Synthèse Bases de données

Jean-Philippe Collette

28 janvier 2012

Table des matières

Chapitre 1

Introduction

Une base de données est un ensemble de données conservées à long terme sur un ordinateur. Elle a comme caractéristiques

- une grande quantité et persistance ;
- la possibilité d'être interrogée et modifiée aisément ;
- la gestion sûre d'accès fréquents et multiples (transactions).

Un système de gestion de bases de données (SGBD ou DBMS) est un programme générique qui permet de définir, de mettre en oeuvre et d'exploiter une base de données.

Pour réaliser un système générique de base de données, il faut un cadre (ou modèle) permettant de définir le schéma des données à traiter. On distingue 3 composantes :

- le modèle (cadre de définition) : concepts utilisés pour structurer et définir les données ;
- le schéma (type, plan) : organisation de la base de données ; description de l'organisation des données et de leur type. Ne varie pas au cours de l'utilisation ;
- l'instance (extension) : contenu réel de la base de données à un moment fixé.

1.1 Modèle entité-relation

C'est un cadre de définition général du type de contenu potentiel d'une base de données. Il y a deux types génériques de base :

- les ensembles d'entités : ensembles d'objets de même structure (pour lesquels on enregistre les mêmes informations) ;
- les relations (ou associations) entre ces ensembles.

1.1.1 Les entités

Objet au sujet duquel on conserve de l'information dans la base de donnée. Une entité doit pouvoir être distinguée d'une autre entité.

Les entités sont regroupées en ensembles d'entités semblables, de même type, pour lesquelles ont veut conserver la même information (ex : employés d'une firme).

Le choix de l'ensemble revient au concepteur, et a pour conséquence que les informations conservées dans la base de données au sujet d'entités d'un même ensemble doivent avoir la même forme.

1.1.2 Les relations

Une relation entre un ensemble d'entités $E_1, \ldots, E_k$ est un sous-ensemble du produit cartésien $E_1 \times E_k$. C'est donc un ensemble de k-tuples $(e_1, \ldots, e_k)$ tels que $e_1 \in E_1, \ldots, e_k \in E_k$.

k est le degré de la relation ; si $k = 2$ la relation est binaire, si $k = 3$ la relation est ternaire.

1.1.3 Description et schéma des ensembles d'entités

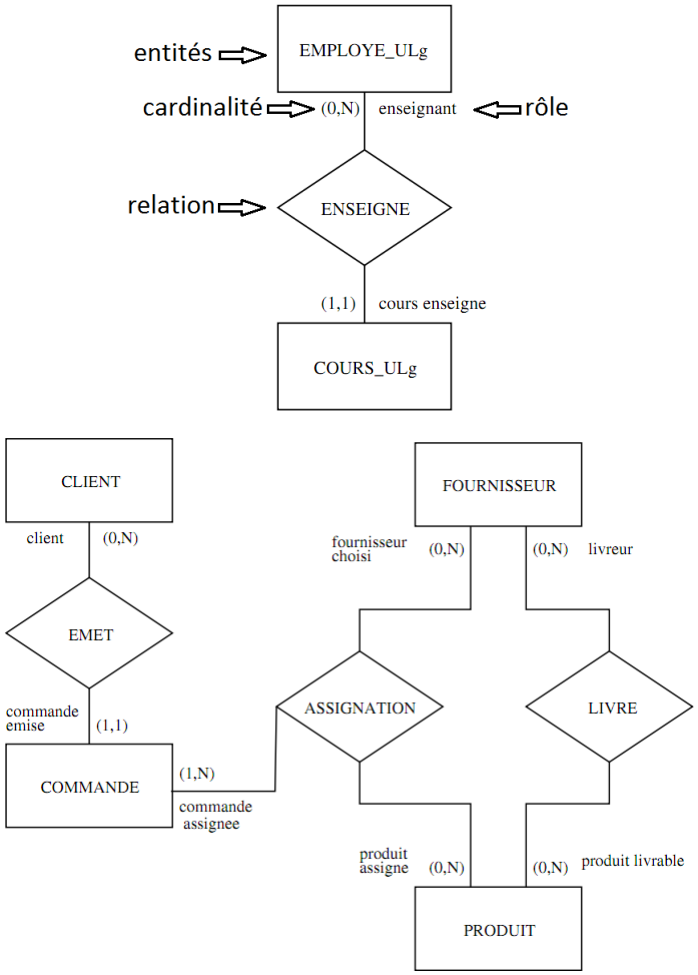
Un ensemble d'entités est décrit par un nom et un type (la liste de ses attributs) (ex : "employés de l'Ulg" que l'on appelle employé_ulg et "cours de l'Ulg" que l'on appelle cours_ulg).

Une relation est décrite par un nom et une liste ordonnée de noms d'ensembles d'entités (son type) (ex : enseigne : employé_ulg, cours_ulg).

