.PUT_LINE('identité:
'||UnTuple.nom||UnTuple.prenom);
        END LOOP:
END;
/
```


Modification via un curseur

- Modifier des tuples accéder par un curseur:
  - verrouillage de la table:
    FOR UPDATE [OF colonne] [WAIT/NOWAIT]
  - utilisation de clause SQL de MàJ
- Exemple: Ajout d'un commentaire aux étudiants >30ans et suppression des étudiants <16ans

```
DECLARE
        CURSOR c1 IS SELECT * FROM etudiant FOR UPDATE OF remarque;
        UnTuple c1%ROWTYPE;
BEGIN
        OPEN c1;
        FETCH c1 INTO UnTuple;
        WHILE (c1%FOUND) LOOP
                IF UnTuple.age>30 THEN UPDATE etudiant SET remarque='Vieux' WHERE CURRENT OF c1;
                ELSIF UnTuple.age<16 THEN DELETE FROM etudiant WHERE CURRENT OF c1;
                END IF;
        FETCH c1 INTO UnTuple;
        END LOOP;
        CLOSE c1;
END;
/
```


Aspect Transactionnel

- Un Bloc PL/SQL supporte la gestion des transactions:
  - COMMIT : termine une transaction
  - ROLLBACK : annule une transaction
  - ROLLBACK TO savepointLabel : annule jusqu'à un point de sauvegarde
- Possibilité d'imbriquer des transactions dans des blocs PL/SQL imbriqués.

Procédure stockée

- Une procédure stockée (ou cataloguée) est une procédure/fonction stockée au sein du SGBD
  - Compilation lors de la définition du bloc PL/SQL "procédure"
  - Recompilation automatique en cas de modification de schéma
  - Appel à l'aide de l'instruction EXECUTE
- Avantage
  - Gestion des droits par le SGBD
  - Traitement au sein du SGBD

Procédure stockée

```
Create procedure nomProcedure(
    nomParam [IN|OUT| IN OUT] typeSQL, …)
    [AUTHID CURRENT_USER|DEFINER]
    [AS PRAGMA AUTONOMOUS_TRANSACTION]
    Zone des variables locales…
BEGIN
    instructions…
END;
/
```

- IN,OUT,IN OUT = paramètres en entrée, sortie, les deux
- AUTHID = change les droits à l'exécution (propriétaire ou utilisateur)
- Une transaction autonome met en pause la transaction principale et lui rend la main après terminaison
- Des procédures imbriquées peuvent être définies dans la zone de déclaration d'une procédure

Procédure stockée

- **Exemple 1**: Corriger les étudiants dont la date d'inscription est < à la date de naissance

```
Create or replace procedure VerifDate
is
Begin
    update etudiants
    set dateNaiss=NULL; dateInscr=NULL;
    WHERE dateInscr<dateNaiss;
End;
/
```

- **Exemple 2**: retourne le nombre d'étudiants inscrits à une date donnée.

```
Create or replace function NbEtud (dinscr in date) return number
Is
Resultat number:=0;
Begin
    select count(*) into resultat from etudiants
    where dateInscr=dinscr;
    return resultat;
End;
/
```


Procédure stockée

- Compilation: automatique lors de la création
- Affichage détaillée des erreurs:
  Show errors (sur la console SQL*PLUS)
- Appel d'une procedure/fonction:

```
Exemple:
Variable nb number;
Begin
    nb:=nbEtud(sysdate);
    verifDate();
    …
    select dateInscr, nbEtud(dateInscr) from etudiants
    groupe by dateInscr;
End;
```

Interaction

- Paquetage DBMS_OUTPUT
  - Pour l'activer: set serveroutput on (sur la console)
  - Procedures disponibles:
    - ENABLE / DISABLE
    - PUT(message IN): affiche un message
    - PUT_LINE(message IN): + saut de ligne
    - GET_LINE(entree OUT): saisi d'un texte

Exemple:

```
DBMS_OUTPUT.ENABLE;
DBMS_OUTPUT.PUT_LINE('Bonjour');
```


Gestion des exceptions

- La gestion des exceptions est classique:
  - Une erreur lève une exception
  - La gestion se fait dans la section EXCEPTION d'un bloc PL/SQL
  - Sinon elle est propagée…

```
EXCEPTION
   WHEN except1 THEN instructions1;
   WHEN except2 OR except3 THEN instructions2;
   WHEN OTHERS THEN instructions;
```

- Exceptions système et Exceptions utilisateurs

Gestion des exceptions

- Exemple:

```
Create function percentEtud(dinscr IN date)
IS
    total number:=0;
    resultat number:=0;
    pas_d_etudiant exception;
Begin
    select count(*) into total from etudiants;
    select count(*)/total into resultat from etudiants
    where dateInscr=dinscr;
    if resultat=0 then raise pas_d_etudiant;
Exception
    when ZERO_DIVIDE then dbms_output.put_line('Aucun Etudiant dans la base!');
    when pas_d_etudiant then dbms_output.put_line('Aucun Etudiant inscrit le
    '||dinscr);
End;
/
```


Gestion des exceptions

- Résultat de requête vide
  - Une requête sans résultat lève une exception NO_DATA_FOUND
  - Comment identifier la requête sans résultat si plusieurs requêtes n'ont pas de résultat

```
Exemple:
dinscr date; dnaiss date;
Requete number;
BEGIN
    requete:=1;
    select dateInscr into dinscr from etudiants where name='Dupont';
    requete:=2;
    select dateNaiss into dnaiss from etudiants where name='Durand';
    …
Exception
    when NO_DATA_FOUND then
            if requete=1 then …
            else …
End;
```


Programmer un déclencheur

- Un trigger est construit sur la base d'une procédure déclenchée par un événement de MAJ:
  - Insert, delete, update sur une table: déclencheurs LMD
  - Create, alter, drop sur un objet (table, index, etc.): déclencheurs LDD
  - Startup/shutdown, ouverture/fermeture session: déclencheurs d'instances
- Nous étudierons les triggers LMD !

Structure d'un trigger LMD

- La définition d'un déclencheur se compose de 7 parties principales
  - Sa commande de création

```
Create or replace trigger nomTrigger
```

  - L'information temporelle de déclenchement relatif à l'événement déclencheur

```
BEFORE | AFTER
```

  - Le type d'événement déclencheur

```
DELETE | INSERT | UPDATE [OF nomAttribut,…]
(disjonction possible: delete OR insert OR update)
```

  - La table sur laquelle porte l'événement

```
ON nomTable (une seule à la fois !!!)
```

  - Le mode du trigger: mode ligne / mode table

```
[FOR EACH ROW]
```

  - La condition de déclenchement (expression SQL de type clause WHERE)

```
WHEN condition
```

  - Le code PL/SQL associé

```
DECLARE…BEGIN… END
```


Restriction du code PL/SQL

- Un trigger n'est pas une application complète : le code doit être court
  - Faire appel à des procédures
  - Éviter la récursivité
- Un trigger est un point critique dans lequel la base n'est pas cohérente
  - Pas de gestion de transaction dans le code d'un trigger
  - COMMIT, ROLLBACK, SAVEPOINT sont interdits
- La création d'un trigger l'active automatiquement
  - **Alter trigger nomTrigger DISABLE/ENABLE** pour activer/désactiver un trigger
  - **Alter table nomTable DISABLE/ENABLE ALL triggers** concerne tous les triggers d'une table
  - **Drop trigger** nomTrigger supprime un trigger

Trigger de type table

- La granularité de l'événement est la table:
  - L'ajout, l'insertion, la modification d'un ou plusieurs tuples dans une table
  - Le raisonnement du trigger (code) porte sur l'accès à une table, pas à un tuple particulier

```
Exemple: Interdiction d'accéder à la table étudiants pendant le weekend
Create trigger periodeOK
Before delete or update or insert on etudiants
Begin
    if to_char(sysdate,'DAY') in ('SAMEDI', 'DIMANCHE') then
            raise_application_error(-20101, 'Accès impossible le weekend!);
    end if;
End;
/
```

- Noter la manière de déclencher facilement une Exception utilisateur avec raise_application_error(numErreur, message).

Trigger de type table

- Utilisation de la clause WHEN
  - Pour éviter l'exécution inutile d'un trigger (couteux pour le système)
  - Pour remonter une condition hors du code

Exemple: La clause WHEN s'applique parfaitement à l'exemple précédent

```
Before delete or update or insert on etudiants
When to_char(sysdate,'DAY') in ('SAMEDI', 'DIMANCHE')
Begin
    raise_application_error(-20101, 'Accès impossible le weekend!);
End;
/
```


Trigger de type ligne

- La granularité de l'événement est la ligne:
  - L'ajout, l'insertion, la modification d'un ou plusieurs tuples dans une table déclenche l'évenement pour chaque tuple
  - Le raisonnement du trigger (code) porte sur l'accès à une ligne
  - Accès possible aux informations du tuple accéder
    - :NEW référence le tuple insérée ou modifiée
    - :OLD référence le tuple supprimer ou l'état du tuple avant modification
    - Les directives :NEW et :OLD s'utilisent "comme une table"
- Un trigger en mode ligne pose des verrous sur les données : Impossible d'accéder à la table ayant déclencher l'événement dans le code du trigger !

Trigger de type ligne

Exemple 1: recopie les étudiants supprimés dans une table archive.

```
Create trigger archiverEtudiants
After delete on etudiants
For each row
Begin
    insert into archive
    values(:old.num,:old.name,:old.prenom,:old.inscr);
End;
/
```


Trigger de type ligne

Exemple 2: Refuse l'ajout d'un étudiant qui a été précédemment supprimé (en regardant dans la table archive).

```
Create trigger TestEtudiant
before insert on etudiants
For each row
Declare
    vnum archive%num:=0;
Begin
    select num into Vnum
    from archive
    where num=:new.num;
    if vnum<>0 then raise_application_error(-20100,'Etudiant déjà supprimé!');
    end inf;
Exception
    when NO_DATA_FOUND then dbms_output.putline('OK');
End;
/
```


Le problème des tables mutantes

- Un trigger en mode ligne interdit l'accès à la table ayant déclenché l'événement
  - **La table est dite en mutation (mutating table)**
  - C'est un problème fréquent qu'il faut résoudre à l'aide d'un trigger en mode table!

Exemple: Empêcher l'ajout d'étudiant lorsqu'on a déjà 1000 étudiants dans la base.

```
Create trigger ControlAjout
before insert on etudiants
For each row
Declare
    nb integer;
Begin
    select count(*) into nb from etudiants;
    if nb>1000 then raise_application_error('Trop de monde dans la base !');
    end if;
End;
/
```

Si requête INSERT into etudiants values (….) arrive et qu'il y a déjà 1000 tuples dans la base alors on obtient
Mutating tables (erreur: ORA-04091: table etudiants en mutation)
Et pas le message attendu: le code du trigger a généré une erreur système !

Le problème des tables mutantes

- Résolution des tables mutantes
  - On passe le trigger en mode table
  - On contrôle l'état de la table
  - On génère l'erreur
- Problème
  - En mode table, pas d'accès possible au directives :NEW et :OLD
  - Il faut ruser… ou bien gérer la cohérence dans l'application (Rollback) !

Le problème des tables mutantes

- Réécriture d'un trigger ligne en trigger table

```
Create trigger ControlAjout2
before insert on etudiants
Declare
    nb integer;
Begin
    select count(*) into nb from etudiants;
    if nb=1000 then raise_application_error('Trop de monde dans la base !');
    end if;
End;
/
```

=> Mais ce n'est pas toujours aussi simple !

Le problème des tables mutantes

- Exemple plus difficile (le trigger en mode ligne se justifie)

Un étudiant ne peut pas être ajouté à la base s'il en existe déjà un avec le même nom à la même adresse (on oublie le prénom pour simplifier un peu). La clé primaire est ici son numéro.

```
Create trigger ControlAjout3
before insert on etudiants
For each row
Declare
    nb integer;
Begin
    select count(*) into nb from etudiants e (=> génère Mutating Table)
    where e.nom=:new.nom and e.adr=:new.adr
    if nb>0 then raise_application_error('L'étudiant existe déjà !');
    end if;
End;
/
```

Le problème des tables mutantes

- Réécriture de notre exemple complexe
  - On doit passer en mode table
  - Changer le moment d'évaluation du trigger (on laisse faire)
  - Et modifier la requête pour tester l'état de la base: incohérence ?

