CHAPITRE 1: BASES DE DONNÉES RELATIONNELLES

1. Les Systèmes de Gestion de Bases de Données (SGBD)
2. Introduction à la conception de bases de données relationnelles
3. Décomposition de schémas relationnels
4. Normalisation des relations
5. Dépendances fonctionnelles et conception de schémas

1.1 Les Systèmes de Gestion de Bases de Données (SGBD)

Un système de base de données peut être décrit par l'équation suivante:

Système de Base de Données = Base de Données + SGBD

Un **SGBD** est un système permettant de gérer et de manipuler la base de données, et une **base de données** est une collection de données en **relation** qui sont:

1. *partagées* par de multiples applications (utilisateurs et/ou programmes),
2. stockées avec une *redondance minimum*,
3. *indépendantes* des applications,
4. organisées afin d'être une *fondation pour de futures applications*.

Le **modèle de données** est le formalisme qui décrit la structure logique de la base de données et les opérations sur celle-ci.

Le **schéma de base de données** est la structure de la base de données. **L'instance de la base de données** est le contenu actuel de la base de données.

Une base de données est donc constituée d'une part d'un *schéma de base de données* et d'une instance de la *base de données* d'autre part. On peut, par analogie avec les langages de programmation, définir les correspondances suivantes:

modèle de données	$\leftrightarrow$	langage de programmation
schéma	$\leftrightarrow$	déclaration d'une structure de données
instance	$\leftrightarrow$	variable

1.1.1 Modèle de Données Relationnel

Une instance de la base de données peut être considérée comme une collection de relations mathématiques. Chaque relation est représentée par une table dont chaque colonne est appelée un attribut et chaque ligne est appelée un tuple ou n-uplet. On associe à chaque attribut un ensemble de valeur, appelé domaine.

Un schéma de relations est alors défini comme un ensemble d'attributs.

EXEMPLE

EMPLOYÉ = {nom, nosécu, nomdépartement, salaire, datenaissance}

CONVENTION DE NOTATION

EMPLOYÉ (nom, nosécu, nomdépartement, salaire, datenaissance)

Un schéma de base de données est une collection de schémas de relations.

EXEMPLE

{ EMPLOYÉ, DÉPARTEMENT, PROJET }

NOTATIONS

schéma de relations R	R(A1,A2,...,An)
n-uplet	<v1,v2,...,vn>
valeur de l'attribut Ai	t[Ai]
valeurs des attributs	t[Au,Aw,...Az]
attribut A du schéma R	R.A

1.1.2 Contraintes d'intégrité

Il existe un certain nombre de propriétés qui doivent être respectées pour chaque instance d'un schéma de relations ou d'un schéma de base de données afin de préserver la cohérence des informations stockées dans la base. Ces propriétés sont appelées contraintes d'intégrité.

Les dépendances entre données sont des contraintes d'intégrité qui spécifient les relations entre les valeurs des attributs. Elles sont très utiles dans la conception de bases de données.

1.2 Introduction à la conception de bases de données relationnelles

1.2.1 Deux approches pour concevoir un schéma de base de données

Il existe deux approches pour concevoir un schéma de base de données: une approche descendante, également appelée méthode par décomposition, et une approche ascendante, également appelée méthode synthétique. Nous présentons rapidement pourquoi la conception d'un schéma de base de données peut être présenter des anomalies. Les dépendances fonctionnelles sont un outil théorique qui permet de vérifier la qualité de conception d'un schéma et de comprendre les conséquences de la redondance de données.

1. **Conception descendante (top-down design)**

   On part d'un schéma de relations unique…

   ↓ décomposition basée sur les dépendance entre données (par exemple, les dépendances fonctionnelles)

   …pour obtenir une collection de schémas de relations.

2. **Conception ascendante (bottom-up design)**

   On part d'une collection d'attributs…

   ↓ synthèse (regroupement)

   …pour obtenir une collection de schémas de relations.

QUESTIONS

- Quelles sont les propriétés intéressantes d'un schéma de base de données relationnelles ?
  - mesure de la qualité du schéma,
  - méthodologie de conception.
- Quels sont les algorithmes permettant d'obtenir un schéma possédant les bonnes propriétés ?
  - outil dédié à la conception,
  - conception automatique.

MESURE DE LA QUALITÉ

1. **sémantique des attributs**

   Plus il est facile d'expliquer la sémantique des attributs d'une relation, plus la conception du schéma de relations est bonne.

   PRINCIPE 1: concevoir un schéma de relations de telle sorte qu'il soit simple d'expliquer ce qu'il représente.

   EXEMPLE

   Que signifie *FOURNIT* (f#, nom, adresse, p#, produit, prix, couleur, nomproj, dép) ?

   Ce schéma de relations doit être décomposé comme suit:

   PROD(p#, produit, couleur)
   FOURNITPROD(f#, nom, adresse,p#, produit, prix)
   PROJET(nomproj, dép)
   UTILISE(nomproj, p#)

   Chaque schéma de relations et chaque attribut ont une sémantique précise.

2. **les informations redondantes**
   Les informations redondantes, à l'origine de problèmes, doivent être supprimées dans la mesure du possible.

   PRINCIPE 2: concevoir un schéma de relations de telle manière que les insertions, les suppressions et les mises à jour ne posent pas de problèmes.

   EXEMPLE
   Si on utilise la relation suivante *FOURNITPROD*(f#, nom, adresse, p#, produit, prix) plutôt que le schéma de relations suivant:
   FOURNISSEUR(f#, nom, adresse)
   FOURNIT(f#, p#, prix)
   alors le nom et l'adresse d'un fournisseur sont stockés plusieurs fois dans les n-uplets correspondant aux produits vendus par le fournisseur.

QUAND Y-A-T-IL REDONDANCE D'INFORMATION ?

On constate assez facilement que la même adresse apparaît autant de fois qu'il y aura de produits si l'on utilise la relation *FOURNITPROD*(f#, nom, adresse, p#, produit, prix). Or on sait **qu'un fournisseur ne possède qu'une seule adresse** et cela même s'il vend plusieurs produits. On appelle cette propriété une dépendance fonctionnelle. Un mauvais schéma relationnel peut alors entraîner des anomalies lors des manipulations.

f#	nom	adresse	p#	produit	prix
F1	Labaleine	Paris	P1	parapluie	110
F2	Lemelon	Lyon	P2	chapeau	50
F3	Toutcuir	Lyon	P3	sac à main	650
F1	Labaleine	Paris	P4	parasol	150
F1	Labaleine	Paris	P5	ombrelle	70
F4	Letour	Nantes	P6	ceinture	55
F5	Legrand	Paris	P3	sac à main	650

Dans la relation ci-dessus, il existe des redondances qui peuvent engendrer les difficultés suivantes:

- **anomalie d'insertion**. Telle qu'elle se présente, cette relation ne permet pas de mémoriser dans la base un produit dont le fournisseur n'existe pas.
- **anomalie de suppression**. La disparition d'un fournisseur, qui est l'unique fournisseur d'un produit, entraîne également la disparition des informations concernant ce produit (exemple: Letour).
- **anomalie de mise à jour**. Toute modification sur le prix d'un produit qui apparaît plusieurs fois dans la relation doit être répercutée sur tous les n-uplets correspondants (exemple: sac à main). Cela augmente le temps de mise à jour et les risques d'incohérence.

On résout ces problèmes en étudiant les dépendances entre les données, et en décomposant et normalisant les relations. On obtient ainsi des schémas de relations qui évitent les anomalies citées ci-dessus. Attention cependant à ne pas voir la normalisation comme un dogme. Il arrive parfois que, pour des raisons d'efficacité notamment, on soit obligé de *dénormaliser*.

1.2.2 Dépendance Fonctionnelle (DF)

Soient R un schéma de relations, et $X=A_1,A_2,\ldots,A_n$ et $Y=B_1,B_2,\ldots,B_m$ des sous-ensembles de R.

X *détermine fonctionnellement* Y, noté $X \rightarrow Y$, ssi pour chaque instance r de la relation R, pour chaque paire de n-uplets t et s appartenant à r, $t[X] = s[X]$ implique $t[Y] = s[Y]$ (où $t[X]$ désigne la projection du n-uplet t sur les attributs X).

Autrement dit, $X \rightarrow Y$ signifie *qu'une valeur de X détermine au plus une valeur de Y*.

Une dépendance fonctionnelle est une propriété définie sur l'intention du schéma et non son extension (elle est donc invariante dans le temps et ne peut être extraite à partir d'exemples). C'est une propriété qui doit être extraite de la connaissance que l'on a de l'application à modéliser et qui permet d'éliminer la redondance.

EXEMPLES

Considérons la relation suivante

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

avec une instance de la relation contenant les n-uplets suivants:

titre	**année**	**durée**	**type**	**studio**	**acteurPrincipal**
StarWars	1977	124	couleur	Fox	Carrie Fisher
StarWars	1977	124	couleur	Fox	Mark Hamill
StarWars	1977	124	couleur	Fox	Harrison Ford
Mighty Ducks	1991	104	couleur	Disney	Emilio Estevez
Wayne's World	1991	95	couleur	Paramount	Dana Carvey
Wayne's World	1992	95	couleur	Paramount	Mike Meyers

On peut en déduire les trois DF suivantes:

{titre, année} → durée
{titre, année} → type
{titre, année} → studio

Par contre, la dépendance suivante n'est pas une DF:

{titre, année} → acteurPrincipal

En effet, étant donné un film, il peut exister dans la base (telle qu'elle est modélisée) plusieurs acteurs principaux.