Le rôle d'un ensemble d'entités est sa participation à une relation (par exemple dans la relation enseigne, le rôle de employé_ulg est "enseignant", celui de cours_ulg est "cours enseigné").

Pour chaque ensemble d'entités d'une relation, on précise un intervalle de tuples de la relation chaque entité peut apparaître : un min (généralement 0 ou 1) et un max (généralement 1 ou N/∞).

Pour un rôle, si min $>$ 0, la participation de l'ensemble d'entités est dite totale. Si $min = 0$, la participation est partielle.



1.1.4 Attributs

Les attributs sont les informations conservées au sujet d'entités d'un ensemble. Toutes les entités d'un ensemble ont les mêmes attributs, qui chacun ont un nom unique (dans le contexte de cet ensemble d'entités) et prend des valeurs dans un domaine spécifique (type de l'attribut).

Un attribut peut être

- simple (atomique) ou composite (décomposable en plusieurs attributs plus simples) (ex : adresse = rue, boîte, ville, pays, ...)
- obligatoire ou facultatif
- à valeur unique ou à plusieurs valeurs
- enregistré ou dérivé (d'un autre attribut)

1.1.5 Notion de clé

Une clé d'un ensemble d'entités est constituée d'attributs de l'ensemble d'entités et/ou de rôles joués par d'autres ensembles d'entités dans leur relation avec cet ensemble d'entités (relations identifiantes).

Deux entités distinctes ne peuvent avoir la même clé.

Un ensemble d'entités faibles est un ensemble qui ne dispose pas d'attributs constituant une clé. Ces entités se distinguent par

- certains de ses attributs et
- le fait qu'elles sont en relation avec d'autres ensembles d'entités

1.1.6 Attributs et clés de relations

Les relations peuvent avoir des attributs.

Une clé d'une relation identifie de façon unique un tuple parmi tous ceux de la relation ; elle est constituée de rôles joués par des ensembles d'entités dans cette relation et/ou d'attributs de la relation.

1.1.7 Contraintes d'intégrité

Ce sont les contraintes que doivent satisfaire les données d'une base de donnée. On compte comme contrainte :

- les contraintes de cardinalité (min, max)
- les clés
- les contraintes sur les attributs (lient les valeurs d'un attribut)
- les contraintes référentielles (on ne fait référence qu'à une entité existante)
- etc

1.1.8 Relation IS-A

Définition d'une hiérarchie entre les ensembles d'entités, qui correspond à une structure d'inclusion.

1.1.9 Spécialisation et généralisation

Un ensemble d'entités inclus s'appelle sous-classe (ou ensemble d'entités spécifiques). Un ensemble d'entités qui en comprend d'autres est une super-classe (ou ensemble d'entités génériques).

Une classe peut être divisée selon plusieurs critères indépendants, avec pour chacun plusieurs spécialisations (taxonomies) différentes.

Une classe peut être la sous-classe de plusieurs sous-classes.

Deux catégories de contraintes indépendantes :

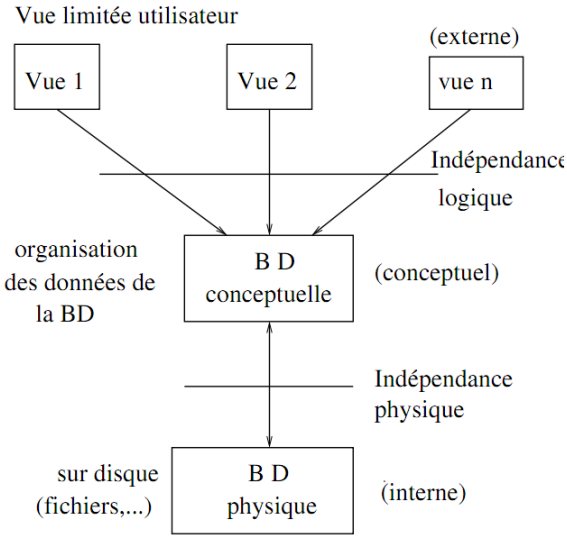
- quand l'union des sous-classes donne la super-classe, les sous-classes constituent une couverture de la super-classe. Dans ce cas, la cardinalité minimum du rôle de la super-classe dans la relation IS-A est 1 s'il y a couverture, 0 sinon.
- les sous-classes d'une super-classe peuvent être disjointes ou non. Dans ce cas, la cardinalité maximum du rôle de la super-classe dans la relation IS-A est 1 si les sous-classes sont disjointes, N sinon.

L'inclusion d'ensemble d'entités implique l'héritage des propriétés (attributs et rôles). Si un ensemble est sous-classe de deux classes différentes, c'est de l'héritage multiple, et on n'a plus une hiérarchie d'inclusion (arbre) mais un graphe dirigé acyclique (DAG).