```
Create trigger ControlAjout3
after insert on etudiants
Declare
    nb integer;
Begin
    select count(*) into nb from etudiants e1, etudiants e2
    where e1.nom=:e2.nom and e1.adr=e2.adr and e1.np!=e2.np
    if nb>0 then raise_application_error('Plusieurs étudiants possèdent le même
    nom');
    end if;
End;
/
```

=> L'application traitera l'exception ou le système annulera la transaction (rollback)

Trigger LDD

- Les évènements sont des modifications du dictionnaire
  - DROP : suppression d'un objet
  - CREATE : création d'un objet
  - etc
- Syntaxe identique, indiquer la base cible
  - BEFORE|AFTER DROP|CREATE ON nom_base.SCHEMA

```
Create trigger trg100
after DROP on jouanotf.SCHEMA
Begin
    …
End;
/
```


Optimisation des triggers

- L'exécution d'un trigger doit être court (traitement dans un état transitoire):
  - Tests + exceptions
  - Répercutions de MAJ (attributs calculés ou liés)
- Limiter le nombre de triggers
  - Bien penser les triggers pour regrouper des contrôles de cohérence dans un même trigger
  - Utiliser le regroupement d'événements:

BEFORE|AFTER insert or update or delete

et dans le code du trigger:
IF (INSERTING) THEN …
IF (UPDATING ('nom_de_colonne') THEN …
IF (DELETING) THEN …

Ordonnancement de l'exécution des triggers

- Il est très difficile de se baser sur un ordre prévisible d'exécution de triggers dont les événements déclencheurs arrivent au même instant.
- En théorie:
  - Tous les triggers table BEFORE
  - Analyse par le SGBD des lignes affectées par la requête
  - Tous les triggers ligne BEFORE
  - Verrouillage et vérification des Contraintes d'intégrités
  - Tous les triggers ligne AFTER
  - Vérifications des contraintes différées
  - Tous les triggers table AFTER
- En pratique on essaie de ne pas avoir besoin de prédire un ordre précis…

Use case: Le Zoo

- Ressortir les contraintes applicatives:
  - Un employé affecté à un rôle de gardien ne peut pas cumuler ce poste avec un rôle de responsable. On prendra en compte le fait qu'une affectation de responsable peut être ajoutée mais aussi modifiée.
  - Des animaux ne peuvent pas être placés dans une cage non gardée. On prendra en compte le fait que des animaux peuvent être ajoutés dans un cage mais aussi déplacés dans une autre cage.

Use case: Le Zoo

- Implémentation
  - Un employé affecté à un rôle de gardien ne peut pas cumuler ce poste avec un rôle de responsable. On prendra en compte le fait qu'une affectation de responsable peut être ajoutée mais aussi modifiée.

```
create or replace trigger Q2_3
before insert or update of NomE on LesResponsables
for each row
declare
   nb integer;
begin
   select count(*) into nb from LesGardiens where NomE=:new.NomE;
   if (nb > 0) then
      raise_application_error(-20008, 'un gardien ne peut pas etre egalement
responsable');
   end if;
end;/
```


Use case: Le Zoo

- Implémentation
  - Des animaux ne peuvent pas être placés dans une cage non gardée. On prendra en compte le fait que des animaux peuvent être ajoutés dans un cage mais aussi déplacés dans une autre cage.

```
create or replace trigger Q2_2
before insert or update of noCage on LesAnimaux
for each row
declare
   g integer;
begin
   select count(*) into g from LesGardiens where noCage = :new.noCage;
   if (g = 0) then
      raise_application_error(-20002, 'ajout d''un animal dans une cage non gardee');
   end if;
end;
/
```


Use case: Le Zoo

- Ressortir les contraintes applicatives:
  - Des animaux de type différent ne peuvent pas cohabiter dans une même cage. On prendra en compte le fait que des animaux peuvent être ajoutés dans une cage mais aussi déplacés dans une autre cage.
  - Si une cage gardée se retrouve sans animaux, un et un seul gardien peut lui être affecté (nettoyage). On prendra en compte le fait que des animaux peuvent être supprimés mais aussi déplacés dans une autre cage.

Use case: Le Zoo

- Implémentation:
  - Des animaux de type différent ne peuvent pas cohabiter dans une même cage. On prendra en compte le fait que des animaux peuvent être ajoutés dans une cage mais aussi déplacés dans une autre cage.

```
create or replace trigger Q2_1
after insert or update of noCage on LesAnimaux
declare
   nb integer;
begin
      select count(*) into nb from (
         select noCage, count(distinct type) from
            LesAnimaux
            group by noCage
            having count (distinct type) > 1  );
 if (nb > 0) then
          raise_application_error(-20001,
'cohabitation impossible entre animaux
de types different');
       end if;
end;
/
```


Use case: Le Zoo

- Implémentation:
  - Si une cage gardée se retrouve sans animaux, un et un seul gardien peut lui être affecté (nettoyage). On prendra en compte le fait que des animaux peuvent être supprimés mais aussi déplacés dans une autre cage.

```
create or replace trigger Q2_4
after delete or update of noCage on LesAnimaux
declare
   nb integer;
begin
select count(*) into nb from (
         select c.noCage, count(distinct nomE) from
            LesCages c, LesGardiens g
            where c.noCage = g.noCage
               and c.noCage not in (select distinct noCage from LesAnimaux)
            group by c.noCage
            having count (distinct nomE) > 1
   );
 if (nb > 0) then
          raise_application_error(-20003, 'un
gardien maximum pour une cage vide');
       end if;
end;
/
```


Use case: Le Zoo

- Ressortir les contraintes applicatives:
  - Un gardien ne peut pas être retiré de la surveillance d'une cage si les animaux qu'elle contient se retrouvent non gardés. On prendra en compte le fait que des gardiens peuvent être retirés mais aussi affectés à une autre cage.
  - Un employé ne peut pas se retrouver responsable d'une allée dont toutes les cages sont vides. On prendra en compte le fait que des animaux peuvent être supprimés d'une cage mais aussi déplacés dans une autre cage.
  - Lorsqu'un gardien voit l'une de ses affectations modifiées, son acienne affectation doit être conservée dans la table LesHistoiresAff.

Use case: Le Zoo

- implémentation:
  - Lorsqu'un gardien voit l'une de ses affectations modifiées, son acienne affectation doit être conservée dans la table LesHistoiresAff.

```
create or replace trigger Q2_7
before delete or update of noCage on LesGardiens
for each row
declare
   g integer;
begin
   insert into LesHistoiresAff values(:old.noCage, :old.nomE, sysdate);
end;
/
```


Use case: Le Zoo

- Fonctionnalité: Ajouter des animaux à une cage.
  - La cage peut être vide
    - Elle peut ne pas avoir de gardien
  - Les animaux peuvent être incompatibles
  - Choix d'implémentation:
    - Laisser les exceptions remonter

    => Nombreux rollback
    - Faire des tests en amont

    => Assurer l'isolation de la transaction

Optimisation de requête (Documents de cours)

Fabrice Jouanot

Optimisation par règles Règles de transformation (1/3)

1) $\sigma_{(p \wedge q \wedge r)}R = \sigma_p(\sigma_q(\sigma_r(R)))$
2) $\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$ (commutativité)
3) $\pi_L \pi_M \ldots \pi_N(R) = \pi_L(R)$
4) $\pi_{A1,\ldots,An}(\sigma_p(R)) = \sigma_p(\pi_{A1,\ldots,An}(R))$ si $p \in \{A1 \ldots An\}$
5) $R *_p S = S *_p R$ et $R \times S = S \times R$
6) $\sigma_p(R *_r S) = \sigma_p(R) *_r S$ si $p \in$ Attribut(R)

   et

   $\sigma_{p \wedge q}(R *_r S) = \sigma_p(R) *_r \sigma_q(S)$

   si $p \in$ Attribut(R) et $q \in$ Attribut(S)

Optimisation par règles Règles de transformation (1/3)

7) $\pi_{L1 \cup L2}(R *_r S) = (\pi_{L1}(R)) *_r (\pi_{L2}(S))$
   si $L1 \in$ Attr(R) et $L2 \in$ Attr(S)

   $\pi_{L1 \cup L2}(R *_r S) = \pi_{L1 \cup L2}((\pi_{L1 \cup M1}(R)) *_r (\pi_{L2 \cup M2}(S)))$
   si $M = M1 \cup M2$ avec $M1 \in$ Attr(R), $M1 \notin L1$, $M \in R$
   $M2 \in$ Attr(S), $M2 \notin L2$, $M \in R$

8) $R \cup S = S \cup R$ et $R \cap S = S \cap R$
9) $\sigma_p(R \cup S) = (\sigma_p(R) \cup \sigma_p(S))$ (idem avec $\cap$ et $-$)
10) $\pi_L(R \cup S) = (\pi_L(R) \cup \pi_L(S))$

Optimisation par règles
Règles de transformation (1/3)

11) $(R * S) * T = R * (S * T)$

$(R \times S) \times T = R \times (S \times T)$

$(R *_p S) *_{q \wedge r} T = R *_{p \wedge r} (S *_q T)$ si $q \in Attr(S) \cap Attr(T)$

si $r \in Attr(R) \cap Attr(T)$

12) $(R \cup S) \cup T = S \cup (R \cup T)$ idem avec $\cap$

Optimisation par règles
Heuristiques

- H1: Appliquer les opérations de sélection au plus tôt
  - Réduire la cardinalité des tables pour optimiser les opérateurs suivants
  - Généralement règle 1, puis 2,4,6 et 9 pour déplacer les sélections
- H2: Combiner les produits cartésiens avec des sélections pour construire des jointures
- H3: Utiliser l'associativité des opérateurs binaires pour que les opérations réduisant la cardinalité (les plus sélectives) s'appliquent en premier
  - Règles 11 et 12
  - L'ordre des jointures peut devenir important
- H4: Appliquer les projections le plus tôt possible
  - Réduire la taille des tuples pour optimiser la mémoire
- Ne calculer qu'une seule fois les expressions redondantes.

Optimisation par règles
Exemple

- On part d'une requête déclarative