Enfin, la DF {titre, année} → durée signifie qu'il ne peut exister dans une instance de *FILM* les deux n-uplets suivants:

titre	**année**	**durée**	**type**	**studio**	**acteurPrincipal**
Farenheit, 9/11	2004	124	couleur	Fox	Georges Bush
Farenheit, 9/11	2004	130	couleur	Fox	Dick Cheney

CONVENTIONS

- DF signifie dépendance fonctionnelle.
- On suppose que la DF X → ∅ est vraie quelque soit X.
- L'ensemble de DF suivant:
  $\{A_1,A_2,\ldots,A_n\} \rightarrow B_1$
  $\{A_1,A_2,\ldots,A_n\} \rightarrow B_2$
  …
  $\{A_1,A_2,\ldots,A_n\} \rightarrow B_m$
  est équivalent à $\{A_1,A_2,\ldots,A_n\} \rightarrow B_1,B_2,...,B_m$.

1.2.3 Propriétés des DF

Les trois premières propriétés (réflexivité, augmentation et transitivité) sont également appelés axiomes d'Armstrong. De ces trois axiomes, on peut en déduire trois autres propriétés (pseudo-transitivité, union et décomposition).

RÉFLEXIVITÉ

Si $Y \subseteq X$, alors on a $X \rightarrow Y$

AUGMENTATION

Si $X \rightarrow Y$, alors on a $X \cup Z \rightarrow Y \cup Z$

TRANSITIVITÉ

Si $X \rightarrow Y$ et $Y \rightarrow Z$, alors on a $X \rightarrow Z$

PSEUDO-TRANSITIVITÉ

Si $X \rightarrow Y$ et $Y \cup W \rightarrow Z$, alors $X \cup W \rightarrow Z$

UNION

Si $X \rightarrow Y$ et $X \rightarrow Z$, alors $X \rightarrow Y \cup Z$

DÉCOMPOSITION

Si $X \rightarrow Y$ et $Z \subseteq Y$, alors $X \rightarrow Z$

1.2.4 Décomposition binaire d'une relation

Si $R(X,Y,Z)$ et $X \rightarrow Y \Rightarrow R(X,Y,Z) = R[X,Y] * R[X, Z]$, alors:

- on peut toujours décomposer une relation suivant une dépendance fonctionnelle,
- on ne peut décomposer une relation s'il n'y a pas de dépendance fonctionnelle,
- la décomposition suivant une dépendance fonctionnelle ne perd pas d'information,
- ce principe de décomposition binaire d'une relation est à la base de l'algorithme de décomposition.

1.2.5 Dépendances Fonctionnelles particulières

DÉPENDANCE FONCTIONNELLE TRIVIALE

Le premier axiome d'Armstrong exprime que si $Y \subseteq X$, alors il existe une DF $X \rightarrow Y$ qui est dite **triviale**.

DÉPENDANCE FONCTIONNELLE CANONIQUE

Une DF $X \rightarrow Y$ est **canonique** ssi sa partie droite ne comporte qu'un seul attribut.

DÉPENDANCE FONCTIONNELLE ÉLÉMENTAIRE (DFE)

Soient $R(X,Y,Z)$ un schéma de relation, X, Y, Z des ensembles d'attributs avec Z éventuellement vide et A un attribut quelconque.

Une DF $X \rightarrow Y$ (avec X ne contenant pas Y) est une **dépendance fonctionnelle élémentaire** (DFE) ssi il n'existe aucun sous-ensemble de X ayant une DF sur Y (X est la plus petite quantité d'information donnant Y).

DÉPENDANCE FONCTIONNELLE DIRECTE

Une DF $X \rightarrow Y$ est **directe** s'il n'existe pas d'attributs Z tel que:

$X \rightarrow Z$ et $Z \rightarrow Y$

EXEMPLE

La DF {titre, année} → titre est triviale. La DF {titre, année} → durée est canonique. Elle est également élémentaire et directe.

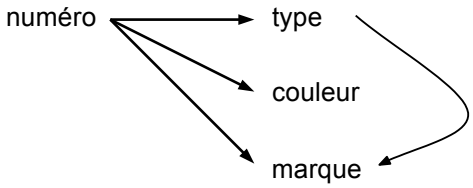
1.2.6 Graphe de Dépendances Fonctionnelles

On peut représenter un ensemble F de DF d'une relation *R* sous la forme d'un graphe dont les sommets sont les attributs de la relation et dont un arc de X vers Y représente la DF $X \rightarrow Y$.

Soit la relation *VOITURE* (numéro, marque, couleur, type), avec l'ensemble F des DF:

numéro → marque
numéro → couleur
numéro → type
type → marque

Le graphe associé est représenté ainsi:



Sur le graphe, on observe clairement que la DF numéro → marque est transitive.

1.2.7 Fermeture des DF

Étant donné un ensemble F de DF, la fermeture transitive de l'ensemble F, notée F+, est l'ensemble des DFE qui peuvent être logiquement induites par application des axiomes d'Armstrong sur l'ensemble F.

F+ = {X → Y | X → Y peut être déduite de F à l'aide des axiomes de Armstrong}

Deux ensembles de DF F et G sont dits **équivalents** lorsqu'ils possèdent la même fermeture.

1.2.8 Couverture minimale

Souvent, on constate que soit certaines dépendances fonctionnelles sont redondantes, soit que certaines de ces dépendances peuvent être déduites de celles qui sont exprimées. Il est donc intéressant de définir un ensemble de DF équivalent à l'ensemble initial qui soit ne composé que de dépendances fonctionnelles « pertinentes ». On nomme cet ensemble la **couverture minimale** de F. On définit la couverture minimale de F comme l'ensemble G de DF tel que:

1. G+ = F+,
2. toute DF est canonique,
3. aucune dépendance de G n'est redondante (i.e., pour toute DF g de G, G-g n'est pas équivalent à G),
4. toute dépendance de G est élémentaire.

La couverture minimale va être un élément essentiel pour décomposer les relations sans perte d'information.

EXEMPLE

Soit l'ensemble F de DF suivant:

A → B (1)
B,C → D (2)
D → E (3)
A,C → D (4)
A,C → E (5)

La couverture minimale de F est l'ensemble de DF suivant:

A → B
B,C → D
D → E

En effet, (4) est redondante, par augmentation de (1) par C et transitivité avec (2). De même, (5) peut être obtenue par transitivité de (4) et (3).

1.2.9 Clés

Les **clés minimales** (ou clés) sont des cas particuliers des DF. Soit X un ensemble d'attributs d'une relation R. X est une **clé minimale** de R ssi X satisfait les conditions suivantes:

1. identification **unique**: $X \rightarrow R$ est vraie,
2. **minimalité** (non-redondance): il n'existe pas de sous-ensemble Y de X tel que $Y \rightarrow R$ est vraie.

Lorsque *R* possède au moins deux clés minimales, une est désignée comme étant la clé primaire.

Une **clé étrangère** est un attribut (ou un groupe d'attributs) appartenant à une relation *R1* et qui apparaît comme clé primaire dans une relation *R2*.

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

P1: tout attribut qui ne figure pas dans le membre droit d'une DF non triviale ($X \rightarrow Y$ et $Y \notin X$) de F doit appartenir à toute clé de *R*,

P2: si l'ensemble des attributs de *R* qui ne figurent pas en membre droit d'une DF non triviale de F est une clé, alors *R* possède une clé minimale unique composée de l'ensemble de ces attributs.

EXEMPLE

Soit la relation *R*(ville, région, code). Les DF {ville, région} → code et code → ville sont vérifiées. {ville, région} est une clé minimale étant donné que {ville, région} → {ville, région, code} est vérifiée et que ni {ville} → {ville, région, code} et {région} → {ville, région, code} ne sont vérifiées.

ATTRIBUTS

A est **attribut clé** s'il appartient à au moins une clé de *R*.

A est **attribut non clé** s'il n'appartient pas à une clé de *R*

A est **transitivement dépendant** de X s'il existe Y, Y ne contenant pas A tel que $X \rightarrow Y$, $Y \rightarrow A$, $\neg(Y \rightarrow X)$, Y n'est pas inclus dans X.

A est **directement dépendant** de X s'il n'est pas transitivement dépendant.

A est **pleinement dépendant** de X si $X \rightarrow$ A est une DFE.

A est **partiellement dépendant** de X si $X \rightarrow$ A n'est pas une DFE.

1.3 Décomposition de schémas relationnels

Pour éliminer les anomalies présentées précédemment, on *décompose* les relations. La décomposition de *R* implique la séparation des attributs de *R* en deux nouvelles relations. Étant donnée une relation *R* composée de $\{A_1,A_2,\ldots,A_n\}$, on peut décomposer *R* en deux relations *S* et *T* dont les attributs sont $\{B_1,B_2,\ldots,B_m\}$ et $\{C_1,C_2,\ldots,C_k\}$ tels que:

- $\{A_1,A_2,\ldots,A_m\} = \{B_1,B_2,\ldots,B_m\} \cup \{C_1,C_2,\ldots,C_k\}$,
- les n-uplets de la relation *S* sont la *projection* sur $\{B_1,B_2,\ldots,B_m\}$ de tous les n-uplets de *R*. La projection est l'opération qui consiste à construire la nouvelle relation *S* à partir de *R* en prenant pour chaque n-uplet *t* de *R* uniquement les attributs $\{B_1,B_2,\ldots,B_m\}$ et en éliminant les n-uplets en double,
- de manière similaire, les n-uplets de la relation *T* sont la *projection* sur $\{C_1,C_2,\ldots,C_k\}$ de tous les n-uplets de *R*.