1.2 SGBD

Un SGBD doit permettre de définir, mettre en oeuvre et exploiter une base de donnée à un niveau suffisamment abstrait ; plus précisément,

- le support d'un modèle de données
- le support de langages de haut niveau pour définir la structure des données, l'accès et la manipulation des données
- la gestion des transactions : parallélisme des opérations sur la base de données
- le contrôle d'accès : restrictions d'accès et vérification de la validité des données
- la capacité de récupération en cas de panne



1.2.1 Indépendance entre niveaux

Une indépendance existe si une modification à un niveau ne nécessite pas une modification du niveau supérieur. Il y a deux types d'indépendance :

- physique : entre niveaux physique et conceptuel
- logique : entre niveaux conceptuel et vue

Le modèle entité-relation sert de référence dans la conception de schémas de bases de données, toutefois il n'est pas implémenté dans les SGBD : le modèle relationnel est plus simple et plus facile à implémenter.

Chapitre 2

Modèle relationnel

On n'utilise qu'un seul type de structure pour représenter les données : la relation. On a ainsi des relations entre des ensembles de valeurs simples, plutôt qu'entre ensembles d'entités.

Intuitivement, on peut représenter le modèle comme une table, où chaque ligne est un tuple et représente une relation entre les valeurs se trouvant dans chaque colonne. Le nom des différentes colonnes sont les attributs de la relation.

2.1 Définitions

Soit un certain nombre d'identificateurs que l'on appelle attributs.

- le schéma d'une relation est un ensemble fini d'attributs (ex : $\{A_1, A_2\}$, $\{A_1, A_3\}$).
- le domaine d'un attribut est l'ensemble de ses valeurs possibles et est noté $dom(A_i) = D_i$. En général, les valeurs sont atomiques (relationnel, 1ère forme normale)
- le domaine d'un schéma de relation est l'union des domaines de ses attributs et est noté $dom(R) = dom(A_1) \cup dom(A_2) \cup dom(A_3)$.
- pour un schéma de relation R, un tuple est une fonction $t : R \to dom(R)$ telle que $\forall A \in R : t(A) \in dom(A)$
- pour un schéma de relation, une relation est un ensemble fini de tuples, donc un type n'apparaît qu'une seule fois dans une relation.
- un schéma de base de données est un ensemble fini de schémas de relations
- une base de donnée est un ensemble fini de relations.

On peut se passer d'attributs et définir une relation comme un sous-ensemble du produit cartésien de domaines pris dans un ordre donné. Il y a des inconvénients :

- l'ordre des composantes des tuples devient important, puisque la seule façon de les distinguer est leur position
- l'ordre est aussi important dans la description du schéma

2.2 Clés

Pour un schéma de relation, une superclé est un ensemble d'attributs qui identifie de manière unique un type de la relation ; il ne peut y avoir dans la relation deux tuples distincts qui ont les mêmes valeurs pour la superclé.

Une clé est une superclé minimale, c'est-à-dire une superclé dont on ne peut enlever aucun attribut sans lui faire perdre son status de superclé.

2.3 Modèle entité-relation vers modèle relationnel

2.3.1 Entités non faibles

Pour un ensemble d'entités E non faibles peut être représenté par la relation dont les attributs sont les attributs simples de E. Les attributs composites sont remplacés par leurs composantes.

2.3.2 Entités faibles

Pour un ensemble d'entités E faibles, on peut le représenter par la relation T dont les attributs sont
- les attributs de l'ensemble d'entités E, plus
- les attributs de la clé de chacun des ensembles d'entités participant aux relations identifiantes de E

La clé de T est la clé de E ; le rôle d'un ensemble d'entités est représenté par la clé de cet ensemble d'éléments.

2.3.3 Relation entre plusieurs ensembles

Une relation R entre les ensembles $E_1, \dots E_k$ est représentée par une relation T dont les attributs sont
- les attributs de R, plus
- les attributs de la clé de chacun des ensembles participant à la relation.

La clé de T est la clé de R.

2.3.4 Rôle(1,1)

Cas particulier : ensemble d'entités dont le rôle est borné par (1, 1). Soient E_1 et E_2 représentés par T_1 et T_2 et une relation R. Au lieu de représenter R, on peut ajouter à T_1
- les attributs de la clé de T_2
- les attributs de la relation R.

2.3.5 Attribut multivalué

Pour un attribut multivalué A d'un ensemble d'entités E (représenté par T) peut être représenté par une nouvelle relation T_A dont les attributs sont
- les attributs de la clé de la relation T correspondant à E
- un attribut correspondant à A ; si A est composite, on prend ses composantes

2.3.6 Spécialisation/généralisation

Soit un ensemble E spécialisé (ISA) par E_i.
E est représenté par une relation T_E dont les attributs de E. La clé de T_E est la clé de E.
E_i est représenté par une relation T_{E_i} dont les attributs sont
- les attributs de E_i plus
- les attributs de la clé de T_E

La clé de T_{E_i} est la clé de E.