```
Select i,numPropriete, i.rue
from client c, visite v, proprieteALouer i
where c.typePref='Appart' and c.numclient=v.numclient and
        v.numPropriete=i.numPropriete and c.loMax>=i.location and
        c.typePref=i.type and i.numPropritaire='CP93';
```

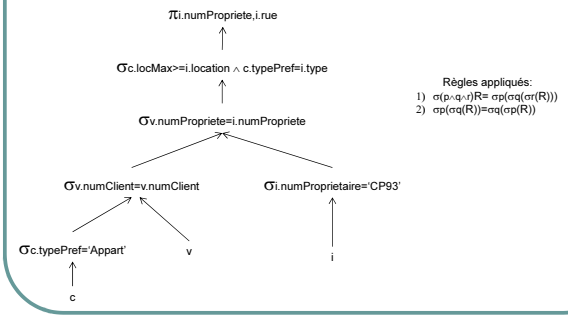
- Et sa version algèbrique avant optimisation

$\sigma\ \pi \wedge \cap \cup\ \times * \in$

$\pi_{i.numpropriete,i.rue}(\sigma_{c.typePref='Appart' \wedge c.numclient=v.numclient \wedge v.numPropriete=i.numPropriete \wedge c.locMax>=i.location \wedge c.typePref=i.type \wedge i.numProprietaire='CP93'}((C \times V) \times P))$

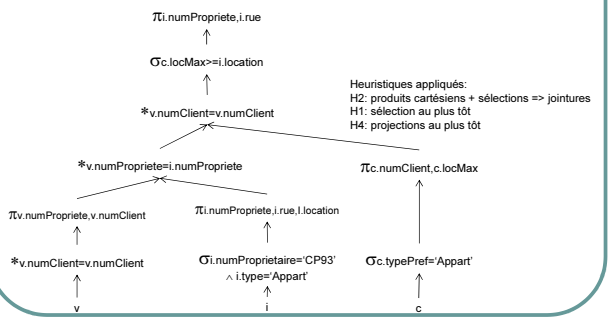
Optimisation par règles
Exemple

- On suppose qu'il existe moins de biens pour CP93 que de locations potentiels



Optimisation par règles
Exemple

- On suppose qu'il existe moins de biens pour CP93 que de locations potentiels



SGBDOO et SGBDRO

Du relationnel à l'objet…

Plan du cours

- Modèle OO et SGBDOO
  - Pourquoi ?
  - Définition et concepts d'un SGBDOO
  - Requête OO
- Modèle RO et SGBRO
  - Introduction et comparaison OO
  - TAD
  - Les, collections, références et méthodes
  - Requête RO, notion d'Oid

Points faible du Relationnel

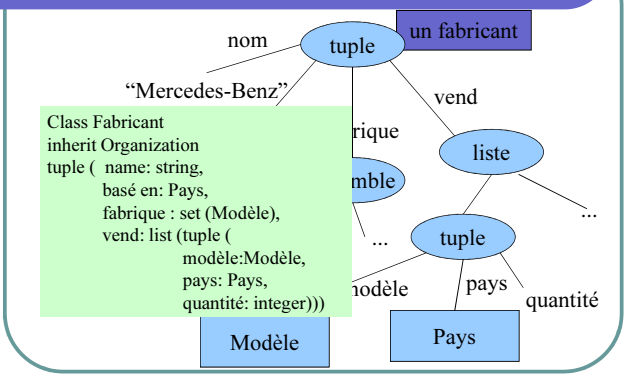
- Modèle
  - structure de données trop simple
  - pas de niveau conceptuel
  - intégration difficile avec les langages de programmation objets sans canevas logiciel
- Sur un SGBDR deux opérations
  - Join (lent)
  - Select (rapide)
- Cas des objets complexes
  - répartis sur plusieurs tables
    - $\Rightarrow$ beaucoup de jointures
  - mauvaises performances (sans optimisation)

Avantage de l'objet

- Classe = ensemble d'objets permanents de même structure ("population")
- Représentation fidèle du monde réel

  => liens sémantiques :
  - hiérarchie de généralisation = inclusion des populations
  - association
  - lien de composition conceptuel
- Requête = navigation dans un graphe d'objets

Un exemple d'objet complexe



SQL vs. OO

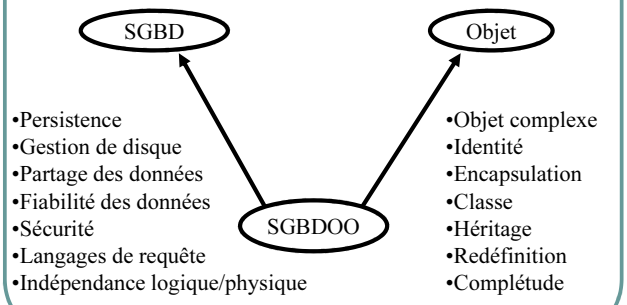
- "Modéles et quantité vendus par Mercedes en France"
  - en SQL

```
select Modèle.nom, ventes.quantité
from Fabricant, Modèle, ventes
where ventes.Fabric = Fabricant.ID.
   and ventes.Mod = Modèle.ID
   and Fabricant.nom = "Mercedes"
   and ventes.pays = « France"
```

  - en OQL

```
select c.vend.modèle.nom, c.vend.quantité
        from c in les_fabricants
        where c.nom = "Mercedes"
                and c.vend.pays="France"
```


SGBDOO : une définition



Identité d'objets

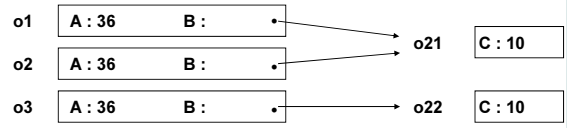
- Identifier les objets indépendamment de leur valeur
  - Chaque objet possède une identité qui ne peut être changée durant toute sa vie.
  - L'identification des objets est gérée par le système (allocation, …).
  - Les objets peuvent être partagés par d'autres objets.
  - Les objets peuvent être cycliques.
- oid (object identifier) géré par le SGBDOO
  - Unique, permanent, immuable
- objet = (oid, valeur)

  => test d'identité = =

  test d'égalité = égalité en surface

  = * égalité en profondeur

Tests d'identité / d'égalité



VRAI	FAUX	
o1.B = = o2.B	o1 = = o2	identité
o1 = o2	o1 = o3	égalité surface
o1 =* o3		égalité profonde
o21 =* o22	o21 = = o22	

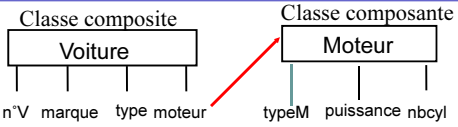
Notion de classe

- Une classe décrit les propriétés partagées par un ensemble d'objets similaires
  - propriétés statiques (structure de données)
  - propriétés dynamiques (méthodes)
- Une classe est composée de 2 parties :
  - l'interface : comment utiliser ses méthodes ?
  - l'implémentation : structure de données interne et code des méthodes
- Attention une classe n'a pas forcément de population (extension)

Exemple de classe

```
Class Employé
    type tuple ( nom : string,
                 emploi : Emploi,
                 expérience : integer,
                 photo : image )
        method
            public augmente_expérience (années: integer) : integer ,
            public affiche,
            public salaire : integer
    end;
```

Lien de composition



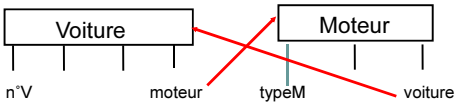
- **Class** Voiture
  tuple (n°V : **int**,
  marque : **char** 25,
  type : **char** 20,
  moteur : Moteur)

Class Moteur
tuple (typeM : **char** 20,
puissance : **int**,
nbcyl : **int**)

attribut référence
sa valeur = oid d'un moteur
ou NUL

Contraintes de composition

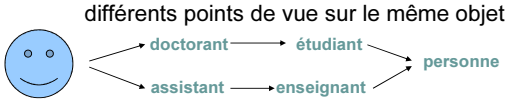
- objet composant partagé / non partagé
- objet composant dépendant / non dépendant
- lien inverse



- **cardinalités :**
  - **minimale, maximale**
  - **inverses**

Hiérarchie de généralisation

- Objectifs :
  LPOO : héritage
  BD : représentation du monde réel :
  différents points de vue sur le même objet



- **Lien de généralisation/spécialisation, IS-A:**
  - **inclusion de population**
  - **héritage des attributs et des méthodes**

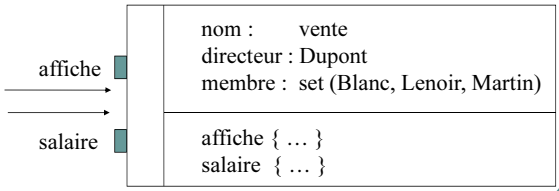
Exemple de hiérarchie

```
Class Emploi
      type tuple ( nom : string,
                   expérience : integer ),
      method affiche,
                   salaire : integer
end;

Class EmploiVente
      inherit Emploi
      type tuple ( bonus : real )
      method salaire : real
end;

Class EmploiTempsPartiel
      inherit Emploi
      type tuple ( durée : integer,
                   bonus : real ),
      method salaire : real
end;

Class EmploiVenteTempsPartiel
      inherit EmploiVente, EmploiTempsPartiel
      method salaire : real
end;
```


Encapsulation

Les utilisateurs des objets (développeurs) ne doivent rien savoir à propos de leur implémentation : seule l'interface est publique

Partie interface *Partie implémentation*



Redéfinition

- Avec une programmation classique :

```
for each emploi in LesEmplois do
    case of type (emploi)
        Emploi : salaire = salaire_standard (emploi)
        EmploiTempsPartiel : salaire = salaire_temps_partiel (emploi)
    end_case;
display (salaire);
```

- Avec une programmation objet :

```
for each emploi in LesEmplois do
    display (salaire);
```


Les collections en OO

- Collections d'objets :
  integer, n-uplets, nested collections ,….
  Exemples : LesNoms : unique set (string);
  class Département
  type tuple (nom : string;
  directeur : Employé,
  membres : set (Employé))
  LesDépartements : set (Département);

- Il est possible de définir des set, bag, list, array …
- Ces collections supportent toutes les opérations dédiées aux ensembles : union, intersection, parcours, sélection, jointure, …

Langage de requête

- Propriétés importantes :
  - supporter toutes sortes d'objets complexes
  - naviguer simplement à travers la BDO
  - tirer parti des méthodes des objets
  - construire des objets complexes comme résultat
  - permettre l'optimisation des requêtes
- Exemple de requête OQL