La décomposition de la relation *R* en $R_1,R_2,\ldots,R_n$ est sans perte si la *jointure* des relations $R_1,R_2,\ldots,R_n$ permet de retrouver tous les n-uplets de la relation *R* d'origine. La jointure est l'opération qui permet de construire une nouvelle relation à partir de deux relations ayant au moins un attribut commun.

EXEMPLE

On peut décomposer la relation *FILM* définie précédemment en deux relations *FILM1* et *FILM2* telles que:

FILM1 = (titre, année, durée, type, studio)
FILM2 = (titre, année, acteurPrincipal)

Si on utilise la décomposition telle qu'elle est définie, on obtient au niveau des n-uplets pour *FILM1* et *FILM2* les résultats suivants:

titre	année	durée	type	studio
StarWars	1977	124	couleur	Fox
Mighty Ducks	1991	104	couleur	Disney

titre	année	acteurPrincipal
StarWars	1977	Carrie Fisher
StarWars	1977	Mark Hamill
StarWars	1977	Harrison Ford
Mighty Ducks	1991	Emilio Estevez
Wayne's World	1991	Dana Carvey
Wayne's World	1992	Mike Meyers

On peut remarquer que cette décomposition élimine les problèmes soulevés au paragraphe 1.2.1. La redondance est éliminée, par exemple la durée de chaque film n'apparaît qu'une fois, dans la relation *FILM1*. Le risque d'anomalie de mise à jour a ainsi disparu. De même, le risque d'anomalie de suppression a disparu. Si on supprime tous les acteurs du film Wayne's World, le film disparaît également. Il apparaît que *FILM2* possède malgré tout de la redondance, puisque titre et année apparaissent plusieurs fois. Cependant, ces deux attributs forment une clé, et il n'est pas possible de représenter plus succinctement un film.

1.4 Normalisation des relations

Les formes normales de schémas de relations sont des propriétés souhaitables que doit posséder chaque schéma de relations. Elles sont nécessaires (mais non suffisantes) pour une « bonne » conception. Historiquement, les formes normales suivantes ont été proposées (de la moins restrictive à la plus restrictive).

Première Forme Normale (1NF)
Deuxième Forme Normale (2NF)
Troisième Forme Normale (3NF)
Forme Normale de Boyce and Codd (BCNF)
Quatrième Forme Normale (4NF)
Cinquième Forme Normale (5NF)

Les formes normales définissent un ordre partiel sur les schémas de relation. On peut donc voir une forme normale comme une classe d'équivalence (on peut comparer deux schémas dans deux classes d'équivalence différentes mais pas dans la même). Il faut aussi noter que le seul élément qui est pris en compte par les formes normales est la non redondance d'informations d'un schéma. Selon les formes normales, un « bon » schéma est un schéma sans redondance (ce qui ne veut pas forcément dire qu'il est efficace par exemple). Un schéma relationnel sans qualité particulière est appelé schéma en 1ère forme normale (on note 1NF) et si on rajoute certaines qualités on obtient les deuxième et troisième formes normales (on note 2NF et 3NF).

On ne présente ici que les formes normales dont la définition utilise exclusivement les dépendances fonctionnelles. Si on prend en compte d'autres dépendances entre données comme les dépendances multivaluées, on obtient alors les 4NF et 5NF. La BCNF reste cependant l'objectif de normalisation le plus « classique ».

1NF: une relation est en première forme normale ssi tout attribut a une valeur atomique (vrai par définition du modèle relationnel).

2NF : une relation est en deuxième forme normale ssi elle est en 1NF et si tous les attributs non clés sont pleinement dépendants des clés (toutes les dépendance entre une clé et un attribut non clé sont des dépendances fonctionnelles élémentaires).

3NF : une relation est en troisième forme normale ssi elle est en 2NF et si tous les attributs non clés sont directement et pleinement dépendants des clés (si tout attribut non clé ne dépend pas d'un autre attribut non clé).

BCNF (*Boyce-Codd*) : une relation est en BCNF ssi elle est en 3NF et si les seules DFE sont celles de la forme C → X avec C clé (seules les clés sont en partie gauche de DF).

1.4.1 Première forme normale (1NF)

Une relation *R* est dite en **première forme normale** (1NF) si tous ses attributs sont atomiques et qu'il n'existe pas d'attributs répétitifs. De plus, chaque attribut doit avoir une sémantique précise. Voyons pourquoi la relation suivante n'est pas en première forme normale.

titre	**année**	**durée**	**type**	**studio**
StarWars	1977	124	couleur	Fox, Los Angeles
Mighty Ducks	1991	104	couleur	Disney, San Francisco
Wayne's World	1992	95	couleur	Paramount, Los Angeles

Les attributs de cette relation ne sont pas atomiques. En effet, l'attribut **studio** contient à la fois le nom et la ville, ce qui est gênant si l'on souhaite extraire tous les studios situés à Los Angeles.

Analysons maintenant un deuxième exemple. La relation suivante n'est pas non plus en 1NF.

titre	**année**	**acteur1**	**acteur2**	**acteur3**
StarWars	1977	Carrie Fisher	Mark Hamill	Harrison Ford
Mighty Ducks	1991	Emilio Estevez		
Wayne's World	1992	Dana Carvey	Mike Meyers	

En effet, la première forme normale stipule que les attributs ne doivent pas être répétitifs. Or, la liste des acteurs ne respecte pas cette propriété. Si on veut rechercher dans quel film a joué Harrison Ford, on est obligé non seulement de parcourir tous les n-uplets, mais également toutes les colonnes. De plus, si l'on souhaite rajouter un quatrième acteur, il faut modifier la relation toute entière.

Enfin, que pensez-vous de la relation suivante:

titre	**année**	**durée**	**type**	**studio**
StarWars	1977	124	SF	Fox
Mighty Ducks	1991	104	couleur	Disney
Wayne's World	1992	95	couleur	Paramount

Il semble que cette relation soit en 1NF. Les attributs sont atomiques, il n'y a pas d'attributs répétitifs. Malgré tout, l'attribut **type** ne possède pas une sémantique claire, représentant soit un genre (SF), soit le type de la pellicule. Cette relation n'est donc pas en 1NF.

1.4.2 Deuxième forme normale (2NF)

Une relation *R* est dite en **deuxième forme normale** (2NF) ssi

- elle est en 1NF,
- tout attribut n'appartenant pas à une clé ne dépend pas d'une partie de cette clé (pas de dépendances partielles).

Autrement dit, dès qu'un attribut non clé dépend d'une partie d'une clé, la relation n'est pas en 2NF.

Voyons dans un premier temps un exemple abstrait. La relation *R*(A,B,C) munie de l'ensemble F des dépendances est en 1NF mais pas en 2NF.

A,B → C
B → C

Regardons maintenant sur un exemple concret. Considérons la relation suivante:

numSalarié	**nom**	**numProjet**	**heures**
20036	Durand	1	18,5
20036	Durand	2	6,7
36900	Leroux	2	8,5
45002	Franck	3	23,5
45002	Franck	1	4,8

L'ensemble F des DF est:

numSalarié → nom
numSalarié,numProjet → heures

La relation n'est pas en 2NF. En effet, la clé minimale de cette relation est {numSalarié,numProjet}. L'attribut non clé **heure** est en totale dépendance fonctionnelle avec la totalité de la clé. Par contre, l'attribut non clé **nom** dépend d'une partie de la clé, à savoir **numSalarié**. Pour que la relation soit en 2NF, il est nécessaire de scinder la relation initiale en deux relations, comme ceci:

numSalarié	**nom**
20036	Durand
36900	Leroux
45002	Franck

numSalarié	**numProjet**	**heures**
20036	1	18,5
20036	2	6,7
36900	2	8,5
45002	3	23,5
45002	1	4,8

Chacune de ces deux relations répond désormais aux exigences de la première et de la deuxième forme normale. La première relation a pour clé primaire l'attribut **numSalarié**, et la deuxième relation a pour clé {numSalarié,numProjet}.

1.4.3 Troisième forme normale (3NF)

OBJECTIFS DE LA 3NF

1. *élimination des dépendances partielles*

   S'il existe une dépendance partielle Y → A, où X est une clé minimale et Y un sous-ensemble de X, alors chaque n-uplet utilisé pour associer une valeur X aux valeurs de A sera répété pour les valeurs de Y (d'où redondance des informations).
   La condition de la 3NF élimine cette possibilité, évitant ainsi les mises à jour incorrectes et la redondance.

2. *élimination des dépendances transitives*

   S'il existe une dépendance transitive X → Y → A, alors on ne peut associer une valeur Y à une valeur X que s'il existe une valeur A associée à une valeur Y. Cela conduit à des problèmes d'insertion et de suppression.

DÉFINITION

On considère les définitions suivantes:

R: un schéma de relations,
F: un ensemble de DF sur *R*,
X: un sous-ensemble de *R*,
A: un attribut.