2.4 Algèbre relationnelle

Il s'agit d'un ensemble d'opérations qui, à partir de relations, permettent de construire de nouvelles relations.

2.4.1 Opérations booléennes

Soient les relations r de schéma R et s de schéma S.

Union

Si $R = S$, l'union $r \cup s$ est la relation de schéma R (ou S) constituée de l'ensemble des tuples qui appartiennent à r ou à s :

$$r \cup s = \{t \in r\} \cup \{t \in s\}$$

Différence

Si $R = S$, la différente $r - s$ est la relation de schéma R (ou S constituée des tuples appartenant à r mais pas à s.

$$r - s = \{t \in r\} - \{t \in s\}$$

Intersection

$$r \cap s = r - (r - s) = \{t : t \in r \text{ et } t \in s\}$$

2.4.2 Projection

Soit une relation r de schéma R et $S \subseteq R$. La projection de r sur S est la relation de schéma S obtenue à partir de r en éliminant les attributs de R qui ne sont pas dans S :

$$\pi_S(r) = \{t(S)|t \in r\} = \{t(S)|\exists t' \in r : t'(S) = t(S)\}$$

2.4.3 Sélection

Soit un schéma de relation $R = \{A_1, \ldots, A_k\}$. Une condition de type R est une combinaison booléenne de formules atomiques $A_i \theta A_j$ ou $a \theta A_i$ ou $A_i \theta a$, où θ est une relation sur le domaine de A_i (A_j) et a une constante de ce domaine.

Pour une relation r de schéma R et une fonction F de type R, la sélection $\sigma_F(r)$ est la relation de schéma R constituée de l'ensemble des tuples de r qui satisfont la condition F :

$$\sigma_F(r) = \{t \in r|F[A_i \leftarrow t(A_i)] = true\}$$

2.4.4 Produit cartésien

Soit r de schéma R et s de schéma R. Si $R \cap S = \emptyset$, le produit cartésien $r \times s$ est la relation de schéma $R \cup S$ obtenue en combinant les tuples de r et de s de toutes les manières possibles :

$$r \times s = \{t|(\exists t' \in r)(\exists t'' \in s)(t(R) = t' \wedge t(S) = t'')\}$$

2.4.5 Jointure/joint

Si $R \cap S \neq \emptyset$, $r \bowtie s$ est une relation de schéma $R \cup S$:

$$r \bowtie s = \{t|(\exists t' \in r)(\exists t'' \in s)(t(R) = t' \wedge t(S) = t'')\}$$

Chaque tuple de $r \bowtie s$ correspond à un tuple t' de r et un tuple t'' de s qui ont des valeurs identiques pour les attributs communs à R et à S (dans $R \cap S$).

2.4.6 Joint conditionnel

Soient les relations $r(R)$ et $s(S)$ et soient les attributs $A_R \in R$ et $A_S \in S$. On définit le joint conditionnel par

$$r \underset{A_R \theta A_S}{\bowtie} s = \sigma_{A_R \theta A_S}(r \times s)$$

2.4.7 Quotient

Soient $r(R)$ et $s(S)$, avec $S \subseteq R$. On veut trouver les tuples t sur $R - S$ tels que pour chaque tuple $t_s \in s$ on a $t_r \in r$.

$$r \div s = \{t | (\forall t_s \in s)(\exists t_r \in r)(t_s(S) = t_s \wedge t_r(R - S) = t)\}$$

$r \div s$ est le sous-ensemble r' maximum de $\pi_{R-S}(s)$ tel que $(r' \times s) \subseteq r$.
On peut exprimer cet opérateur ainsi :

$$r \div s = \pi_{R-S}(r) - \pi_{R-S}((\pi_{R-S}(r) \times s) - r)$$

2.4.8 Relations constantes

On va admettre les relations constantes, dont les tuples ne dépendent pas de l'état de la base de données.

2.4.9 Changement de nom

Peut être utile lorsqu'on veut appliquer des opérations lorsque les noms des attributs ne correspondent pas, où quand il y a des attributs identiques et qu'on ne veut pas les voir dans le résultat.

$$\delta_{A \leftarrow B}(r) = \{t | (\exists t' \in r)(t(B) = t'(A) \wedge (\forall A_i \neq A)(t(A_i) = t'(A_i)))\}$$

Chapitre 3

Dépendances et normalisation des relations

Une série de problèmes peuvent apparaître lorsqu'on utilise un schéma de relation :
- de la redondance
- des problèmes d'inconsistance
- des anomalies d'insertion
- des anomalies de suppression

La solution est de décomposer le schéma en plusieurs sous-schémas.