```
select tuple ( nom  : e.titre, salaire : e.salaire, directeur : dept.directeur.nom)
from   dept in LesDépartements, e in dept.membre
where dept.directeur.emploi.salaire > 100 000
   and e.emploi.nom = « Assistant »
```

RO: Modèle de données NF2 (*Non First Normal Form*)

- Structure des objets complexes dans un contexte relationnel.
- Cohabitation des notions de table (données en 1ère FN) et de données de type complexe (ou objet).
- Une table en NF2 peut contenir un attribut composé d'une liste de valeurs et/ou un attribut composé de plusieurs attributs.
  - Meilleur niveau conceptuel,
  - Plus besoin d'aplatir la réalité,
  - Moins de jointures.

Exemple : table Personnes

nom	{prenoms}	Date_naissance	{voitures}			…
	prenom		modele	annee	no	
Duran	Pierre	16-05-1963	2ch	1975	128	
	Paul		Mégane	2008	371	
	Jacques					
Piona	Marie	29-02-1994	Twingo	1999	17	
	Jeanne					

Concepts SGBDOO vs SGBDRO

Concepts SGBD
- Persistance
- Gestion du disque
- Partage des données
- Fiabilité
- Sécurité
- Langages de requête
- Indépendance Logique / Physique

- Objet (+ou-)
- encapsulation (+ou-)
- objet complexe
- identité d'objet
- Classe (+ou-)
- héritage
- redéfinition

- Persistance des objets complexes
- Gestion de large collection de données (+ou-)
- Langage de requête pour les objets

Le SGBDRO de Oracle

- Pas de classe.
  - Il faut créer un type abstrait de données (TAD), puis une table représentant ou utilisant ce type.
  - Implémentation des méthodes liée au TAD
- Héritage depuis oracle 9i
- Surcharge des noms de méthodes possible
  - 2 méthodes d'1 TAD avec le même nom et des paramètres différents => Oracle choisit à l'exécution.
  - Surcharge des opérateurs Oracle interdite.
- Encapsulation simulée à travers un package PL/SQL.

Concrètement

2 extensions par rapport au modèle relationnel :

- I - Les *TAD* (types abstraits de données),
  1. Composition de types,
  2. Collections,
  3. Références,
  4. Méthodes.

  } Types complexes

- II - Les *oid* via les pointeurs.

I – Les TAD

- Dans tout TAD peuvent être ajoutées des **méthodes** (procédures ou fonctions PL/SQL) pour manipuler les objets du TAD, mais **pas de contraintes**.
- Impossible de créer une instance d'un TAD directement, il faut créer et passer par une table !

  => Initialisation des instances impossible (pas de valeur par défaut associée à un TAD).
- Navigation à travers les instances des TAD avec la notation de chemin et un alias sur chaque table.

Instances de TAD

- Une table d'un TAD est une **table d'objets**.
  - Uniquement avec la commande :
    create table *nom_table* of *nom_TAD*;
  - Toute instance d'une telle table possède un oid unique, ce sont des n-uplets objet.
  - La portée de cet oid est globale.
- Attention,
  - Les autres tables (non directement construite sur un TAD) ne sont pas des tables d'objets.
  - Les instances des autres tables n'ont pas d'oid.
    - portée de l'oid locale à la table.

I - Les TAD : 1 – Composition de types

- Un TAD en RO peut être perçu comme :
  - Un nouveau type d'attribut simple défini par l'utilisateur (concerne 1 seul attribut),

  Ou
  - Une structure de données partagées.

A- TAD perçu comme un nouveau type d'attribut

- Cela concerne un seul attribut,
- Créé à partir d'un type d'Oracle,
- Permet d'enrichir les types prédéfinis d'oracle,
  - En ajoutant des méthodes de manipulation, avec la commande CREATE TYPE,
  - En ajoutant des contraintes de domaines, exemple : CREATE TYPE typemot IS varchar2 [NOT NULL] create table code (mot typemot(8), code typemot(25));

B- TAD perçu comme une structure partagée

- Types utilisables par un ou plusieurs autres types (composition) et par une ou plusieurs tables.
- Permet de construire une structure complexe,
  - Soit directement,
  - Soit indirectement.

TAD perçu comme une structure partagée

- Construction directe :
  1. Création d'un type,
  2. Création d'une table de ce type (=> table d'objets).
- Construction indirecte :
  1. Création d'un type *t* ayant au moins 1 attribut de type composé défini par l'utilisateur (i.e. : *t* utilise un TAD dans sa définition),
  2. Création d'une table de type *t* (=> table d'objets).

  Ou
  2. Création d'une table ayant au moins 1 attribut de type composé défini par l'utilisateur (=> **pas** table d'objets).

Composition directe de types

Create type tvoiture as object
(modele varchar2(15),
annee date, no integer)

/ *-- Remarque : après la création d'un TAD et uniquement dans ce cas, '/' remplace le ';' pour terminer un ordre PL/SQL.*

Create table voitures ***of* tvoiture**;
-- voitures est une table d'objets.

TAD Tvoiture :

Tvoiture		
Modele	Annee	No

Exemple de composition directe de TAD

TAD Tvoiture :

Tvoiture		
Modele	Annee	No

L'insertion des données peut se faire comme d'habitude en SQL :
Insert into voitures values ('2ch', '1975', 128);

Table voitures :
(ordre SQL :
select *
from voitures)

Modele	Annee	No
2ch	1975	128
Mégane	2008	371

Composition indirecte de types

Create type Tpersonne as object
(nom varchar2(15), prenom varchar2(15),
date_naissance date, **voiture Tvoiture**)

/

Create table personnes *of* Tpersonne;
-- personnes est une table d'objets.

TAD Tpersonne :

Tpersonne					
Nom	prenom	Date_naissance	Voiture		
			modele	annee	No

Exemple de composition indirecte de TAD

A chaque TAD créé est associé un constructeur (même fonctionnement qu'en BDOO) du même nom que le TAD *(obligation de valuer tous les attributs).*

⇒ Insertion des données dans une table avec TAD :

Insert into personnes values (Tpersonne('Duran', 'Pierre', '16-05-1963', Tvoiture('2ch', '1975', 12)));

facultatif

obligatoire

Table personnes :
(ordre SQL :
select *
from personnes)

Nom	prenom	Date_naissance	Voiture		
			modele	annee	No
Duran	Pierre	16-05-1963	2ch	1975	12
Piona	Marie	29-02-1994	Twingo	1999	17

Construction indirecte de table

Création d'une table utilisant un TAD :

Create table personnes
(nom varchar2(15), prenom varchar2(15),
date_naissance date, **voiture Tvoiture**)

/ -- personnes **n'est pas** une table d'objets.

Insertion de données dans une table utilisant un TAD :

Insert into personnes values ('Duran', 'Pierre',
'16-05-1963', Tvoiture('2ch', '1975', 12));

Valeurs par défaut

- Il est possible de définir des valeurs par défaut pour un TAD dans sa définition :

Create table personnes
(nom varchar2(15), prenom varchar2(15),
date_naissance date,
voiture Tvoiture DEFAULT voiture('2ch', '1975', 12))

TAD interdépendants

- Si références mutuelles des types, impossible de créer un TAD complet et correct sans que l'un des deux ne soit déjà créé => problème.
- Exemple :
  TAD *Tpersonne* utilise *Tvoiture*, et
  TAD *Tvoiture* utilise *Tpersonne*.
- Solution de Oracle : déf. de TAD incomplet.
  Syntaxe : create type nom_type
  / -- ni attribut, ni méthode déclarés.
- TAD complété plus tard, mais référençable dans l'état.

Les TAD : 2 – Les collections

- Exprimer qu'une personne possède plusieurs prénoms ou plusieurs voitures dans une même table.
- 2 possibilités de stocker des collections :
  1. Varray : dans une valeur
  2. Nested Table : dans une sous-table

Personnes

Nom	liste-prénoms	age	voitures
'Liera'	('Pierre','Paul')	32	Id1
'Liera'	('Zoé','Marie')	27	Id2

1 valeur

1 table

ID	Tvoiture		
Id1	'2ch'	'1975'	12
Id1	'206'	'2000'	3
Id2	'2ch'	'1975'	12
Id1	'607'	'2002'	1

Varray vs. Nested Table

	Varray	Nested Table
Relation d'ordre entre les éléments de la collection	oui	non
Stockage	1 valeur ou BLOB	1 table
Type des éléments	Type objet ou scalaire*	
Requête	avec l'opérateur TABLE	
Manipulation	non	oui
Optimisation	non	index
Imbrication de collections	non (mais ruse possible, avec Références)	

*types prédéfinis Oracle, y compris REF

Stocker des collections liées

- Construction d'un Varray :
  1. Création du type de la collection, avec :
     create type *nom* as Varray (nb_max) of type_elt,
  2. Création du type contenant la collection, Tpersonne par exemple (ou création directe de la table)
  3. Création de la table de ce type complexe (Tpersonne).
- Manipulation d'un Varray :
  Stocké et manipulé comme une valeur atomique
  ⇒ En 1ère forme normale
  ⇒ Manipulation de la liste, pas des éléments
  ⇒ Pas de jointure

Exemple de collections liées

Plusieurs prénoms pour une personne :

```
Create type liste-prenom as Varray(10) of varchar2(15)
-- Varray(10) : Varray est le mot-clé pour définir une collection et 10 le
     nombre maximum d'éléments de cette liste.
--  type des éléments de la collection indiqué après le mot-clé 'of'.
/
Create type or replace Tpersonne as object
        (nom varchar2(15), Date_n Date,
        prenoms liste_prenom, voiture Tvoiture)
/
Create table personnes of Tpersonne;
    -- personnes est une table d'objets.
```


Exemple de table avec collections liées

- La encore on utilise le constructeur pour les listes :

```
Insert into personnes values (Tpersonne('Duran',
   liste_prenom('Pierre', 'Paul', 'Jacques'),
        '16-05-1963', Tvoiture('2ch', '1975', 12)   );
```

Table personnes :
(ordre SQL : select * from personnes)

Nom	{prenoms}	Date_naissance	Voiture		
	Liste_prenom		modele	annee	No
Duran	Pierre	16-05-1963	2ch	1975	12
	Paul				
	Jacques				
Piona	Marie	29-02-1994	2ch	1975	12
	Jeanne				

Et moins simple

- Une personne peut avoir plusieurs voitures.

nom	{liste_prenoms}	Date_naissance	{voitures}			…
	prenom		modele	annee	no	
Duran	Pierre	16-05-1963	2ch	1975	128	
	Paul		Mégane	2008	371	
	Jacques					
Piona	Marie	29-02-1994	Twingo	1999	17	
	Jeanne					

Stocker des collections libres

Construction d'une Nested Table :

1. Création du type de la collection, avec :
   create type *nom* as Table of type_elt,
2. Création du type contenant la collection puis 3, ou création directe de la table, par exemple Personnes.
3. Création de la table du type 2 (s'il y a lieu).

⇒ Génération automatique d'un identifiant superficiel Nested_Table_Id pour chaque collection

⇒ Possibilité de créer un index dans la Nested Table

Attention : Une Nested Table n'est pas une table objet

⇒ Pas d'oid associé à un élément de collection

Créer des collections libres

```
Create type Tvoiture as object
                (modele varchar2(15),
                annee date, no integer)
/
Create type ens_voiture as Table of Tvoiture
/
Create table Personnes (nom varchar2(15),
          prenoms liste_prenom,
          date_naissance Date,
          voitures ens_voiture);
    Nested table voitures Store As ToutesVoitures
```

Personnes

Nom	liste-prénoms	age	voitures
'Liera'	('Pierre','Paul')	32	Id1
'Liera'	('Zoé','Marie')	27	Id2

Nested_table_Id	Tvoiture		
Id1	'2ch'	'1975'	12
Id1	'206'	'2000'	3
Id2	'2ch'	'1975'	12
Id1	'607'	'2002'	1

index

Manipuler des collections libres

Une collection dans une Nested Table peut être

- Insérée,
- Supprimée,
- Mise à jour : les éléments dans la nested table peuvent être insérés, supprimés ou maj

```
insert into Personnes values('mila',
        liste_prenom('Karim', 'Amin'), '16-03-75',
        ens_voiture(Tvoiture('2ch', 1975, 128),
                Tvoiture('Mégane', 2008, 179)));
insert into Table(select p.voitures
                        from Personnes p where p.nom= 'mila')
                values ('206', '2000', 3)
```


Manipuler ses éléments

- Supprimer une collection libre :
  Update Personnes SET voitures = NULL where nom= 'mila'
- Re-créer : avec le constructeur (sinon, aucune opération possible)
  - Créer un ensemble vide :
    Update Personnes SET voitures = ens_voiture() where nom= 'mila'
  - Créer un singleton :
    Update Personnes
    SET voitures = ens_voiture(Tvoiture('Mégane', 2008, 179))
    where nom= 'mila'
- **Attention :** une opération sur 1 élément d'1 collection dans une Nested Table pose un verrou sur son propriétaire
  => 1 opération à la fois dans une collection !!!!!!

Requête avec collections

- Create type liste_voiture as Varray(10) of Tvoiture
  /
  Create table personnes (nom varchar2(15),
  prenoms liste_prenom,
  date_naissance Date,
  voitures liste_voiture);

  insert into personnes values('mila',
  liste_prenom('Karim', 'Amin'), '16-03-75',
  liste_voiture(Tvoiture('2ch', 1975, 128),
  Tvoiture('Mégane', 2008, 179)));
- On veut exécuter des requêtes sur les collections de voitures, telle No des voitures des personnes.

Parcourir une collection

- Pour parcourir des collections libres ou liées : voire **une** collection comme **une** table.

⇒ Utilisation de l'opérateur **TABLE**
(liste ou 'sous-table' -> table).

⇒ Puis utilisation d'un alias pour parcourir les tables ainsi obtenues.

⇒ Parcours de chaque collection de chaque n-uplet.
Ici, parcours (sélection) de chaque collection 'prénoms' et 'voitures' de chaque personne de la table personnes.

Exemple de parcours de listes

- Numéro des voitures des personnes :

```
Select v.No
  from personnes p, Table (p.voitures) v;
```

- Numéro des voitures de Piona :

```
Select v.No
  from personnes p, Table (p.voitures) v
  where p.nom = 'Piona';
```


Les TAD : 3 – Les références

- Référencer une voiture dans une table sans l'expliciter à chaque fois.
  - Attention, impossible de référencer une collection !
- Possibilité de référencer des données via des pointeurs.
  - Créer le TAD t dont on veut référencer les instances,
  - Créer la table contenant les instances de t,
  - Puis :
    - Créer le TAD qui référence t (mot-clé REF),
    - Créer la table associée.

  ou
    - Créer directement la table.

Exemple de références

Référencer la voiture d'une personne :

```
Create type Tvoiture as object (modele varchar2(15),
                                annee date, No integer)
/
Create table voitures of Tvoiture;
Create or replace type Tpersonne as object
      (nom varchar2(15),
      prenoms liste_prenom,
      -- Rq : impossible de référencer une collection !
      -- prenoms REF liste_prenom est illégal.
      date_naissance Date,
      voiture REF Tvoiture)
/
Create table personnes of Tpersonne;
```

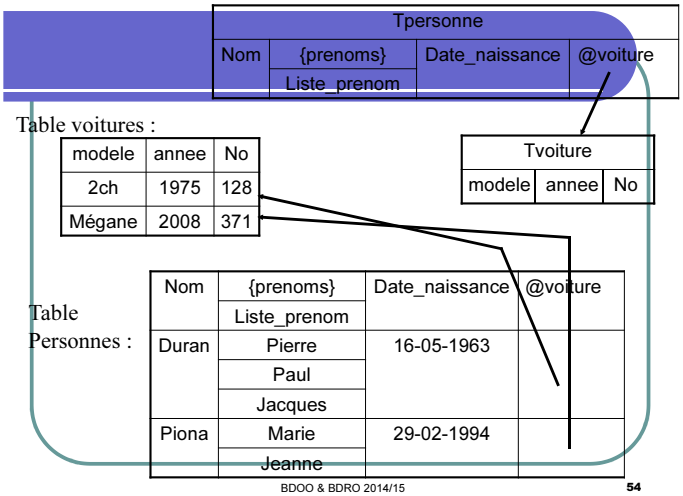

Insertion avec références

- Pour insérer un n-uplet dans la table personnes, il faut référencer la voiture de cette personne dans la table voitures

⇒ Récupérer l'adresse de cette voiture pour la placer dans la table personne.

⇒ Utilisation de l'opérateur REF qui renvoie l'adresse d'un n-uplet dans une table (REF : n-uplet -> pointeur).

⇒ Insert into personnes values ('Duran',
liste_prenom('Pierre', 'Paul', 'Jacques'),
'16-05-1963',
(select **REF**(v) from voitures v
where v.modele = '2ch'));



Requêtes avec références

- Résultat de la requête : select * from personnes;

Nom	{prenoms}	Date_naissance	@voiture
	Liste_prenom		
Duran	Pierre	16-05-1963	03F43970135A39847C… -- adresse du pointeur. Ne veut rien dire !
	Paul		
	Jacques		

- Pour obtenir la valeur référencée :

⇒ Utilisation de l'opérateur DEREF (pointeur -> valeur)

Select **DEREF**(p.voiture), p.nom from personnes p
where *p.voiture.année* < 1990;

Exemple – suite : Création directe de la table

Référencer la voiture d'une personne :