On rappelle que A est un attribut clé ssi il fait partie d'une clé minimale de *R* et que dans le cas contraire, A est appelé un attribut non clé.

La relation R est dite en **troisième forme normale** (3NF) ssi lorsque X → A est dans F+ et A n'est pas dans X, soit X est une clé primaire, soit A est un attribut clé.

Il existe deux cas dans lesquels X → A ne respecte pas la 3NF:

Cas 1: X est un sous-ensemble d'une clé minimale. X → A est une dépendance partielle.

Cas 2: X n'est pas un sous-ensemble de toutes les clés minimales. X → A est une dépendance transitive. Elle crée une chaîne non triviale de DF Z → X → A pour une clé minimale Z.

Autrement dit, une relation est dite en 3NF ssi:

- elle est en 2NF,
- il n'existe pas de dépendance entre attributs non clé.

EXEMPLE 1

R=ABCD
F = {AB → C,B → D,BC → A}
AB et BC sont des clés minimales et ce sont les seules. A, B et C sont des attributs clés. D est un attribut non clé.

EXEMPLE 2

R=VRC *R*(ville, rue, code)
F = {VR → C,C → V}
R est en 3NF.

EXEMPLE 3

R=SAIP
F = {SI → P,S → A}
SI est la seule clé, et A est un attribut non clé.

S → A ne respecte pas la 3NF, étant donné que S n'est pas une clé primaire. De plus, S → A est une dépendance partielle.

On remarque dans cet exemple que la dépendance S → A est dans F. Cependant, en général, les dépendances non conformes à la forme normale étudiée sont dans F+ plutôt que dans F.

1.4.4 Forme normale de Boyce-Codd (BCNF)

OBJECTIFS DE LA BCNF

1. élimination des dépendances partielles,
2. élimination des dépendances transitives,
3. élimination des anomalies non prises en compte par la 3NF.

DÉFINITION

On considère les définitions suivantes:

R: un schéma de relations,
F: un ensemble de DF sur R,
X: un sous-ensemble de R,
A: un attribut.

R est en BCNF ssi lorsque X → A appartient à F+ et que A n'est pas dans X, alors X est une clé de *R*.

Une relation en 3NF qui ne possède qu'un seule clé minimale est en BCNF.

EXEMPLE 1

M=NAC *MEMBRE*(nom,adresse,cotisation)
F = {n → a, n → c}
M est en BCNF.

EXEMPLE 2

R=VRC *R*(ville,rue,code)
F = {VR → C,C → V}

R est en 3NF mais pas en BCNF. Dans R, on ne peut pas stocker la ville correspondante à un code à moins de connaître une rue associée à ce code (anomalie d'insertion). En effet, C → V est vraie et V n'appartient pas à C, alors que C ne contient aucune des clés minimales de R.

1.5 Dépendances fonctionnelles et conception de schémas

Nous l'avons déjà abordé au paragraphe 1.2.1, il existe deux méthodes pour concevoir un schéma de bases de données qui soit en 3NF: la méthode dite synthétique, et la méthode par décomposition.

1.5.1 La méthode synthétique

L'algorithme dit de synthèse permet d'obtenir une décomposition 3NF qui préserve les DF. Il est basé sur le calcul de la couverture minimale (ou irredondante) d'un ensemble de DF. Le point de départ de cette méthode est la connaissance des attributs et d'un ensemble de dépendances fonctionnelles. On construit le graphe des DF dans lequel chaque attribut constitue un noeud, et chaque DF est représentée par un arc. L'algorithme de synthèse va travailler sur ce graphe et construire un ensemble de relations en 3NF.

ALGORITHME DE SYNTHÈSE

Pour obtenir un schéma de relations en 3NF, on applique l'algorithme suivant:

1. on construit une couverture minimale à partir de l'ensemble F des DF,
2. on en déduit les relations en 3NF en regroupant dans une même relation tous les attributs ayant la même partie gauche dans le graphe des DF,
3. si aucune des relations obtenues à l'étape précédente ne contient pas (ou ne permet pas d'obtenir) la clé de la relation initiale, on ajoute une relation composée uniquement des attributs de la clé.

EXEMPLE

Soit la relation *GESPROD* (**numEmp**, nomEmp, **numProj**, budProj, emploi, taux, nbhr) qui décrit la gestion de projet dans une entreprise. Les attributs sont les suivants: le numéro d'employé, le nom de l'employé, le numéro du projet, le budget du projet, l'emploi occupé par l'employé, le taux de rémunération de l'employé, et le nombre d'heures de travail de l'employé sur un projet donné. La seule clé minimale de la relation est constituée des attributs {numEmp, numProj}. L'ensemble F des DF est le suivant:

numEmp → nomEmp
numEmp → emploi
emploi → taux
numEmp → taux
numProj → budProj
numEmp,numProj → nbHr

Pour obtenir une décomposition de *GESPROD* muni de F en un ensemble de relations en 3NF, on applique l'algorithme suivant:

1. construction de la couverture minimale: on élimine les dépendances transitives:
   numEmp → taux est une DF transitive car numEmp → emploi et emploi → taux.
2. déduction des relations en 3NF en regroupant dans une même relation tous les attributs ayant même partie gauche dans F.
   EMPLOYÉ (**numEmp**, nomEmp, emploi)
   PROJET (**numProj**, budProj)
   RÉMUNÉRATION (**emploi**, taux)
   OCCUPE (**numEmp**, **numProj**, nbHr)
3. on vérifie que la clé est présente dans au moins une relation. Dans notre exemple, la relation *OCCUPE* possède la clé de la relation initiale.

1.5.2 La méthode par décomposition

Une autre manière de concevoir un schéma relationnel en troisième forme normale est de partir du schéma complet (ensemble de tous les attributs) et de décomposer cette « grosse » relation (appelée également *relation universelle*) en respectant les dépendances fonctionnelles. Cette approche est appelée approche par décomposition. Le problème est d'ordonner l'ordre des décompositions de manière à obtenir un schéma en 3ème forme normale. En effet, chaque relation produite ne conserve qu'un certain nombre de DF (celles définies sur ses attributs propres) et n'est donc pas forcément en 3ème forme normale. De plus, l'ensemble des DF du schéma complet n'est pas forcément préservé.

ALGORITHME DE DÉCOMPOSITION

entrée: un schéma relationnel (ensemble d'attributs) et un ensemble E de DF entre ses attributs

sortie: une ou plusieurs relations en 3NF dont la jointure redonne la relation initiale (par contre des DF de E ont pu être perdues)

principe: l'algorithme peut se voir comme la construction d'un arbre binaire. La racine de cet arbre est la relation à décomposer. L'arbre se construit récursivement de la manière suivante:

1. on choisit une DF dfi dans l'ensemble E des DF,
2. le fils gauche du noeud racine est une relation composé de tous les attributs de dfi,
3. dfi est retirée de l'ensemble E,
4. le fils droit du noeud racine est une relation composée de tous les attributs de la racine excepté ceux présents en partie droite de dfi.

Problèmes: la solution dépend du choix des DF selon lesquelles on choisit de décomposer et il ne préserve pas nécessairement les DF.

On sait néanmoins que toute relation admet une décomposition en 3NF qui préserve les DF.

EXEMPLE

Soit la relation universelle associée à la gestion de production définie au paragraphe 1.5.1:

GESPROD (**numEmp**, nomEmp, **numProj**, budProj, emploi, taux, nbhr)

On constate qu'elle n'est pas en 2NF car

numEmp → nomEmp, taux, emploi

numProj → budProj

Ceci conduit à une décomposition de la relation *GESPROD* en trois relations en 2NF:

PROJET (**numProj**, budProj)

OCCUPE (**numEmp**, **numProj**, nbHr)

EMPREM (**numEmp**, nomEmp, emploi, taux)

On constate que la relation *EMPREM* n'est pas en 3NF car

emploi → taux (un attribut non clé dépend fonctionnellement d'un autre attribut non clé)

On décompose la relation *EMPREM* en deux relations en 3NF

EMPLOYÉ (**numEmp**, nomEmp, emploi)

RÉMUNÉRATION (**emploi**, taux)

On obtient finalement quatre relations en 3NF

PROJET (**numProj**, budProj)

OCCUPE (**numEmp**, **numProj**, nbHr)

RÉMUNÉRATION (**emploi**, taux)

EMPLOYÉ (**numEmp**, nomEmp, emploi)

EXEMPLE SUR LES FORMES NORMALES

Soit le schéma *R* = (P, H, N, Y, T), et F = {P → T | P, H → Y | H, N → P | H, Y → N}.