3.1 Les dépendances fonctionnelles

La redondance vient du fait que l'information a une structure particulière. Par exemple *FOURNISSEURS(NOM_F, ADRESSE_F, COMB, PRIX)* : si on connaît *NOM_F*, on connaît *ADRESSE_F*. Un fournisseur n'a qu'une adresse ; on ne peut avoir deux tuples qui ont la même valeurs *NOM_F* et des valeurs différentes de *ADRESSE_F*.

Le problème est de choisir les schémas de relation en accord avec cette structure, et pour cela on a besoin d'une représentation de la structure de l'information.

3.1.1 Définition

Soit un schéma de relation $R(A_1, \ldots, A_n)$ et soient $X, Y \subseteq \{A_1, \ldots, A_n\}$.

Les ensembles X et Y définissent une dépendance fonctionnelle, notée $X \to Y$.

Une relation r de schéma R satisfait la dépendance fonctionnelle $X \to Y$ si, pour tout tuples $t_1, t_2 \in r$, si $t_1[X] = t_2[X]$, alors $t_1[Y] = t_2[Y]$.

On associe une dépendance au schéma d'une relation ; elle indique une caractéristiques des données que l'on veut modéliser (tout comme le schéma). On suppose dès lors que toute relation se trouvant dans la base de données satisfait les dépendances. Le système pourrait refuser toute modification qui violerait une dépendance.

Pour des ensembles d'attributs X, Y, on définit l'union comme $XY (= X \cup Y)$.

3.1.2 L'implication des dépendances

Soient F et G deux ensembles de dépendances.
- une dépendance $X \to Y$ est impliquée logique par F (noté $F \models X \to Y$) si toute relation qui satisfait F satisfait aussi $X \to Y$
- F et G sont (logiquement) équivalents (noté $F \equiv G$) si toute dépendance de G est impliquée logiquement par F et vice-versa
- la fermeture de F est l'ensemble F^+ des dépendances logiquement impliquées par F, c'est à-à-dire

$$F^+ = \{X \to Y | F \models X \to Y\}$$

3.1.3 Propriétés des dépendances fonctionnelles

Les dépendances satisfont les règles suivantes :
- réflexivité : si $Y \subseteq X$, alors $X \to Y$
- augmentation si $X \to Y$ et $Z \subseteq U$, alors $XZ \to YZ$
- transitivité : si $X \to Y$ et $Y \to Z$, alors $X \to Z$
- union : si $X \to Y$ et $X \to Z$, alors $X \to YZ$
- décomposition : si $X \to Y$, alors $X \to Z$ si $Z \subseteq Y$

La conséquence des deux dernières règles est qu'une dépendance $X \to A_1 A_2 \dots A_n$ est équivalente à l'ensemble des dépendances $X \to A_1$, $X \to A_2$, …, $X \to A_n$.

3.1.4 Fermeture d'un ensemble d'attributs

Soit F un ensemble de dépendances et $X \subseteq U$ un ensemble d'attributs pris parmi un ensemble d'attributs U. La fermeture X^+ à partir de X par rapport à l'ensemble de dépendances F est l'ensemble des $A \in U$ tels que $F \models X \to A$.

Si on peut calculer la fermeture d'un ensemble d'attributs, il est simple de déterminer si une dépendance $X \to Y$ est impliquée logiquement par un ensemble de dépendances F :

1. calculer X^+ par rapport à F
2. si $Y \subseteq X^+$, alors $F \models X \to Y$, sinon $F \not\models X \to Y$

Cela permet de calculer F^+ :

$$F^+ = \{X \to Y | F \models X \to Y\} = \{X \to Y | X \subseteq U \text{ et } Y \subseteq X^+\}$$

Le nombre de dépendances dans F^+ peut être très grand, exponentiel en fonction du nombre d'attributs.

Algorithme de calcul de fermeture

Soit F un ensemble de dépendances et $X \subseteq U$ un ensemble d'attributs. L'algorithme calcule une suite d'ensembles d'attributs $X^{(0)}, X^{(1)}, \dots$.

Données : X, U, F

1. $X^{(0)} = X$
2. $X^{(i+1)} = X^{(i)} \cup \{A | \exists Z : Y \to Z \in F \text{ et } A \in Z \text{ et } Y \subseteq X^{(i)}\}$
3. si $X^{(i+1)} = X^{(i)}$, l'algorithme s'arrête.

L'algorithme s'arrête toujours puisque $X^{(0)} \subseteq X^{(1)} \subseteq \dots \subseteq U$

Théorème L'algorithme donné calcule bien X^+, c'est-à-dire que, quand l'algorithme s'arrête, $X^{(i_f)} = \{A | F \models X \to A\}$.

Preuve : On va monter la double inclusion.