```
Create type Tvoiture as object
      (modele varchar2(15),
       annee date, No integer)
/
Create table voitures of Tvoiture;
 -- voitures est une table d'objets.
Create table personnes
      (nom varchar2(15), prenoms liste_prenom,
      date_naissance Date, voiture REF Tvoiture);
 -- personnes n'est pas directement construite avec un
TAD, donc n'est pas une table d'objets.
```


Mise à jour avec références

- Soit la table suivante :
  Create type Tadresse as object (ville varchar2(15), rue varchar2(15))
  /
  Create table personnes
  (nom varchar2(15), prenoms liste_prenom,
  adresse Tadresse, date_naissance Date,
  voiture REF Tvoiture);
- Modifier l'adresse :
  update personnes p set p.adresse.rue = 'Comédie'
  where p.adresse.ville = 'Montpellier';
- Changer de voiture :
  update personnes p set p.voiture =
  (select REF(v) from voitures v where v.modele = 'Mégane')
  where p.voiture.annee = 1975;
- **Attention** : modifier une voiture à travers une personne est impossible !

Référencer du vide

- Pour tester l'existence d'une instance référencée :
  Utilisation du prédicat IS DANGLING.
- Exemple :
  update Personnes set voiture = NULL
  Where voiture IS DANGLING

Références croisées

- Comme d'habitude, on commence par créer un type vide pour être référencé dans un second type et on modifie le premier type ensuite. On crée les tables en dernier.

```
Create type Tvoiture /
Create type Tpersonne as object
      (nom varchar2(15), prenoms liste_prenom,   date_naissance Date,
   voiture REF Tvoiture) /
Create type Tvoiture as object
      (modele varchar2(15), annee date, No integer,
      personne REF Tpersonne) /
Create table voitures of Tvoiture;
Create table personnes of Tpersonne;
```

Imbrication de collections

- Impossible dans l'absolu, mais des tableaux fixes (Varray) peuvent être imbriqués grâce aux références.

```
Create type tpanne as object
                (typep varchar2(20),
                datep date, detail varchar2(200)); /
Create type tpannes as Varray(10) OF REF tpanne; /
Create type tvoiture as object
                (modele varchar2(15), annee date, no integer,
                sespannes tpannes); /
Create type tvoitures as Varray(10) OF REF tvoiture;/
REMPLACE "Create type liste_voiture as Varray(10) of Tvoiture"
Create table pannes OF tpannes;
Create table voitures OF tvoitures;
Create table personnes (nom varchar2(15),
        prenoms liste_prenom, date_naissance Date, voitures tvoitures);
```


Les TAD : 4 – Les méthodes

- Déclaration des méthodes (signature) insérée après les attributs dans la déclaration des TAD

```
Create type [or replace] nom_type as object
        (att1 type1, …, atti typei,
        MEMBER signature_methode1, …,
        STATIC signature_methodej),
```

Avec

- MEMBER : méthode invoquée sur les instances du TAD,
- STATIC : méthode invoquée sur le TAD,
- signature_methodel : *nom_méthode* (var1 type_var1, …).
  - typel ou type_varl défini par l'utilisateur ou prédéfini
  - *nom_ méthode* différent de celui du TAD et de ses attributs.

Corps des méthodes

- Code des méthodes dans le corps du TAD.
- Syntaxe :
  Create or replace type body nom_type as …
- Contient uniquement le corps des méthodes.
- Si uniquement des attributs dans un TAD, corps inutile.
- Possibilité d'utiliser le paramètre SELF dans le corps des méthodes (idem qu'en OO)
  - Implicitement ou explicitement déclaré,
  - Si déclaration implicite :
    IN pour les fonctions, IN OUT pour les procédures.

Exemple de méthode

```
Create or replace type Tpersonne as object
        (nom varchar2(15),
        prenoms liste_prenom,
        date_naissance Date,
        voiture REF Tvoiture,
        MEMBER procedure ajoute_prenom (prenom_sup
                IN OUT Tprenom) )
   /
Create or replace type body Tpersonne as
   MEMBER procedure ajoute_prenom
                (prenom_sup IN OUT Tprenom)  IS
   Begin …..
   End;
End
   /
```


Appel de méthode

- Avec notion de chemin (comme pour les attributs)
- Chaînage des appels de méthodes possible
  - Exécution des méthodes de gauche à droite,
  - ⇒ Pas d'appel à droite d'un appel de procédure
  - ⇒ Si chaînage, la 1ère fonction doit renvoyer un objet pouvant être passé à la 2nde.
- Exemple :

```
Update personnes p set
        p.prenoms =
        p.ajoute_prenom('Clémentine')
        where p.nom = 'Piona';
```


II – Les oid

- Chaque instance d'une table d'objets possède un **identificateur d'objet** qui :
  - Identifie l'objet stocké dans ce n-uplet de façon unique,
  - Sert à référencer l'objet (seul oid de portée globale).
- Comme en OO, utilisation des chemins (notation pointée) pour naviguer à travers les objets référencés.
- Rappel :
  - Table d'objets : uniquement créée avec la commande ''create table nom_table of nom_tad;''.

Rappel : Extraction d'un oid

- Utilisation de la primitive REF.
- Exemple (avec 1 voiture par personne):

```
Update personnes p SET
   p.voiture = (select REF(v) from voitures v
                where v.modele='2ch')
   where p.nom = 'Piona';
```

- Utilisation identique dans les requêtes, insertions et suppressions.
- Permet d'assurer l'intégrité référentielle.

Sélection d'un objet

- Avec select : récupération d'ensemble de valeurs des attributs d'une table.
- Si on veut récupérer un ensemble d'objet :
  - Utilisation de l'opérateur VALUE
  - Exemple :
    - SQL> Select * from voitures;

```
'2ch'       ¦ '1975'        ¦ 128
'Mégane'    ¦ '2008'        ¦ 371
```

    - SQL> Select value(v) from voitures v;

```
Tvoiture('2ch', '1975', 128)
Tvoiture('Mégane', '1998', 371)
```

  - Idem avec insert et update (valeur/objet).

LMD et TAD

- Comme d'habitude en SQL,
  - Insert, Update, Delete, Select.
- Avec en plus les fonctions :
  - VALUE,
  - REF,
  - DEREF
- Le constructeur associé à chaque TAD,
- Le prédicat IS DANGLING,
- La clause RETURNING pour PL/SQL,
  ajout à la fin d'un insert ou update de la clause :
  Returning ref(alias_table) into nom_var
- L'opérateur TABLE.

LDD et TAD

- Créer : deja vu ….
- Modifier : Alter type
  Seules modifications possibles : les méthodes (insertions, suppressions, modifications)
- Supprimer : Drop type
- Rien n'est changé pour la modification et suppression des tables (y compris les tables d'objets).

Notion de Hiérarchie: Héritage

- Création de hiérarchie de types d'objet
  - Clause UNDER pour définir le sur-type d'un sous-type
  - Clause NOT FINAL pour autoriser un type à devenir sur-type
- Exemple

```
CREATE TYPE T-Personne AS OBJECT
( ninseeNUMBER ,
nom VARCHAR(18) ,
prénom VARCHAR(18) ,
adresse VARCHAR(200) )
NOT FINAL /

CREATE TYPE T-Etudiant UNDER T-Personne
( etab VARCHAR(18) ,
cycle VARCHAR(18) )
```


Notion de Hiérarchie: instances

- Une table T-Personne peut recevoir des tuples de type T-Personne et T-Etudiant

```
CREATE TABLE LesPersonnes OF T-Personne ;

INSERT INTO LesPersonnes VALUES (T-Person
        (11111111, 'TTT', 'SSSS', 'Rue des oiseaux, 1, MMMM') );
INSERT INTO LesPersonnes VALUES ( T-Etudiant
        (22222222, 'Muller', 'Annie', 'Rue du marché, 22, RRRR', 'UJF', 'master'));
```

- Exemple de requête
  - **VALUE(objet) IS OF type**: test intéressant relatif aux relations sur-type/sous-type

```
SELECT p FROM LesPersonnes p WHERE VALUE(p) IS OF T-Etudiant
```

Renvoie les objets de LesPersonnes qui sont de type T-Etudiant

Hibernate : ORM solution
(In Action, C. bauer & G. King
Ed. Manning)

Fabrice Jouanot

Plan du cours

- Introduction, Pourquoi un ORM ?
- Découvrir Hibernate
- Mapping classes – tables
- Gestion des objets persistants
- Querying avec Hibernate
- Transactions et concurrence

Introduction

Pourquoi un ORM ?

Introduction

- Il était une fois SQL
  - Modèle relationnel solide
  - Démocratisé dans tous les SI
  - Outil de définition de schéma
  - Outil d'interrogation
  - Gestion des transactions + concurrences
  - Fiabilité et robustesse
  - Performant et scalable

Introduction

- Et Java fut
  - Java est reconnu comme plateforme multi-tier (J2EE)
  - Langage à objets très utilisé et assez performant
- Lien Java – SQL
  - La BDR = source de persistance
  - Java = application
  - JDBC permet de relier les deux mondes
    - Les requêtes SQL sont écrites (à la main) dans le code
    - Les résultats sont parcourus dans une table "résultat"
    - Les objets d'entreprise sont instanciés à la main
  - Beaucoup de travail fastidieux de bas niveau sans relation direct avec les problèmes métiers !

Cohabitation objet - relationnel

- La persistance des objets restent un problème
  - La sérialisation n'est pas une solution: accès tout ou rien (sauvegarde de l'état du graphe d'objets)
  - Le modèle relationnel ne peut pas capturer directement les paradigmes du modèle à objets
- Le modèle relationnel reste LA solution de persistance, mais comment résoudre
  - Le problème de granularité
  - Le problème d'héritage (sous type)
  - Le problème d'identification
  - Le problème d'associations
  - Le problème de navigation dans un graphe d'objets

Paradigm mismatch

- On peut avoir l'illusion que tout se passe bien: exemple de factures d'un utilisateur

```
public class User {
        private String userName;
        private Set billingDetails;
        …}

Create table USER(
        username varchar(15) not null
primary key,
        …);
```

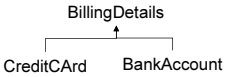
```
public class Billingdetails {
        private String accountNumber;
        private User user;
        …}
Create table billingdetails(
        accountnumber varchar(10) not
null primary key,
        username varchar(15) foreign key
references user(username)
        …);
```

- Les problèmes apparaissent si on introduit un objet Adresse !

Le problème de granularité

- lié à la limite de taille des objets manipulés: exemple de l'adresse (pour un utilisateur)
  - Une classe Adresse en Java qui peut:
    - être décomposée en des éléments à granularité plus fins (numéro, rue, ville, etc.)
    - être référencée dans différentes objets
  - Une table Adresse n'est pas possible (règles de normalisation)
    - utiliser un type utilisateur Adresse (extension SQL)
    - utiliser des attributs associés à l'utilisateur (peu flexible)
- Le problème est facile à résoudre (connaissance de l'application)

Le problème d'héritage



- Java permet de définit aisément cette relation de sous-typage
  - un simple lien d'héritage
  - qui autorise le polymorphisme
- Une BD SQL ne permet pas directement ce type de modélisation
  - pas de sous-type construit par héritage (ou SGBDRO)
  - aucune notion de polymorphisme (selon SGBDRO)

Le problème d'identification

- Le modèle à objet est plus riche en matière d'identité
  - identité d'objets (même emplacement mémoire)
  - égalité d'objets (même objet de surface, opérateur equals())
- Une BD SQL se limite à la définition d'une clé primaire (ou identifiant)
  - le choix d'une clé est difficile (ou OID dans SGBDRO)
  - la définition de clé de remplacement est fréquent (chaîne de caractères = mauvais candidat)

Le problème d'associations

(exemple des factures)

- En java une association est une référence vers un ensemble d'objets
  - Factures et utilisateurs sont stockés indépendamment
  - la relation est explicitement bidirectionnel
- En BD SQL l'association est modélisé par les clés étrangères (sur des clés primaires)
  - Factures et utilisateurs sont stockés dans leur propre table
  - la relation n'est pas bidirectionnel: one-to-many ou one-to-one
  - Une relation bidirectionnelle implique la création d'une table de liaison : artificielle et sans intérêt au niveau métier !

Le problème de navigation

- En java, il est naturel de naviguer dans le graphe d'objets pour construire les objets résultats
  - notation chainée (aUser.getBillingdetails().getAccountNumber())
  - pas de surcoût à la navigation
- En relationnel, la notion de navigation est remplacée par la notion de jointure