- ensemble des DFE engendrées:

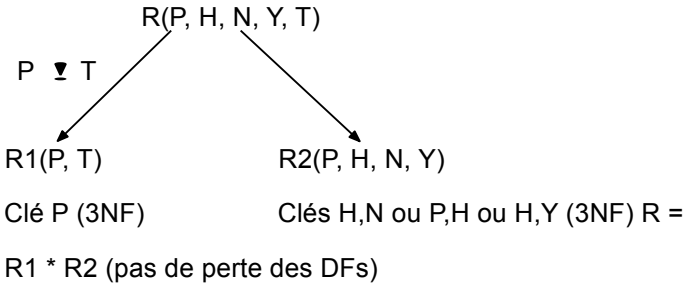
  H, N → T

  P, H → N

  H,N → Y

H, Y → P
P, H → T
H, Y → T

- on a donc trois clés potentielles (H, N ⌈ P, H ⌈ H, Y):
  H, N → P, T, Y
  P, H → T, Y, N
  H, Y → N, P, T
- les attributs clés sont donc: H, N, P, Y, et un attribut non clé est T.
- par définition le schéma est en 1ère forme normale. Est il en 2NF ?
  Non, car P, H → T n'est pas une DFE (on a P → T), R est donc en 1NF.
- application de l'algorithme de décomposition:



CHAPITRE 2: SÉCURITÉ DE FONCTIONNEMENT

1. Les transactions
2. La tolérance aux pannes
3. La gestion de la concurrence
4. Le verrouillage sous Oracle
5. JDBC et les transactions

2.1 Les transactions

2.1.1 Définition

L'objectif de ce chapitre est d'étudier les mécanismes que fournit un SGBD pour s'assurer que les données demeurent dans un état cohérent, c'est-à-dire qu'elles ne sont pas mises en cause par un problème logiciel ou matériel, ou par le fait que plusieurs utilisateurs manipulent les données simultanément. En effet, les SGBD sont utilisés par plusieurs utilisateurs simultanément, à la fois pour les requêtes ou pour des modifications sur la base de données.

Par exemple, on suppose que deux personnes souhaitent retirer 100€ d'un même compte. Un SGBD doit s'assurer qu'aucun des deux retraits ne soit perdu. Par comparaison avec un système d'exploitation, deux personnes peuvent très bien éditer le même fichier, mais c'est le dernier qui écrit qui voit ses modifications conservées. Ce comportement n'est pas imaginable dans le contexte des bases de données.

Pour résoudre ce problème, on a introduit le concept de **transaction**; Une **transaction** est un programme qui forme une unité logique de traitement. Elle est constituée d'une ou plusieurs opérations d'accès aux objets de la base de données (insertion, suppression, modification ou lecture). Enfin, une transaction fait passer la base d'un état cohérent E1 à un état cohérent E2. En particulier:
- une transaction doit commencer à partir d'un état cohérent de la base,
- pendant la transaction, la base peut éventuellement être dans un état incohérent,
- lorsque la transaction se termine avec succès, la base doit être dans un état cohérent,
- lorsqu'une transaction a été validée, les modifications sur la base sont persistantes, même en cas de problèmes logiciels ou matériels,
- plusieurs transactions peuvent s'exécuter en parallèle.

On distingue deux problèmes à prendre en compte:
- *la tolérance aux pannes*: il faut restaurer la base dans l'état où elle se trouvait avant l'incident. Les mécanismes utilisés sont les transactions, la tenue d'un journal, la reprise après panne.
- *l'exécution concurrente de plusieurs transactions*: il faut veiller à ce que l'action simultanée de plusieurs utilisateurs sur les mêmes données ne conduise pas à une incohérence de ces données. Les mécanismes utilisés sont la gestion des accès concurrents, les transactions, le verrouillage et la gestion de l'interblocage.

2.1.2 Propriétés ACID

Un SGBD supporte ce que l'on appelle des transactions ACID. Les propriétés ACID sont:
- *atomicité* : lors de l'exécution d'une transaction, soit toutes ses actions sont exécutées, soit aucune ne l'est,
- *cohérence* : après une transaction, la base se trouve dans un état cohérent,
- *isolation* : il n'y a pas d'incohérence liée à la concurrence d'accès,
- *durabilité* : les effets d'une transaction correctement exécutée survivent à une panne ultérieure.

Voyons sur un exemple ce que cela signifie. On suppose une transaction qui effectue un transfert d'argent (100€) d'un compte A vers un compte B:

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

La propriété d'atomicité spécifie que si la transaction échoue après l'étape 3 et avant l'étape 6, le système doit s'assurer qu'aucune mise à jour n'aura été effectuée, sous peine de voir une incohérence entre la balance des deux comptes.

La propriété de cohérence spécifie que la somme des soldes de A et B est inchangée après l'exécution de la transaction.

La propriété d'isolation spécifie que si une autre transaction essaie d'accéder à la base entre les étapes 3 et 6, elle ne verra pas les modifications faites aux étapes 1 à 3. L'isolation peut être assurée de manière triviale en exécutant les transactions en séquence (de manière sérialisable). Cependant, nous verrons que permettre l'exécution concurrente des plusieurs transactions apporte des avantages, à condition de ne pas avoir d'interférences entre les transactions simultanées.

Enfin, la propriété de durabilité spécifie que lorsque l'utilisateur a validé la transaction, les mises à jour doivent persister, quelques soient les défaillances logicielles ou matérielles qui pourraient survenir.

2.1.3 États d'une transaction

Une transaction est une unité atomique (au sens indivisible) de travail qui doit être soit totalement terminée soit pas du tout (atomicité). Le système doit donc savoir quand la transaction commence, se termine, est validée ou est abandonnée. Pour cela, une transaction peut se trouver dans plusieurs états:

- *active*: l'état initial. La transaction entre dans cet état immédiatement après son début d'exécution, et elle y reste durant son exécution,
- *partiellement validée*: lorsque la dernière instruction de la transaction a été exécutée,
- *échouée*: dès que l'exécution ne peut plus continuer normalement,
- *annulée*: après que la transaction a été annulée, et que la base ait été restaurée dans l'état cohérent d'avant le début de la transaction. Il existe alors deux options après qu'une transaction ait été annulée, soit on redémarre la transaction, soit on tue la transaction.
- *validée*: après une exécution totalement terminée.

En fait, SQL supporte les transactions, souvent de manière invisible. En effet, chaque commande effectuée dans l'outil d'interrogation est une transaction. Ainsi, chaque fois que vous faites une (ou plusieurs) requête(s) dans SQL/Plus, vous débutez sans le savoir une transaction.

2.1.4 Commit, rollback

L'instruction SQL `COMMIT` permet de valider une transaction. Cette instruction peut être explicite dans une transaction où, comme c'est souvent le cas, implicite et exécutée à la fin de la transaction par le SGBD. Une fois l'instruction `COMMIT` exécutée, les modifications faites sont permanentes et la transaction s'achève.

L'instruction `ROLLBACK` permet également d'achever la transaction, mais en l'annulant. Il n'y alors aucun effet sur la base, et le SGBD doit alors revenir à l'état précédent le début de la transaction puisque c'est le dernier point garanti de cohérence de la base.

2.1.5 Exemple

Voyons sur un exemple l'interaction entre deux transactions. On suppose la relation *BarÀBière*(bar, bière, prix), et on suppose que le bar Jo vend de la Chimay (3,00€) et de la Leffe (2,50€). Paul interroge la base, et la relation *BarÀBière* pour connaître le prix minimum et le prix maximum du bar Jo. Entretemps, Jo décide de cesser de vendre de la Chimay et de la Leffe, pour ne plus vendre que de la Kwak à 3,50€.

Paul exécute les deux instructions SQL suivantes afin de connaître les prix minimum et maximum:

```
(max)     SELECT MAX(prix) FROM BarÀBière
          WHERE bar = 'Jo';
(min)     SELECT MIN(prix) FROM BarÀBière
          WHERE bar = 'Jo';
```

Pendant ce temps, Jo exécute les instructions suivantes:

```
(del)     DELETE FROM BarÀBière
          WHERE bar = 'Jo';
(ins)     INSERT INTO BarÀBière
          VALUES('Jo', 'Kwak', 3.50);
```

On a des instructions SQL qui peuvent être entrelacées, avec les contraintes suivantes: (max) doit être exécuté avant (min), et (del) doit être exécuté avant (ins). Que se passe-t-il lorsque Jo et Paul mettent à jour la base ? Si les instructions ont lieu dans l'ordre suivant: (max)(del)(ins)(min), on obtient:

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

Ainsi, Paul voit que MAX < MIN !

Comment peut-on résoudre le problème avec les transactions ? Si l'on regroupe les instructions (max)(min) de Paul dans une seule transaction, alors il n'y aura plus de problèmes. En effet, soit la transaction s'exécutera avant les modifications de Jo, et Paul verra alors les anciens prix, soit la transaction s'exécute après la modification, et il n'y aura pas de problème. La transaction permet donc aux deux instructions (max)(min) de s'exécuter sur les mêmes données.

Il existe cependant encore une possibilité de comportement étrange. Supposons que Jo exécute sa modification de prix, mais qu'au dernier moment, il change d'avis et exécute un ROLLBACK. Si Paul exécute sa transaction après l'instruction (ins), mais avant l'annulation, il va voir une valeur 3,50 qui n'existe pas dans la base. Pour résoudre ce problème, on va donc exécuter les instructions de Jo dans une transaction. Ainsi, les effets de (del)(ins) dans une transaction ne pourront être vus par d'autres transactions que lorsque la transaction aura été validée. Si la transaction venait à être annulée, ses effets ne seraient pas vus par les autres transactions.

2.1.6 Gestion des transactions dans Oracle

Une transaction débute avec la première instruction SQL exécutable et se termine lors de l'exécution d'un **commit** ou d'un **rollback**, lors de la déconnexion de l'utilisateur (commit), ou lors d'un arrêt anormal du processus utilisateur (rollback).