1. montrons que $X^{(i_f)} \subseteq X^+ = \{A | F \models X \to A\}$ par induction :
   (a) initialement, $X^{(0)} = X \subseteq X^+$
   (b) si $X^{(i)} \subseteq X^+$, alors $X^{(i+1)} \subseteq X^+$. Par la définition de X_{i+1}, si $F \models X \to A$ pour tout $A \in X^{(i)}$, alors on a $F \models X \to A$ pour tout $A \in X^{(i+1)}$
2. montrons que $X^+ \subseteq X^{(i_f)}$, c'est-à-dire $X^+ - X^{(i_f)} = \emptyset$.
   Pour ce faire, on va montrer que pour toute dépendance $X \to B$ telle que $B \notin X^{(i_f)}$, on a $F \not\models X \to B$ et donc $B \notin X^+$. Il faut donc prouver qu'il existe une relation qui satisfait les dépendances de F, mais qui ne satisfait pas $X \to B$, avec un exemple :

r :

$X^{(i_f)}$		$U - X^{(i_f)}$	
1	$1 \cdots 1$	1	$1 \cdots 1$
1	$1 \cdots 1$	0	$0 \cdots 0$

Cette relation ne satisfait pas $X \to B$ vu que $X \subseteq X^{(i_f)}$ et que $B \notin X^{(i_f)}$. On va montrer qu'elle ne satisfait pas F.

Si r ne satisfait pas F, il y aurait dans F une dépendance $V \to W$ telle que $V \subseteq X^{(i_f)}$ et $W \nsubseteq X^{(i_f)}$, par définition de R.

Mais dans ce cas, l'algorithme de calcul de X^+ ne se serait pas arrêté et aurait ajouté les attributs de W.

3.1.5 Couverture d'un ensemble de dépendances

Par définition, deux ensembles de dépendances F et G sont équivalents si pour tout $X \to Y \in F$ on a $G \models X \to Y$ et vice-versa, ou bien si $F^+ = G^+$. On cherche une description la plus succincte possible d'un ensemble de dépendances à l'équivalence près.

F est une couverture pour un ensemble de dépendances G si $F \equiv G$. Une couverture F de G est minimale si

1. F contient uniquement des dépendances du type $X \to A$
2. $\nexists (X \to A) \in F : F - \{X \to A\} \equiv F$
3. $\nexists (X \to A) \in F$ et $\nexists Z \subset X : (F - \{X \to A\}) \cup \{Z \to A\} \equiv F$

Théorème Tout ensemble de dépendances a une couverture minimale.

Preuve : On peut obtenir une couverture minimale en procédant ainsi :

1. toute dépendance $X \to Y$ avec $Y = \{A_1, \ldots, A_n\}$ est remplacée par les dépendances $X \to A_1$, …, $X \to A_n$
2. on élimine le plus de dépendances (redondantes) possible
3. on élimine le plus d'attributs possibles des membres de gauche

Les couvertures minimales ne seront en général pas uniques.

3.1.6 Notion de clé

La notion de clé peut se définir à partir des dépendances ; soit F un ensemble de dépendances :

- un ensemble d'attributs X est une clé d'une relation de schéma R par rapport à F si X est un ensemble minimum tel que $F \models X \to R$
- un ensemble d'attributs X est une super-clé d'une relation de schéma R par rapport à F si X est un ensemble (non nécessairement minimum) tel que $F \models X \to X$.

3.2 Normalisation

La présence de dépendances peut mener à une duplication de l'information, il faut éliminer celles qui sont néfastes.

Pour les dépendances fonctionnelles, il faut éviter d'avoir une dépendance non-triviale $Y \to A$ pour laquelle la partie de tuple $y_1, \ldots, y_k$ puisse apparaître plusieurs fois. Il faut définir des contraintes sur les schémas de relation pour éviter ces répétitions : ce sont les formes normales

3.2.1 Forme normale de Boyce-Codd (BCNF)

Un schéma de relation est en BCNF si pour toute dépendance non-triviale $X \to A$, alors X est une super-clé.

Le problème de la normalisation est de remplacer un schéma de base de données par un autre équivalent qui est en BCNF. Il faut ainsi qu'avec le nouveau schéma il soit toujours possible de reconstituer la base de données de départ.

Normalisation par décomposition

Un schéma r est remplacé par un ensemble de schémas R_i, où $R_i \subset R$. La relation correspondante r est remplacée par les projections $\pi_{R_i} r$.

Ce n'est possible que si les relations $\pi_{R_i} r$ contiennent la même information que r, ce qui n'est pas toujours le cas.

Décomposition sans perte

Soit un schéma R et $\rho = (R_1, \ldots, R_k)$ une décomposition de R telle que $R = R_1 \cap \cdots \cap R_k$.

- soit une relation r de schéma R. La décomposition ρ de R est sans perte pour r si

$$r = \pi_{R_1}(r) \bowtie \ldots \bowtie \pi_{R_k}(r) = m_\rho(r)$$

- soit F un ensemble de dépendances sur R. La décomposition ρ de R est sans perte par rapport à F si pour toute relation r de schéma R qui satisfait F, la décomposition est sans perte pour r.