```
select *
from user u
left outer join billingdetails bd on bd.user_id=u.user_id
where u.user_id=123
```

  - le passage objet > BD SQL peut générer une requête de jointure pour chaque nœud du graphe (*n+1 selects problem*)

Modèle en couche : solution au surcoût ?

- 30% du code applicatif
  - gestion SQL/JDBC bas niveau
  - résolution des problèmes de paradigmes
- Modèle de persistance en couche
  - séparation des préoccupations
  - communication top to bottom
  - modèle en 3 couches (BD : en dehors de l'application)
    - Présentation (interface et interaction)
    - Métier (application)
    - Persistance (stockage et gestion des objets persistants)

Différentes approches

- Sérialisation : inadaptée
- SLQ/JDBC : très coûteuse pour mettre en concordance les 2 modèles
- EJB entity beans :
  - peu efficace en pratique
  - mauvaise granularité, pas de polymorphisme, très intrusif
- BDOO : quasi disparu, standard ODMG en déclin
- ORM : Object Relational Mapping
  - mapping transparent entre objets persistants (Java) et un schéma de BDR(O)
  - fournit une multitude de facilité
    - une API pour la persistance,
    - un langage de requête OO (même plusieurs)
    - une définition facile et précise du mapping
    - la prise en compte de l'aspect transactionnel
    - la garantie de performance (partiellement scalable)

Découverte d'Hibernate

Hello World

- Soit la classe suivante qui doit persister:

```
package hello
public class Message {
      private Long id;
      private String text;
      private Message nextMessage;
      private Message() {}
      public Message (String text) { this.text = text; }
      public Long getId() { return id;}
      private void setId(Long id) { this.id= id; }
      public String gettext() { return text; }
      public void setText(String text) { this.text = text; }
      public Message getNextMessage() { return nextMessage; }
      public void setNextMessage(Message nextMessage) {
            this.nextMessage = nextMessage; }
}
```


et son mapping

- Un fichier de mapping permet le lien entre une classe et la BDR (le fichier se nomme Message.hbm.xml)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-maping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD//EN"
      http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd>
<hibernate-mapping>
      <class
            name="hello.Message"
            table="MESSAGES"
            <id name ="id" column="MESSAGE_ID"> <generator class="increment"/></id>
            <property name="text"  column="MESSAGE_TEXT"/>
            <many-to-one name="nextMessage" cascade="all" column="NEXT_MESSAGE_ID"/>
      </class>
</hibernate-mapping>
```


Squelette de programme

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class FirstExample {
  public static void main(String[] args) {
    Session session = null;
    Configuration cf ;

    try{

// Nos exemples Ici !

    }catch(Exception e){
      System.out.println("catch !:"+e.getStackTrace());
    }
  }
}
```


Premiers pas

- Instanciation habituelle

```
Message message = new Message("Hello World");
System.out.println(message.getText());
```

- Session Hibernate

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Hello World");
session.save(message);
tx.commit();
session.close();
```

- Type de requête produite

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
    values (1,'Hello World', null);
```


Extraction d'un ensemble d'objets

- Comment récupérer des messages de la BD :

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages = newSession.find("from Message as m order by m.text asc");
System.out.println(messages.size() + " message(s) found:");
for( Iterator iter= messages.iterator(); iter.hasNext();) {
      Message message = (Message) iter.next();
      System.out.println( message.getText());
}
newTransaction.commit();
newSession.close();
```

- Aucune présence de code SQL (qui est généré dynamiquement à l'exécution)

Association entre objet persistant et un nouvel objet (1)

- Hibernate est capable d'automatiser les MAJ à partir de manipulation sur les objets métiers :

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = (Message) session.load(Message.class, new Long(1));
message.setText("Cher Monsieur");
Message nextMessage = new Message("Considérer ma demande
    d'augmentation");
message.setNextMessage( nextMessage);
tx.commit();
session.close();
```


Association entre objet persistant et un nouvel objet (2)

- Génération des requêtes SQL :

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGE m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT,
    NEXT_MESSAGE_ID) values (2, 'Considérer ma demande
    d'augmentation ', null);

update MESSAGES
set MESSAGE_TEXT = 'Cher Monsieur', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```


Interfaces centrales

- Session
  - interface la plus importante qui définit une session de travail sur les objets persistants
  - créée et détruite sans cesse : coût faible
  - associée à un seul thread à la fois
- SessionFactory
  - une session est obtenue de cette interface
  - une seule interface pour toute l'application, créée à son initialisation
  - Gestion des caches
- Configuration
  - Objet assurant la configuration et le bootstrap Hibernate
  - premier objet créé
- Transaction
  - assure une gestion des transactions indépendantes de l'environnement
  - interface optionnelle
- Requête et critère
  - exécution et optimisation des requêtes
  - gestion des paramètres

Interfaces secondaires

- Callback (option)
  - Observateur des événements sur les objets
  - permet la gestion d'audit
- Types
  - Permet l'association d'un type Hibernate à un type de BD
  - Possibilité de créer des types utilisateurs
- Extension
  - La majorité des fonctionnalités Hibernate est hautement configurable : choix de stratégie
    - PK génération
    - support SQL
    - Cache
    - JDBC, etc.

Configuration de base

- Deux modes
  - Environnement géré: J2EE au dessus de Java.
  - Environnement non géré: Servlet, tomcat, ligne de commande.
- Cours centré sur environnement non géré:
  - Hibernate joue le rôle de client d'un pool de connection JDBC
    - acquérir une nouvelle connexion est couteux
    - Maintenir beaucoup de connexions est couteux
    - Créer des statements est souvent couteux (=>drivers)
  - Définition d'un fichier de propriétés ou d'un fichier de configuration XML

Properties file

- Gestion du POOL JDBC

```
hibernate.connection.driver_class = org.postgresql.Driver
// JDBC driver (dans le CLASSPATH de l'application)
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
// chemin de connection à la BD
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.POstgresSQLDialect
// Dialect SQL pour la BD
hibernate.c3P0.min_size = 5
// minimum de connections à conserver
hibernate.c3P0.max_size = 20
//  nombre limite de connections
hibernate.c3P0.timeout = 300
// avant suppression d'une connection
hibernate.c3P0.max_statements = 50
// maximum de statements cachés
hibernate.c3P0.idle_test_period = 300
```


Démarrage Hibernate

- Méthode
  - Placer hibernate2/3.jar dans le CLASSPATH de l'application
  - Ajouter les dépendances Hibernate dans le CLASSPATH
  - Choisir un pool de connection JDBC supporté par Hibernate => properties file
  - créer une instance de Configuration dans l'application et ajouter le mapping
- Exemple :

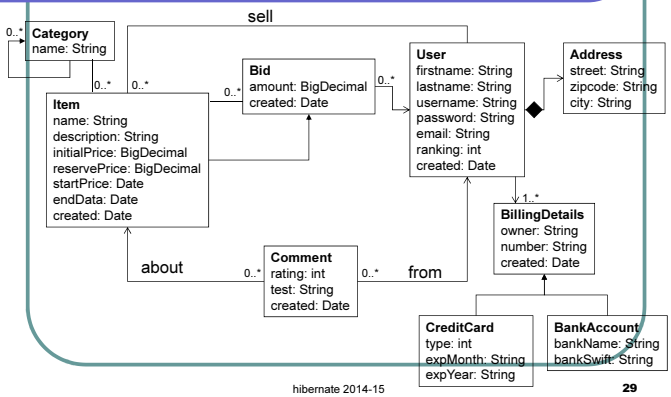
```
Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties(System.getProperties());
SessionFactory sessions = cfg.buildSessionFactory();
ou
SessionFactory sessions = new Configuration()
    .addResource("hello/Message.hbm.xml")
    .setProperties(System.getProperties())
    .buildSessionFactory();
```


Fichier de configuration xml

- Hibernate peut utiliser un fichier qui centralise tous les paramètres (par défaut *hibernate.cfg.xml*)

```
<hibernate-configuration>
    <session-factory name="java:/hibernate/HibernateFactory">
            <property name="show_sql">true</property>
            <property name="connection.datasource>
                    java:/comp/env/jdbc/AuctionDB
            </property>
            <property name="dialect">
                    net.sf.hibernate.dialect.PostgresSQLDialect
            </property>
            <property name="transaction.manager_lookup_class">
                    net.sf.hibernate.transaction.JBossTransactionManagerLookup
            </property>
            <mapping resource="auction/Item.hbm.xml"/>
            <mapping resource="auction/Category.hbm.xml"/>
            <mapping resource="auction/Bid.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Exemple fil rouge (site d'enchères)



Mapping des classes persistantes

Hibernate = POJO model

- Hibernate repose sur un modèle de programmation POJO (Plain Old Java Objects) issu des Java beans
  - préférer les classes sérialisable
  - utiliser des méthodes d'introspections
  - définir les méthodes métiers

```
public class User implements Serializable {
    private String username;
    private Address address;
    public User(){}
    public String getUsername() { return username;}
    public void setUsername(String username) { this.username = username;}
    public Address getAddress()…
    public void setAddress(Address address)…
    public MonetaryAmount calcShippingCosts(Address fromLocation)…}
```


Les Associations en POJO

- L'objectif est de faire apparaître les associations bidirectionnelles:

public class Category implements Serializable{	Public class Item {
private String name;	private String name;
private Category parentCategory;	private String description;
private Set childCategories = new HashSet();	private Set categories = new HashSet();
private Set items = new HashSet();	…
public String getName()… }	public void addCategory(Set categories)
	{ if (category==null) throw …
	category.getItems().add(this);
	categories.add(category); }…

Avec l'utilisation qui va avec:

```
Category aParent = new Category();
Category aChild = new Category();
aChild.setParentCategory(aParent);
aParent.getChildCategory().add(aChild);
```


Ajouter la logique aux classes

- Il faut prévoir d'adapter la logique des méthodes get/set au contenu de la BD:

```
public void setName(String name) {
    StringTokenizer t = new StringTokenizer(name);
    firstname = t.nextToken();
    lastname = t.nextToken(); }
```

si la BD stocke le nom dans 1 ou 2 attribut !

Mapping de la classe Category

```
<hibernate-mapping>
    <classe name="org.hibernate.auction.model.Category"
    table="CATEGORY">
        <id name="id" column="CATEGORY_ID" type="long">
            <generator class="native"/>
        </id>
        <property name="name" column="NAME" type="string"/>
    </class>
</hibernate-mapping>
```

- Mapping de la classe sur une relation
- Mapping des propriétés sur des attributs
- Mapping de l'identifiant sur la les attributs de la relation

Mapping des propriétés

- Hibernate utilise la réflexion pour déterminer le type d'une propriété:

```
<property name="name" column="NAME" type="string"/>
```

équivalent à

```
<property name="name" column="NAME"/>
```

- Hibernate autorise les propriétés dérivés (ou calculés):

```
<property name="totalIncludingTax"
    formula="TOTAL + TAX_RATE * TOTAL"
    type="big_decimal/>
```

ou encore

```
<property name="averageBidAmount"
    formula="(select AVG(b.AMOUNT) from BID b where b.ITEM_ID=ITEM_ID)"
    type="big_decimal"/>
```


Mapping étendu

- Contrôle des MAJ: le dictionnaire peut autoriser ou non les MAJ

```
<property name="name" column="NAME" type ="string" insert="false" update="true"/>
```

- Déclaration des classes: le Package Java peut être déclarée de 2 manières

```
<hibernate-mapping>
    <class name="org.hibernate.auction.model.Category" table="CATEGORY">…
```

peut s'écrire

```
<hibernate-mapping
    package="org.hibernate.auction.model"
    <class name="Category" table="CATEGORY">…
```

- Les informations de mapping peuvent être définies pendant l'exécution !