Les mises à jour faites par une transaction ne sont visibles pour les autres utilisateurs qu'après la validation de la transaction (tant qu'une transaction n'est pas validée, les autres utilisateurs accèdent aux anciennes valeurs des données modifiées).

2.2 La tolérance aux pannes

2.2.1 Journal

Il existe plusieurs types d'incidents qui peuvent survenir sur une base de données :

- échec d'une transaction: une erreur logique dans le programme l'empêche de se poursuivre normalement (erreur lors d'une insertion, suppression ou mise à jour), ou une erreur système arrête le déroulement normal du programme (verrouillage, saturation, processus tué),
- crash système: il entraîne la perte des données en mémoire centrale,
- panne disque: elle provoque la perte physique des données.

Les algorithmes de récupération (*recovery*) sont des techniques qui permettent aux SGBD d'assurer la cohérence, l'atomicité et la durabilité malgré les incidents qui peuvent survenir. En général, les algorithmes de récupération possèdent deux parties: la première prend en charge les actions qui ont lieu lors du déroulement normal de la transaction afin d'avoir assez d'informations pour permettre la récupération en cas de problème, et la deuxième qui prend en charge les actions à effectuer après un problème afin de pouvoir récupérer la base dans un état cohérent.

Pour assurer la reprise sur panne, les SGBD et Oracle en particulier utilisent la notion de **journal**. Un **journal** est un fichier texte dans lequel le SGBD inscrit, dans l'ordre d'exécution, toutes les actions de mise à jour qu'il effectue. Ainsi, le journal est une séquence d'enregistrements qui mémorise les mises à jour faites sur la base. Lorsqu'une transaction T_i démarre, elle s'enregistre dans le journal sous la forme <*début transaction*, T_i >. Avant que la transaction T_i exécute une instruction écrire(X), un enregistrement <T_i, X, V_1, V_2> est écrit dans le journal (V_1 est la valeur de X avant l'écriture, et V_2 est la valeur de X après l'écriture). Lorsque la transaction T_i a lu une valeur, un enregistrement <*lecture*, T_i, X> est écrit dans le journal. Enfin, <*valider*, T_i> ou <*annuler*, T_i> sont écrits dans le journal suivant que la transaction a été validée ou annulée.

En Oracle, la gestion des transactions utilise deux fichiers :
- un fichier journal **redo log**: il enregistre toutes les modifications effectuées (insert, update, delete) dans les tables de la base par chaque transaction.
- un fichier **rollback segment**: il enregistre les anciennes valeurs des lignes des tables modifiées par une transaction.

Le système affecte à chaque transaction un numéro chronologique (SCN – *System Control Number*) qui permet de l'identifier de manière unique. Toute opération modifiant une donnée de la base est enregistrée dans le fichier **rollback segment** et dans le fichier journal **redo log** avant mise à jour de la base. Le journal **redo log** est chronologique.

Lors du **commit** d'une transaction, il faut :
- enregistrer le commit de la transaction dans les fichiers **redo log** et **rollback segment**,
- débloquer les lignes des tables verrouillées par la transaction.

Lors du **rollback** d'une transaction, il faut :
- redonner les anciennes valeurs à toutes les lignes des tables modifiées par la transaction (à partir des **rollback segment**),
- débloquer les lignes des tables verrouillées par la transaction.

2.2.2 Reprise après incident

Elle est disponible uniquement pour les incidents sans perte de données sur disque. L'objectif est de restaurer la base dans un état cohérent telle qu'elle se trouvait juste avant la panne. Le mécanisme de reprise est le suivant:
- appliquer toutes les modifications enregistrées dans le fichier **redo log** à la base de données (*roll-forward*)
- pour toutes les transactions sans commit, on remet les lignes des tables concernées aux anciennes valeurs (*rollback*). Ceci est fait à partir des **rollback segment**.

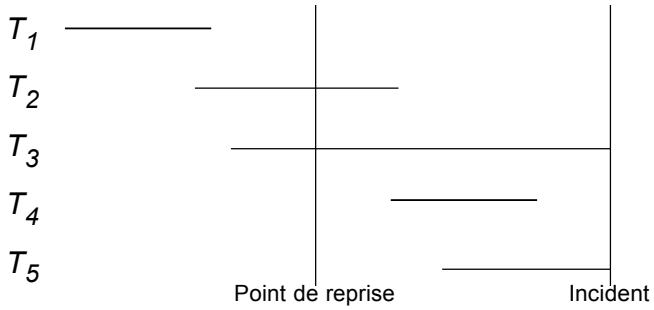
Après un incident, on ne traite que les modifications effectuées après le point de reprise (ou point de contrôle). Un point de reprise consiste à 1) écrire les buffers du fichier **redo log** sur disque, 2) écrire les buffers de la base de données sur disque, 3) écrire un article point de reprise dans le fichier **redo log**.

On peut préciser un point de reprise en utilisant la commande `SAVEPOINT pdr ;`. Par la suite, on pourra revenir à ce point en utilisant la commande `ROLLBACK TO pdr;` qui permet de retrouver l'environnement tel qu'il était lors de la déclaration du point de reprise.

SQL	**Résultats**
SAVEPOINT a;	Premier point de reprise de la transaction
DELETE …;	Modification de la base
SAVEPOINT b;	Deuxième point de reprise de la transaction
INSERT INTO…;	Modification de la base
SAVEPOINT c;	Troisième point de reprise de la transaction
UPDATE …;	Modification de la base

ROLLBACK TO c;	L'ordre UPDATE est annulé, le point C reste défini
ROLLBACK TO b;	L'ordre INSERT est annulé, le point C est perdu, le point B reste défini
ROLLBACK TO c;	ORA-01086 error; savepoint C no longer defined
INSERT TO…;	Modification de la base
COMMIT;	Valide toutes les actions effectuées par le DELETE (première modification) et le INSERT (dernière modification). Toutes les autres modifications ont été annulées avant le COMMIT. Le point A n'est plus actif.

EXEMPLE: traitement des transactions
La mise en œuvre du mécanisme de reprise s'effectue comme ceci. Soient les 5 transactions suivantes au moment d'un incident :



- T1 est achevée lors du point de reprise, elle n'est pas traitée.
- T2 et T4 sont validées (**commit**) lorsque survient l'incident: on applique toutes les modifications enregistrées dans le fichier redo log à la base de données (**rollforward**); pour T2, on repart du point de reprise.
- T3 et T5 ne sont pas validées (pas de **commit**) lorsque survient l'incident, on remet les lignes des tables concernées aux anciennes valeurs (**rollback**).

2.3 La gestion de la concurrence

La gestion des accès concurrents aux données permet d'éviter que la mise à jour simultanée des mêmes données par plusieurs utilisateurs n'introduise pas des incohérences dans la base de données. On dit que deux transactions sont concurrentes si elles accèdent simultanément aux mêmes données.

2.3.1 Accès concurrents et incohérence des données

Voici quelques problèmes classiques qui peuvent survenir avec des transactions concurrentes.

La perte de mise à jour
Il y a perte de mise à jour quand une transaction T2 vient écraser une écriture effectuée par T1.

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

Après une exécution séquentielle de T1 et T2 ou T2 et T1, on obtient A=70. Si les transactions sont exécutées de manière concurrentes, on obtient alors A=60. On a donc perdu la première mise à jour.

Les lectures impropres
Supposons une transaction T2 qui lit une donnée A modifiée par T1. La transaction T1 est annulée et les modifications effectuées par cette transaction sont défaites. La valeur lue et traitée par T2 est alors incorrecte.

Temps	Transaction T1	État de la base	Transaction T2
t1	lire(A)	A = 10	…
t2	A = A +20		…
t3	écrire(A)	A = 30	…
t4	…		lire(A)
t5	rollback	A = 10	…

Les lectures non reproductibles
La transaction T2 lit la même donnée A à deux instants différents et n'obtient pas la même valeur. Entre les deux lectures, une autre transaction a modifié la valeur de A.

Temps	Transaction T1	État de la base	Transaction T2
t1	…	A = 10	lire(A)
t2	lire(A)		…
t3	A = A + 10		…
t4	écrire(A)	A = 20	…
t5	commit		lire(A)

2.3.2 Le mécanisme de verrouillage

Les principales techniques mises en œuvre pour le contrôle de l'exécution concurrente des transactions reposent sur le concept de verrouillage des données. Un **verrou** est une variable associée à une donnée qui décrit l'état de la donnée, compte tenu des opérations qui peuvent lui être appliquées. Ainsi, avant d'accéder à une donnée pour modifier sa valeur, la transaction T1 doit la verrouiller pour éviter son accès par une autre transaction. Une fois terminée, la transaction T1 libère alors les données verrouillées pour les rendre disponibles à d'autres transactions.

Nous reprenons l'exemple du transfert d'une somme d'argent entre deux comptes: T1 fait un transfert de c1 vers c2, T2 de c1 vers c3.

T1	**T2**
lock(c1)	
read(c1) for update	
c1:=c1-m1	T2 attend la libération de c1
écrire(c1)	(demande accès pour update de c1)
lock(c2)	"
read(c2) for update	"
c2:=c2+m1	"
écrire(c2)	"
unlock(c1,c2)	lock(c1)
	read(c1) for update
	c1:=c1-m2
	écrire(c1)
	lock(c3)
	read(c3) for update
	c3:=c3+m2
	écrire(c3)

unlock(c1,c3)

Le verrouillage tel qu'il est appliqué ci-dessus permet d'éliminer les pertes de mise à jour et les lectures impropres. Par contre, il ne permet pas d'éliminer les lectures non reproductibles.