Propriétés d'une décomposition

1. $$r \subseteq m_\rho(r)$$

r :

A	B	C
0	0	0
1	0	1

$\pi_{R_1}(r)$:

A	B
0	0
1	0

$\pi_{R_2}(r)$:

B	C
0	0
0	1

$m_\rho(r)$:

A	B	C
0	0	0
0	0	1
1	0	0
1	0	1

et $r \subset m_\rho(r)$

Preuve : soit un tuple quelconque t de r. On a $t[R_1] \in \pi_{R_1}(r), \ldots, t[R_k] \in \pi_{R_k}(r)$. Donc, par définition du joint, on a $t \in m_\rho(r)$

2. $$\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$$

Preuve : on va démontrer la double-inclusion :

- On a $\pi_{R_i}(m_\rho(r)) \subseteq \pi_{R_i}(r)$, car par la propriété 1 on a $r \subseteq m_\rho(r)$, d'où l'inclusion.
- On peut montrer que $\pi_{R_i}(r) \subseteq \pi_{R_i}(m_\rho(r))$
  On considère un tuple $t_i \in \pi_{R_i}(m_\rho(r))$. Il existe donc un tuple $t \in m_\rho(r)$ tel que $t_i = t[R_i]$, par définition de π_{R_i}.
  De plus, puisque $t \in m_\rho(r)$, il existe un tuple $t' \in r$ tel que $t'[R_i] = t[R_i]$, par définition de $m_\rho(r)$.
  Donc $t'[R_i] = t_i$, et par conséquent $t_i \in \pi_{R_i}(r)$

Du coup, il n'est pas possible de distinguer la relation r de la relation $m_\rho(r)$ à partie des projections. Autrement dit, si $r \neq m_\rho(r)$, il n'est pas possible de reconstituer r à partir de $\pi_{R_i}(r)$.

Critère de décomposition sans pertes

Soit $\rho = (R_1, R_2)$ une décomposition de R. Cette décomposition est sans perte par rapport à un ensemble de dépendances fonctionnelles F si

$$(R_1 \cap R_2) \rightarrow (R_1 - R_2) \in F^+$$

ou

$$(R_1 \cap R_2) \rightarrow (R_2 - R_1) \in F^+$$

Preuve : on va supposer que $(R_1 \cap R_2) \rightarrow (R_1 - R_2) \in F^+$ ou $(R_1 \cap R_2) \rightarrow (R_2 - R_1) \in F^+$ est vrai.

On montre que pour toute relation r de schéma R, si r satisfait F, alors $r = m_\rho(r)$.

Soit une relation r de schéma R qui satisfait F. On sait que $r \subseteq m_\rho(r)$, montrons que $m_\rho(r) \subseteq r$.

Soit un tuple $t \in m_\rho(r)$, on veut montrer que $t \in r$. Puisque $t \in m_\rho(r)$, il existe des tuples $t_1, t_2 \in r$ tels que $t_1[R_1] = t[R_1]$ et $t_2[R_2] = t[R_2]$.

Supposons que $(R_1 \cap R_2) \rightarrow (R_1 - R_2) \in F^+$. Puisque $t_2[R_2] = t[R_2]$, on a $t_2[R_1 \bigcap R_2] = t[R_1 \bigcup R_2]$.

Et puisque $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$, alors $t_2[R_1] = t[R_1]$, donc $t[R] = t_2[R]$, et puisque $t_2 \in r$, on a $t \in r$.

3.2.2 Algorithme de décomposition en BCNF

Soit un schéma R et un ensemble de dépendances fonctionnelles F. L'algorithme procède par décompositions successives pour obtenir une décomposition de R sans perte par rapport à F, mais sans garantie de conservation des dépendances.

- si R n'est pas en BCNF, soit une dépendance non triviale $X \rightarrow A$ de F^+, où X n'est pas une super-clé
- on décompose R en $R_1 = R - A$ et $R_2 = XA$ (sans perte vu le critère ; $R_1 \bigcap R_2 = X$ et $R_2 - R_1 = A$)
- on applique l'algorithme $R_1, \pi_{R_1}(F)$ et $\pi_{R_2}(F)$

Puisque les relations deviennent de plus en plus petites, la décomposition doit s'arrêter.

3.2.3 La troisième forme normale - 3FN

Il n'est parfois pas préférable de décomposer en BCNF, car cela peut casser des dépendances . Cela peut ne pas poser problème s'il est en troisième forme normale.

Exemple : R = (city, street, zip) et $F = \{citystreet \rightarrow zip, zip \rightarrow city\}$. Le schéma peut être décomposé en BCF : (street zip) et (city zip). Cependant, la dépendance $citystreet \rightarrow zip$ n'est pas conservée, et pour retrouver un code postal à partir d'une ville et d'une rue, il faudra effectuer une jointure.

On va distinguer deux types d'attributs :

- les attributs premiers, qui font partie d'une clé
- les attributs non premiers, qui ne font pas partie d'une clé.