Gestion des identités

- 3 méthodes pour identifier des objets (comme en BDOO):
  - Identité objet : même emplacement mémoire
  - Egalité d'objet : même valeurs entre objets
  - Identité de BD : même PK
- Implémentation d'une propriété identifiant

```
public class Category {
    private Long id;
    public Long getId() { return this.id;}…}
avec son mapping
<class name="Category" table="CATEGORY">
    <id name="id" column="CATEGORY_ID" type="long">
            <generator class="native"/>
    </id>
```


Choix du type de PK pour le mapping d'identifiant

- native: utilise le générateur d'identifiant de la BD (identity, sequence, hilo)
- identity: utilise des attributs d'une relation
- sequence: disponible dans Oracle, DB2, Postgres, etc.
- increment: incrémentation de la valeur maximum d'un attribut
- hilo: générateur d'identifiant unique (sur une BD)
- uuid.hex: générateur d'une chaine unique (sur un réseau)

Granularité d'un objet

- Hibernate dispose de 2 grains:
  - l'entité (entity) qui persiste et qui dispose de son propre identifiant BD. Une entité possède un cycle de vie.
  - une valeur (value type) qui ne dispose pas d'identifiant BD et dont le cycle de vie dépend d'une entité.
- Problème entre les classes User et Address : relation *part of*
  - Correct en UML mais Address n'est pas une entité en Hibernate
  - on utilise la notion de composant (component) d'une entité

```
<class name="User" table="USER">
    <id name="id" column="USER_ID" type="long"><generator class="native"/></id>
    <property name="username" column="USERNAME" type="string"/>
    <component name="homeAddress" class="Address">
            <property name="street" type="string" column="HOME_STREET" notnull="true"/>
    …</component>
    <component name="billingAddress" class="Address">
            <property name="street" type="string" column="BILLING_STREET" notnull="true"/>
    …</component>
</class>
```


Mapping pour l'héritage

- 3 stratégies selon les besoins
  - une table par classe concrète: héritage et polymorphisme disparaissent, une classe définie par classe concrète (solution à éviter)
  - une table pour la hiérarchie: mapping avec une seule table dont un attribut pour discriminer les propriétés.
  - une table par sous-classe: chaque table ne contient que les informations spécifiques, le mapping permet d'expliciter le concept de sous-classe.

Une table = la hiérarchie

- Solution la plus performante mais
  - les attributs des sous-classes doivent être Nullable… Pb de cohérence des données
  - nécessité d'un attribut discriminant
- Exemple : Billingdetails avec ses 2 sous classes CreditCard & BankAccount

```
<hibernate-mapping>
    <class name="BillingDetails" table="BILLING_DETAILS" discriminator-value="BD">
            <id name="id" column="BILLING_DETAILS_ID" type="long">
                    <generator class="native"/> </id>
            <discriminator column="BILLING_DETAILS_TYPE" type="string"/>
            <property name="name" column="OWNER" type="string"/> …
            <subclass name="CreditCard" discriminator-value="CC">
                    <property name="type" column="CREDIT_CARD_TYPE"/>
                    …
            </subclass>
    … </class>
</hibernate-mapping>
```


Une table par sous-classe

- Chaque table représentant une sous-classe peut être liée à la table représentant la super classe via PK et FK:

```
<hibernate-maping>
<class name="BillingDetails" table="BILLING_DETAILS">
        <id name="id" column="BILLING_DETAILS_ID" type="long">
                <generator class="native"/> </id>
        <property name="name" column="OWNER" type="string"/> …
        <joined-subclass name="CreditCard" table="CREDIT_CARD">
                <key column="CREDIT_CARD_ID">
                <property name="type" column="TYPE"/>
                …
        </joined-subclass>
    … </class>
</hibernate-maping>
```


Gestion des associations

- Hibernate n'implémente pas de gestionnaire d'associations:
  - les associations sont "naturellement" unidirectionnelles
  - le mapping propose des liens one-to-one, one-to-many, many-to-one et many-to-many
- Reprenons l'exemple d'un objet avec plusieurs enchères:
  - coté classe enchère

```
public class Bid { private Item item; …}
```

  - et son mapping

```
<class name="Bid" table="BID">
    …
    <many-to-one name="item" column="ITEM_ID" class="Item" not-null="true"/>
</class>
```


Transformer une association unidirectionnelle en bidirectionnelle

- L'objectif est de pouvoir naviguer dans le graphe d'objets:
  - coté classe Item

```
public class Item{ private Set bids = new hashSet();
    …
    public void addBid(Bid bid) {
            bids.setItem(this);
            bids.add(bid); }
}
```

  - et son mapping

```
<class name="Item" table="ITEM">
    …
    <set name="bids">
            <key column="ITEM_ID"/>
            <one-to-many class="Bid"/>
    </set>
</class>
```

- 2 problèmes se posent:
  - double détection des MAJ dans addBid car rien n'explicite l'association bidirectionnelle (2x unidirectionnelles)
  - pas de persistance automatique des enchères: il faut appeler la méthode save() sur l'interface Session

Transformer une association unidirectionnelle en bidirectionnelle

- Le mapping final explicite l'association bidirectionnelle avec persistance auto:

```
<class name="Item" table="ITEM">
    …
    <set name="bids" inverse="true" cascade="save-update">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

- Relation parent/enfants = non indépendance des cycles de vie: "save-update" n'est pas suffisant

```
<class name="Item" table="ITEM">
    …
    <set name="bids" inverse="true" cascade="all-delete-orphan">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```


Système de typage Hibernate
"Built-in mapping types"

mapping type	Java Type	SQL type
integer	Int	INTEGER
long	Long	BIGINT
short	Short	SMALLINT
float	Dloat	FLOAT
double	Double	DOUBLE
big_decimal	Java.math.BlgDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte	TINYINT
boolean	boolean	BIT
yes_no	boolean	CHAR(1)('Y' or 'N')
true_false	boolean	CHAR(T)('Y' or 'F')
date	java.util.Date	DATE
time	java.util.Date	TIME
timestamp	java.util.Date	TIMESTAMP
calendar	java.util.calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
binary	byte[]	VARBINARY (ou BLOB)
text	java.lang.String	CLOB
serializable	java.io.Serializable class	VARBINARY (ou BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

Mapping des collections de valeurs

- Nous savons expliciter les associations d'entités… mais pour les valeurs ? => Exemple d'une collection d'images associées aux objets.
- Hibernate propose 4 types de collection: set, bag, list, map

- Collection Set
  - coté Item

```
public class Item{
    private Set images = new hashSet();
    …}
```

  - coté mapping

```
<set name="images" lazy="true" table="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```


Mapping des collections de valeurs

- Collection Bag : Hibernate ne propose pas directement de liste non ordonnée
  - coté Item: on remplace Set par List (l'ordre n'est pas préservée lors de la persistance avec une sémantique de bag)
  - coté mapping: on reprend la même structure avec une clé supplémentaire (doublon)

```
<idbag name="images" lazy="true" table="ITEM_IMAGE">
    <collection-id type="long" column="ITEM_ID"><generator class="sequence"/>
    <collection-id/>
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

- Collection List : il faut un attribut servant d'index pour repérer la position d'un élément dans la liste (ordre)

```
<list name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <index column="POSITION"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```


Collections de composants

- Une classe "component" est considérée comme une valeur.
- Exemple d'une classe Image composant de Item, avec les propriétés name, filename, sizeX et sizeY

```
<set name="images" lazy="true" table="ITEM_IMAGE" order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <composite-element class="Image">
        <parent name="item"/>
        <property name="name" column="IMAGE_NAME" not-null="true"/>
        <property ….>
    </composite-element>
</set>
```

(ici le mapping est une association bidirectionnelle)

Association one-to-one

- Comment construire le mapping d'une association 1:1 entre deux entités (ici une classe Adresse et une classe User)
  - définition de la classe

```
<class name="Adress" table="ADRESS">
    <id name="id" column="ADDRESS_ID"><generator class="native"/></id>
    <property name="street"/>…
</class>
```

  - on ajoute un attribut BILLING_ADDRESS_ID dans la table USER

```
<many-to-one name="billingAddress" class="Address" column="BILLING_ADDRESSE_ID" cascade="all" unique="true"/>
```

  - on ajoute une propriété user dans la classe Adresse pour assurer la bidirection

```
<one-to-one name="user" class="User" property-ref="billingAddress"/>
```

  - Exemple d'utilisation:

```
Address address=new Adress();
address.setStreet("681 rue de la Passerelle");
address.setCity("Grenoble"); address.setZipcode("38000");
Transaction tx=session.beginTransaction();
User user=(User) session.get(user.class, userId);
address.setuser(user);
user.setBillingAddress(adress);
tx.commit();
```


Association many-to-many

- Considérons qu'une catégorie est associée à un ensemble d'objet et les objets sont associés à un ensemble de catégorie.
  - On veut pouvoir faire :

```
Transaction tx=session.begintransaction();
Category cat=(Category) session.get(Category.class, categoryId);
Item item=(Item) session.get(Item.class, itemId);
cat.getItems().add(item);
item.getCategories().add(category);
tx.commit();
```

  - coté Category

```
<class name="Category" table="CATEGORY">…
    <set name="items" table="CATEGORY_ITEM" lazy="true" cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
</class>
```

  - coté Item

```
<class name="Item" table="ITEM"> …
    <set name="categories" table="CATEGORY_ITEM" lazy="true" inverse="true" cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```


Mapping & annotations

- Des annotations remplacent les fichiers de mapping
  - Code + mapping au même endroit
  - Plus lisible ou plus brouillon selon les gouts
- Exemple basique

```
@Entity
@Table(name="Category")
public class Category {
    private Long id;
    @Id
    @Column(name="CATEGORY_ID")
    public Long getId() { return this.id;}…}
  … }
```

- Exemple de gestion des associations

```
@Entity
@Table (name="ITEM")
public class Item{ private Set bids = new hashSet();
    …
    @OneToMany(mappedBy="ITEM_ID", inverse="true",cascade="all-delete-orphan")
    @OrderBy("created")
    public List<bid> getBids { return bids; }    …}
```

Gestion du Polymorphisme

- Supposons que nous ayons le mapping suivant pour BillingDetails:

```
<many-to-one name="user" class="User" column="USER_ID"/>
```

- Et celui-ci pour Users:

```
<set name="billingDetails" lazy="true" cascade="save-update" inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

- Le polymorphisme est immédiat sous Hibernate (pour des collections comme pour des associations simple) si des classes sont déclarées avec <subclass> ou <joined-subclass>:

```
CreditCard cc=new CreditCard();
cc.setNumber(ccNumber); cc.setType(ccType); cc.setexpirydate(ccExpiryDate);
Session session=getSessionFactory.openSession();
Transaction tx=session.beginTransaction();
user user=(User) session.get(User.class,uid);
user.addBillingDetails(cc);
tx.commit();
session.close();
```


Gestion du polymorphisme

- Itération sur l'ensemble des paiements d'un utilisateur:

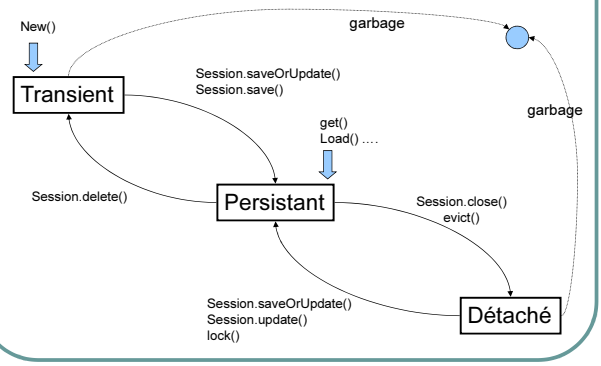
```
Session session=getSessionFactory.openSession();
Transaction tx=session.beginTransaction();
user user=(User) session.get(User.class,uid);
Iterator iter=user.getBillingDetails().iterator();
while (iter.hasnext()){
    BillingDetails bd=(BillingDetails) iter.next();
    bd.pay(ccPaymentAmount);
tx.commit();
session.close();
```


Travailler avec des classes persistantes

Cycle de vie des objets

- Objets transients
  - tous les objets Hibernate ne sont pas persistants
  - un objet transient n'est pas attaché à une BD et ne participe pas aux transactions
  - un objet transient peut devenir persistants
- Objets persistants
  - un objet persistant possède un identifiant lié à une clé primaire de la BD
  - les objets extraits à l'aide d'une requête hibernate sont persistants
  - Hibernate détecte les objets persistants à mettre à jour : transparent pour le développeur.
- Objets détachés
  - un objet persistant devient détaché après fermeture d'une session
  - un objet détaché peut redevenir persistant

Cycle de vie des objets



Portée des objets persistants

- Deux objets utilisant la même clé sont identiques (au niveau référence mémoire)