Les verrous classiques sont les **verrous binaires**, c'est-à-dire des verrous qui peuvent avoir deux valeurs distinctes: verrouillé ou non verrouillé (1 ou 0). Un verrou est associé à chaque élément X. Si la valeur du verrou sur X est 0, alors il est possible d'accéder à l'élément. Le verrouillage binaire utilise deux opérations: `verrouillerÉlément` et `déverrouillerÉlément`. Une transaction demande l'accès à un élément X en exécutant pour commencer l'opération `verrouillerÉlément(X)`. Si le verrou est à 1, la transaction doit attendre. Si le verrou est à 0, le verrou est mis à 1, et la transaction est autorisée à accéder à l'élément X. Lorsque la transaction n'a plus besoin d'utiliser l'élément, elle exécute une opération `déverrouillerÉlément(X)` qui positionne le verrou à 0 afin que les autres transactions puissent accéder à X. Un verrou binaire applique donc une **exclusion mutuelle** sur une donnée.

On se rend compte rapidement que les verrous binaires sont trop restrictifs pour les bases de données, car certaines transactions ne veulent que lire certaines données. On distingue donc deux types de verrous, les **verrous partagés**, qui permettent à plusieurs transactions de lire une donnée verrouillée par une transaction, et les **verrous exclusifs**, qui permettent à la transaction qui a posé le verrou d'effectuer une opération d'écriture.

2.3.3 Le risque d'interblocage

Un interblocage (ou verrou mortel) survient lorsque *chaque* transaction T d'un ensemble de *n (n ≥ 2) transactions* attend une donnée verrouillée par une autre transaction T' de l'ensemble. Autrement dit, T1 attend une donnée verrouillée par T2, et T2 attend une donnée verrouillée par T1.

EXEMPLE : transfert entre deux comptes.
T1 transfère une somme m d'un compte c1 vers un compte c2, et T2 transfère une somme n de c2 vers c1. L'exécution simultanée des deux transactions conduit à la situation suivante :

T1	**T2**
lock(c1)	
read(c1)	
c1:=c1+m	lock(c2)
écrire(c1)	read(c2)
	c2:=c2+n
T1 attend libération de c2	écrire(c2)
"	T2 attend libération de c1
"	"
"	"

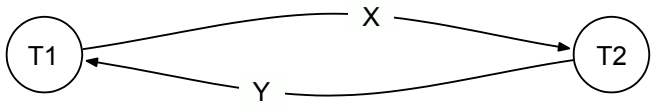
Les données sont verrouillées en fonction des demandes de mises à jour, mais ne sont libérées qu'en fin de transaction

2.3.4 Résolution de l'interblocage (*deadlock*)

Une approche pratique de la gestion des interblocages repose sur la détection de ceux-ci: le système vérifie s'il y a une situation d'interblocage. Un procédé pour détecter une situation d'interblocage consiste pour le système à construire et à maintenir un **graphe d'attente des ressources** (*wait-for graph*). Dans ce graphe, un nœud est créé pour chaque transaction en cours d'exécution. Chaque fois qu'une transaction Ti attend de pouvoir verrouiller une donnée X verrouillé par une transaction Tj, un arc est créé entre les deux transactions. Il y a situation d'interblocage si et seulement si il existe un cycle dans le graphe. Il se pose alors le problème de savoir quand le système doit vérifier la présence d'un interblocage (nombre de transactions, durée d'attente de plusieurs transactions,…). Si le système détecte l'interblocage, il tue une (ou plusieurs) des transactions impliquées. Il existe différentes solutions pour le choix de la transaction à tuer : la plus récente, celle qui a fait le moins de mises à jour … D'autres solutions plus élaborées peuvent être utilisées, mais ne seront pas traitées dans le cadre de ce cours. Une autre possibilité pour gérer les interblocages consiste à employer des **mises hors délais** (*timeouts*).

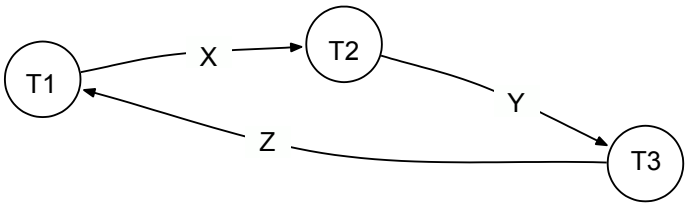
Cette méthode est pratique en raison du peu d'activité qu'elle implique sur le système. Si une transaction attend plus longtemps que la durée spécifiée pour la mise hors délai, le système suppose qu'elle est bloquée et il tue cette transactions, qu'il y ait ou non interblocage.

La détection des interblocages
Pour détecter les interblocages, on construit le graphe d'attente des ressources :



X, Y,… représentent des éléments des tables de la base. T1, T2 …. identifient des transactions. Un arc orienté étiqueté X ou Y entre Ti et Tj signifie que la transaction Ti attend la libération de X ou Y. L'interblocage se traduit par un circuit dans le graphe d'attente des ressources, comme dans la figure ci-dessus.

RAPPEL: l'interblocage peut mettre en jeu n (n≥2) transactions.



2.3.5 Niveaux d'isolation

SQL définit quatre niveaux d'isolation qui permettent de choisir quelles sont les types d'interactions autorisées pour les transactions qui s'exécutent simultanément (concurrentes). Ainsi, dans une transaction, on peut utiliser l'instruction suivante:

```
SET TRANSACTION ISOLATION LEVEL X
où X = SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
```

Transactions de niveau *serializable*

Si on reprend l'exemple de la section 4.1.5, et si on suppose que Paul = (max)(min) et Jo = (del)(ins) sont des transactions, et si Paul utilise le niveau d'isolation `SERIALIZABLE`, alors il verra la base dans l'état soit avant les modifications de Jo, soit après, mais pas entre les deux opérations de la transaction de Jo.

Le niveau d'isolation est un choix qui affecte uniquement la vue de la base de la transaction que le spécifie. Si on reprend notre exemple (cf section 4.1.5), si Jo choisit le niveau `SERIALIZABLE`, mais pas Paul, alors Paul peut éventuellement ne voir aucun prix si la requête est exécutée entre (del) et (ins).

Transactions de niveau *repeatable read*

Seules les données validées peuvent être lues, ce qui implique que les lectures répétées dans une même transaction retourne les mêmes valeurs (pas de phénomène de lectures non reproductibles). Cependant, une transaction pourra voir apparaître au cours des lectures répétées de nouveaux n-uplets (ou des n-uplets qui ont disparu).

Sur notre exemple, si Paul a choisi le niveau `REPEATABLE READ`, et que l'ordre d'exécution est (max)(del)(ins)(min), alors (max) verra les prix 2,50 et 3,00, et (min) verra 3,50, mais également 2,50 et 3,00 car ils ont été vus par la lecture précédente de (max) dans la transaction.

Transactions de niveau *read committed*

Seules les données validées peuvent être lues, mais des lectures successives peuvent renvoyer des valeurs (validées) différentes. Cela permet de voir les modifications apportées par d'autres transactions.

Sur notre exemple, si Paul a choisi le niveau `READ COMMITTED`, alors il peut voir les valeurs validées, mais pas nécessairement les mêmes à chaque requête. Ainsi, avec `READ COMMITTED`, l'entrelacement (max)(del)(ins)(min) est autorisé, aussi longtemps que Jo valide (mais Paul peut voir `MAX < MIN`).

Transactions de niveau *read uncommitted*

Une transaction qui s'exécute en niveau d'isolation `READ UNCOMMITTED` peut voir des données dans la base qui ne sont pas validées par une autre transaction (et qui ne le seront peut-être jamais).

Sur notre exemple, si Paul a choisi le niveau `READ UNCOMMITTED`, il pourra voir le prix 3,50, même si Jo abandonne la transaction.

2.3.6 Verrouillage et niveaux de granularité

Les performances du contrôle de la concurrence est affectée par la taille des éléments à verrouiller. Par exemple, dans une base de données, on a la valeur d'un attribut d'un n-uplet, un n-uplet, une table, ou toute la base. Selon que l'on pose un verrou sur un élément particulier, on peut bloquer ou non plusieurs transactions. En particulier, il faut remarquer que plus la taille des éléments à verrouiller est importante, plus le degré de concurrence autorisé est faible. Par exemple, si une transaction veut bloquer toute une table (et si elle ne souhaite accéder qu'à un n-uplet particulier), toutes les autres transactions seront mises en attente alors qu'il n'est peut être pas nécessaire de bloquer toute la table. Inversement, si on verrouille des éléments de taille plus petite, il peut y avoir beaucoup d'éléments à verrouiller, et donc le gestionnaire de verrouillage devra gérer un plus grand nombre de verrous (beaucoup d'opérations de verrouillage/déverrouillage). La nature des verrous à poser dépend donc de la nature des transactions. Par exemple, si une transaction accède à un petit nombre d'enregistrements, il est préférable que la granularité soit l'enregistrement. Par contre, si une transaction accède à de nombreux enregistrements d'une même table, alors la granularité qui semble la plus appropriée semble être la table.

2.4 Le verrouillage sous Oracle

2.4.1 Principe

Oracle implante un verrouillage au niveau de la ligne, et utilise un mécanisme appelé *Read consistency* (lecture cohérente). Grâce à ce mécanisme, les mises à jour et les lectures ne se bloquent pas mutuellement. Toutes les données retournées par un `SELECT` proviennent d'un seul point dans le temps. On ne voit que les données validées au moment du `SELECT`. Pour les données non validées, on voit les anciennes valeurs (grâce au *Rollback segment*). Les données modifiées par une autre transaction après le début du `SELECT` ne sont pas vues.

Il existe une concurrence d'accès en mise à jour: la première transaction qui accède à une donnée positionne un verrou sur les données accédées (n-uplets ou tables), les autres transactions souhaitant accéder les mêmes données seront mises en attente.

Ce mécanisme ne prend pas en compte les lectures non reproductibles. On peut, pour résoudre ce problème, utiliser une transaction spéciale appelée une transaction à lecture seulement (*Read Only Transaction*). Cette transaction conserve, lorsqu'elle démarre, l'état de la base et travaille ensuite uniquement sur cet état, jusqu'à la fin de la transaction. Une telle transaction ne doit comporter que des lectures de la base, et les modifications des données de la base sont interdites. La syntaxe est:

```
SET TRANSACTION READ ONLY
instruction 1
…
instruction n
COMMIT
```

2.4.2 Niveaux d'isolation

En fixant le niveau d'isolation des transactions, on définit le mode de fonctionnement en parallèle des différentes transactions qui s'exécutent au même moment. La commande suivante indique le niveau d'isolation de la transaction qui vient de commencer :

```
SET TRANSACTION ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED}
```

Oracle permet deux niveaux d'isolation des transactions les unes par rapport aux autres :

- `READ COMMITTED` : c'est le mode par défaut des transactions d'Oracle. Il empêche les principaux problèmes de concurrence mais pas les lectures non reproductibles. Tout d'abord, les modifications effectuées par une transaction ne sont connues des autres transactions que lorsque la transaction a été confirmée (`COMMIT`). Oracle gère automatiquement les accès concurrents de plusieurs transactions sur les mêmes n-uplets de table. Si une transaction est en train de modifier des n-uplets d'une table, les autres transactions peuvent lire les données telles qu'elles étaient avant ces dernières modifications, mais les autres transactions sont bloquées automatiquement par Oracle si elles souhaitent modifier ces mêmes lignes.
- `SERIALIZABLE` : si une transaction `SERIALIZABLE` essaie de modifier une donnée qui pourrait avoir été modifiée par une autre transaction non validée au début de la transaction `SERIALIZABLE`, alors l'ordre de modification échoue (résolution du problème de lecture non reproductibles). Ce mode est coûteux puisqu'il limite le fonctionnement en parallèle des transactions (les transactions s'exécutent concurremment comme si elles s'exécutaient les unes après les autres).

2.4.3 Différents niveau de verrouillage

Verrouillage de niveau n-uplet

Par défaut, Oracle met en œuvre un verrouillage de niveau n-uplet, c'est-à-dire une stratégie de verrouillage au niveau des n-uplets, chaque n-uplet d'une table pouvant être verrouillé individuellement. Les n-uplets verrouillés peuvent alors être mis à jour uniquement par la transaction qui les a verrouillés. Tous les autres n-uplets de la table peuvent être mis à jour par d'autres transactions. De plus, les autres transactions gardent également la possibilité de lire les n-uplets, y compris ceux qui sont mis à jour. Pour ces derniers, les transactions ont accès à l'ancienne valeur (grâce au *rollback segment*) jusqu'à ce que la transaction soit validée (lecture cohérente).

Lorsqu'une transaction pose un verrou sur un n-uplet, Oracle effectue les opérations suivantes:

- Premièrement, un verrou DML (*data manipulation language*) est posé. Ce verrou évite que d'autres transactions puissent mettre à jour (ou poser un verrou) sur le n-uplet. Ce verrou ne sera relâché que lorsque la transaction qui a posé le verrou aura été validée ou annulée,
- Deuxièmement, un verrou DDL (*data dictionary language*) est posé sur la table afin d'éviter les modifications structurelles de la table (suppression de la table, retrait ou ajout d'un attribut,…). Ce verrou ne sera relâché que lorsque la transaction qui a posé le verrou aura été validée ou annulée.

Verrouillage de niveau Table

Avec le verrouillage au niveau table, la table entière est verrouillée. Dès qu'une transaction a verrouillé une table, seule cette transaction peut mettre à jour (ou verrouiller) n'importe quel n-uplet de la table. Aucun des n-uplets de la table ne peut être mis à jour par une autre transaction. Cependant, les autres transactions peuvent malgré tout lire n'importe quel n-uplet, y compris ceux qui sont mis à jour.

Relâcher les verrous

Souvent, les utilisateurs pensent être seuls au monde. C'est précisément ce genre de comportement qui crée les problèmes de blocage. Souvenez-vous que poser un verrou (sur un n-uplet ou sur une table) n'est pas une opération anodine, et que cela peut bloquer beaucoup d'autres transactions. C'est pourquoi il faut veiller à verrouiller de manière approprié, c'est-à-dire qu'il faut protéger uniquement la partie de votre transaction qui est critique (mise à jour, calcul d'une moyenne, somme,…), et qu'il faut également relâcher le verrou. Il faut donc être conscient que l'on relâche un verrou en validant (`COMMIT`) ou en annulant (`ROLLBACK`) la transaction.

2.4.4 Les différents modes de verrouillage

Oracle propose deux modes de verrouillage:

- le mode exclusif (X) qui empêche les ressources verrouillées d'être partagées. C'est le mode privilé- gié pour mettre à jour des données. La transaction qui pose un verrou exclusif la première est la seule transaction qui peut modifier la table, et ce jusqu'à ce que le verrou soit relâché.
- le mode partagé (S) permet à une ressource d'être partagée. Plusieurs transactions peuvent avoir un verrou partagé et peuvent ainsi éviter qu'une autre transaction obtienne un verrou exclusif.

Oracle gère les types de verrous suivants: **row share** (RS), **row exclusive** (RX), **share** (S), **share row exclusive** (SRX), et **exclusive** (X).

Verrous exclusifs

SQL	**Verrou**
SELECT … FROM table…	pas de verrou
INSERT INTO table…	RX
UPDATE table…	RX
DELETE FROM table…	RX
LOCK TABLE table IN ROW EXCLUSIVE MODE	RX
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE	SRX
LOCK TABLE table IN EXCLUSIVE MODE	X

Verrous partagés

SQL	**Verrou**
SELECT … FROM table FOR UPDATE OF…	RS
LOCK TABLE table IN ROW SHARE MODE	RS
LOCK TABLE table IN SHARE MODE	S
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE	SRX

2.5 JDBC et les transactions

2.5.1 JDBC = Java DataBase Connectivity

JDBC est une API d'accès aux systèmes de gestion de base de données relationnelles qui permet d'exécuter des requêtes SQL au sein d'un programme Java et de récupérer les résultats, ce qui représente une alternative aux solutions propriétaires. C'est de plus une tentative de standardiser l'accès aux bases de données car l'API est indépendante du SGBD choisi, pourvu que le pilote JDBC existe pour ce SGBD, et qu'il implémente les classes et interfaces de l'API JDBC.

Pour effectuer un traitement avec une base de données, il faut :

1. charger un pilote en mémoire,
2. établir une connexion avec la base de données,
3. récupérer les informations relatives à la connexion,
4. exécuter des requêtes SQL et/ou des procédures stockées,
5. récupérer les informations renvoyées par la base de données (si nécessaire),
6. fermer la connexion.

Pour plus de détails, consulter le polycopié sur JDBC.

2.5.2 Gestion des transactions

L'instance de l'objet **Connection** fournit les méthodes pour :

Activer/désactiver la validation automatique de chaque requête passée (activé par défaut) :
public void setAutoCommit(boolean autoCommit) throws SQLException;
Valider manuellement la/les requête(s) passée(s) :
public void commit() throws SQLException;
Annuler les dernières modifications faites par la/les requête(s) passée(s) :
public void roolback() throws SQLException;

Exemple :

```
String sql1 = "UPDATE vendeur SET sal = 1215 WHERE sal < 1215";
String sql2 = "DELETE FROM vendeur WHERE sal < 1215";
try {
       connect.setAutoCommit(false);
       Statement state = connect.createStatement();
       state.executeUpdate(sql1);
       Statement state2 = connect.createStatement();
       state2.executeUpdate(sql2);
       connect.commit();
       ...
} catch (Exception e) {
       try{
              connect.rollback();
       } catch (SQLExection e) {
              ...
       }
       ...
}
```

JDBC supporte le mode transactionnel qui consiste à valider tout ou une partie d'un ensemble d'instructions. Nous avons déjà décrit à la section « Interface Connection » les méthodes qui permettent à un programme Java de coder des transactions (setAutoCommit, commit et rollback).

Par défaut, chaque instruction SQL est validée (on parle d'***autocommit***). Lorsque ce mode est désactivé, il faut gérer manuellement les transactions avec commit ou rollback.

Quand le mode ***autocommit*** est désactivé :

- La déconnexion d'un objet Connection (par la méthode close) valide implicitement la transaction (même si commit n'a pas été invoqué avant la déconnexion).
- Chaque instruction (CREATE, ALTER, DROP) valide implicitement la transaction.