```
Session session1=sessions.OpenSession();
Transaction tx1=session1.beginTransaction();
Object a=session1.load(Category.class,new Long(1234));
Object b=session1.load(Category.class,new Long(1234));
if (a==b) {System.out.println("a et b identiques.");}
tx1.commit(); session1.close();
```

- Hibernate ne garantit pas l'identité hors d'une session

```
Session session2=sessions.OpenSession();
Transaction tx2=session2.beginTransaction();
Object b2=session2.load(Category.class,new Long(1234));
if (a!=b2) {System.out.println("a et b différents.");}
System.out.println( a.getId().equals(b2.getId()) );
tx2.commit(); session2.close();
```


Implémenter equals() et hashCode()

- Hibernate utilise ces 2 méthodes pour identifier les duplications d'éléments
  - java.long.Object par défaut marche tant que les objets sont de mêmes natures.
  - sinon il faut les redéfinir
- 2 méthodes de redéfinitions
  - par comparaison de valeurs

```
public class User {…
    public boolean equals(Object other) {
            if (this==other) return true;
            if (!(other instanceof User)) return false;
            final User that=(User) other;
            if (!this.getUsername().equals(that.getUsername()) return false;
            if (!this.getPassword().equals(that.getPassword()) return false;
            return true; }
    public int hashCode() { int result=14;
            result=29*result+getUsername.hashCode();
            result=29*result+getPassword.hashCode();
            return true; }
```

  - basé sur l'égalité métier des clés

Utilisation du gestionnaire de persistance

- Faire persister un objet

```
User user=new User();
user.getName().setFirstName("John");
user.getName().setLastName("Doe");
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
session.save(user);
tx.commit(); session.close();
```

  - Il est possible de modifier user après save(): l'état de user sera modifié au commit() car il est devenu persistant.
- Mettre à jour un objet détaché

```
user.setPassword("secret");
Session session2=sessions.OpenSession();
Transaction tx=session2.beginTransaction();
session2.lock(user,LockMode.NONE);
session2.update(user);
user.setUsername("johnny");
tx.commit(); session2.close();
```

Utilisation du gestionnaire de persistance

- Récupérer un objet persistant

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
User user=(User) session.get(User.class, new Long(1234));
tx.commit(); session.close();
```

- Mettre à jour un objet persistant

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
User user=(User) session.get(User.class, new Long(1234));
usert.setPassword("secret");
tx.commit(); session.close();
```


Utilisation du gestionnaire de persistance

- Faire d'un objet persistant, un objet transient

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
User user=(User) session.get(User.class, new Long(1234));
session.delete(user);
tx.commit(); session.close();
```

- Faire d'un objet détaché, un objet transient

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
session.delete(user);
tx.commit(); session.close();
```


Persistance transitive

- Un objet est persistant transitivement si il est accessible via le graphe d'objet par un objet persistant
  - par défaut Hibernate n'assure pas cette persistance: il faut l'expliciter dans le mapping, **cascade=**
    - "none" : ignore les associations
    - "save-update" : considère les associations lors d'un commit ou d'un save/update pour sauver de nouveaux transients (ou pour faire persister des détachés)
    - "delete" : considère les associations lors de la suppression d'un objet persistant
    - "all" : idem "save-update"+"delete"
    - "all-delete-orphan" : idem "all" + suppression des persistants ne faisant plus partie de l'association
    - "delete-orphan" : suppression des persistants hors association

Exemple de persistance transitive

- Gestion des enchères

```
<class name="Category" table="CATEGORY">
    …
    <property name="name" column="CATEGORY_NAME"/>
    <many-to-one name="parentCategory" class="Category"
            column="PARENT_CATEGORY_ID" cascade="none"/>
    <set name="childCategories" table="CATEGORY" cascade="save-update"
            inverse="true">
            <key column="PARENT_CATEGORY_ID"/>
            <one-to-many class="Category"/>
    </set>
</class>
```


Exemple de persistance transitive

- Persistance automatique d'une sous-catégorie:

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
Category computer=(Category) session.get(Category.class,computerID);
Category laptops=new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
tx.commit(); session.close();
```

- Persistance automatique d'une sous-catégorie définie hors session:

```
Category computer= … détaché d'une session précédente
Category laptops=new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
session.save(laptops);
tx.commit(); session.close();
```


Comment récupérer des objets

- par l'identifiant

```
User user = (User) session.get(User.class, userId); (peut renvoyer NULL)
ou
User user = (User) session.load(User.class, userId); (ne renvoie jamais NULL => exception)
```

- par requête HQL (uniquement interrogation)

```
Query q = session.createQuery("from User u where u.firstname= :fname");
q.setString("fname","Max");
List result = q.llist();
```

- par critères

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Expression.like("firstname","Max");
List result = criteria.list();
```


Comment récupérer des objets

- par l'example (QBE)

```
User exampleUser = new User();
exampleUser.setFirstname("Max");
Criteria criteria = session.createCriteria(User.class);
criteria.add( Example.create(exampleUser));
List result = criteria.list();
```

- Fetching
  - Le fait de compléter un objet avec des informations complémentaire (un utilisateur avec ses adresses, un objet avec ses enchères, etc.) => jointure externe en SQL
  - Hibernate gère plusieurs stratégies de Fetching
    - Immediate : récupère tout le sous-graphe d'objets
    - **Lazy : récupère les éléments du graphe en fonction des accès**
    - Eager : les objets associés à charger sont explicités
  - Hibernate utilise un proxy/cache pour réaliser ces stratégies.

Interrogation avancée

Exécution d'une requête

- L'interface standard est HQL via createQuery()

```
Query hqlQuery = session.createQuery("from User");
```

- équivalente à la requête SQL:

```
Query sqlQuery = session.createSQLQuery (
    "select {u.*} from USERS {u}", "u", User.class) ;
```

- qui peut s'écrire sous forme de critère

```
Criteria crit = session.createCriteria(User.class);
```


Exploitation des résultats

- Pagination des résultats:

```
Query query = session.createQuery("from User u order by u.name asc");
query.setFirstResult(0);
query.setMaxResults(10);
```

- Récupération d'une liste

```
List result = query.list();
```

- Récupération d'un élément

```
Bid maxBid = (Bid) session.createQuery("from Bid b order by b.amount desc")
    .setMaxResults(1).uniqueResult();

Bid bid = (Bid) session.createCriteria(Bid.class)
    .add(Expression.eq("id",id).uniqueResult();
```


Paramétrage des requêtes

- On évitera de construire une règle sous la forme d'une String prêt à l'emploi : on utilise l'instanciation de paramètres
  - en nommant les paramètres

```
String query="from Item item where item.description like :searchString;
List result = session.createQuery(query)
    .setString("searchString",searchString).list();
ou  .setParameter("searchString",searchString,Hibernate.STRING).list();
```

  - en positionnant les paramètres

```
String query = "from Item item where item.description like ? and item.date > ?";
List result = session.createQuery(query).setString(0,searchString).setDate(1,minDate)
    .list();
```


Requête de base

from Bid ⇔ Session.createCriteria(Bid.class)
est traduit en
select B.BID_ID, B.AMOUNT, B.ITEM_ID, B.CREATED from BID B

- Utilisation d'Alias

from Bid as bid (ou Bid bid)

- requête Polymorphique

from BillingDetails
et pour avoir les objets concrets des sous-classes :
from CreditCard

- Filtrage : via la clause WHERE
  - opérateur de comparaison classique
  - opérateur arithmétique
  - String matching LIKE
  - opérateur logique
  - opérateur IS NULL

```
from User u where u.email is not null
⇔
session.createCriteria(User.class).add(Expression.isNotNull("email")).list();
```


Jointure

- 4 manières d'exprimer des jointures
  - jointure ordinaire dans la clause from
  - Fetch jointure
  - Theta-style
  - implicite
- Jointure en mode Fetch (jointure externe optimisée)
  - mode HQL

```
from Item item
left join fetch item.bids where item.description like "%gc%"
```

  - mode critère

```
session.createCrieria(Item.class).setFetchMode("bids",FetchMode.EAGER)
    .add(Expression.like("description","gc",MatchMode.ANYWHERE)).list();
```

  - traduction SQL

```
select I.DESCRIPTION, I.CREATED, I.SUCCESSFUL_BID, B.BID_ID,
    B.AMOUNT,B.ITEM_ID,B.CREATED
from ITEM I left outer join BID B on I.ITEM_ID=B.ITEM_ID
where I.DESCRIPTION like '%gc%'
```


Jointure ordinaire

- Une jointure classique utilise l'opérateur join et les Alias:

```
from Item item join item.bids bid
where item.description like '%gc%' and bid.amount>100
```

- A la différence du Fetch mode, l'association n'est pas reformatée : le résultat est un tableau de paires (Item, Bid)

```
Query q= session.createQuery("from Item item join item.bids bid");
Iterator pairs=q.list().iterator();
while (pairs.hasNext() ) {
    Object[] pair=(Object[]) pairs.next();
    Item item=(Item)pair[0]; Bid bid=(Bid)pair[1];
}
```


Jointure implicite

- Permet une interrogation à la manière OQL (BDOO)
  - pour interroger des objets composants
  - pour naviguer dans des associations implicites

```
from User u  where u.address.city = 'Grenoble'
⇔
session.creatCriteria(User.class)
    .add(Expression.eq("address.city","Grenoble"));

from Bid bid where bid.item.category.name like 'Laptop%'
```


Jointure Theta-Style

- Utile pour réaliser une jointure avec un critère ne faisant pas intervenir des attributs liés par une association

```
from User user, LogRecord log
where user.username = log.username

qui s'utilise via un Iterator
Iterator i= session.createQuery("from User user, LogRecord log
where user.username = log.username").list().iterator();
while (i.hasNext() ) {
    Object[] pair=(Object[]) i.next();
    User user=(User)pair[0]; LogRecord log=(LogRecord)pair[1];
}
```

Agrégation

- Hibernate permet d'écrire via HQL des projections, des Group By Having, des fonctions d'agrégations, la suppression de doublon (distinct)

```
Select p.LASTNAME, count(A)
        From Person p join Addresse a
        group by p.LASTNAME
        having count(a)>10
```


Notion de sous-requêtes

- Hibernate est l'un des rares ORM a proposé des sous requêtes dans les clauses FROM et WHERE

```
from User u where 10<(
    select count(i) from u.items i where i.successfulBid is not null )

from Bid bid where bid.amount +1 >= (
    select Max(b.amount) from Bid b )
```

- Opérateurs ANY, ALL, SOME, IN disponibles.

Gestion des transactions

Définition d'une transaction

- Une transaction est définie entre les appels beginTransaction() et commit()

```
Session session=sessions.OpenSession();
Transaction tx=null;
try {
      tx=session.beginTransaction();
      concludeAuction();
      tx.commit();
} catch (Exception e) {
      if (tx!=null) {
            try { tx.rollback(); }
            { catch (HibernateException he {…}
      } throw e;
} finally {
      try { session.close(); }
      catch (HibernateException he) { throw he;}
}
```


Niveau d'isolation

- Par défaut celui de JDBC (soit read committed soit repeatable read)
- Option de configuration (donc pour tout le pool de connections):
  Hibernate.connection.isolation = x
- Où x peut être :
  - 1 - Read uncommitted
  - 2 - Read commited
  - 3 - Repeatable read
  - 8 - Serializable