Un schéma de relation est en 3FN si, pour toutes dépendances non-triviales $X \rightarrow A$ avec A non-premier, alors X est une super-clé.

La définition est semblable à BCNF, mais on ne tient pas compte des attributs non premiers. L'intérêt est qu'il existe des algorithme capable de décomposer en 3FN tout en gardant les dépendances.

3.2.4 1FN et 2FN

La première forme normale (1FN) précise que les attributs d'une relation ont une valeur atomique (c'est une hypothèse du modèle relationnel).

La deuxième forme normale (2FN) exclut les dépendances non-triviales $X \rightarrow A$, où A est non-premier et où X est un sous-ensemble propre d'une clé. C'est une contrainte moins forte que celle donnée par 3FN, et de fait par BCNF.

3.3 Les dépendances à valeurs multiples

Dans une relation R, on a une dépendance à valeurs multiples entre X et Y ($X, Y \subseteq R$) si les valeurs possibles de Y sont déterminées par X, indépendamment du contenu du reste de la relation. Cela s'écrit $X \twoheadrightarrow Y$, X multi-détermine Y.

Exemple

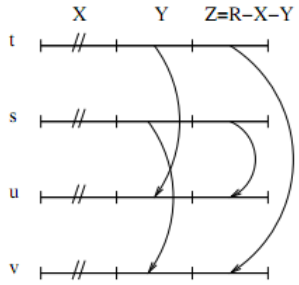
$COURS \twoheadrightarrow JOUR$

$COURS \twoheadrightarrow ETUDIANT$

COURS	JOUR	ETUDIANT
#1	Mardi	A. Dupont
#1	Jeudi	A. Dupont
#1	Mardi	B. Durand
#1	Jeudi	B. Durand

Plus formellement, une relation r de schéma R satisfait une dépendance à valeurs multiples $X \twoheadrightarrow Y$ si, pour toute paire de tuples $t, s \in r$ tels que $t[X] = s[X]$, il existe deux tuples $u, v \in r$ tels que

1. $u[X] = v[X] = t[X] = s[X]$
2. $u[Y] = t[Y]$ et $u[R - X - Y] = s[R - X - Y]$
3. $v[Y] = s[Y]$ et $v[R - X - Y] = t[R - X - Y]$



Exemple : on a $C \twoheadrightarrow HR$, $C \twoheadrightarrow SG$ et $HR \twoheadrightarrow SG$, mais pas $C \twoheadrightarrow H$ et $C \twoheadrightarrow R$.

C : course
T : teacher
H : hour
R : room
S : student
G : grade

$C \rightarrow T$ $\quad CS \rightarrow G$
$HR \rightarrow C$ $\quad HS \rightarrow R$
$HT \rightarrow R$

C	T	H	R	S	G
INFO009	ProfX	lun 14 :00	R3	A. Dupont	18
INFO009	ProfX	mer 14 :00	R7	A. Dupont	18
INFO009	ProfX	ven 10 :00	I21	A. Dupont	18
INFO009	ProfX	lun 14 :00	R3	B. Durand	14
INFO009	ProfX	mer 14 :00	R7	B. Durand	14
INFO009	ProfX	ven 10 :00	I21	B. Durand	14

Au niveau de dépendances multiples triviales, quelques exemples :

1. pour $R(A, B)$, on a toujours $A \twoheadrightarrow B$ (toujours satisfaite)
2. pour $R(A, B, C)$, on a $A \twoheadrightarrow B$ selon les cas.

A	B	C
a	b	d
a	c	e

non

A	B	C
a	b	d
a	c	e
a	b	e
a	c	d

oui

3. avec $R(A, B, C)$ et $A \rightarrow B$. La dépendance à valeurs multiples $A \twoheadrightarrow B$ est toujours satisfaite lorsque $A \rightarrow B$ est satisfait.
   En effet, soient $t, s \in R$ tels que $t[A] = s[A]$. Vu que $t[B] = s[B]$, il existe bien des tuples $u, v \in r$ tels que $u[A] = v[A] = t[A] = s[A]$, $u[B] = t[B]$, $u[C] = s[C]$, $v[B] = s[B]$ et $v[C] = t[C]$. Ces tuples sont respectivement s et t.

3.3.1 Propriétés des dépendances à valeurs multiples

- réflexivité : si $Y \subseteq X$, alors $X \twoheadrightarrow Y$
- complémentation : si $X \twoheadrightarrow Y$, alors $X \twoheadrightarrow (R - XY)$
- reproduction : si $X \rightarrow Y$, alors $X \twoheadrightarrow Y$
- union : si $X \twoheadrightarrow Y$ et $X \twoheadrightarrow Z$, alors $X \twoheadrightarrow XZ$
- intersection : si $X \twoheadrightarrow Y$ et $X \twoheadrightarrow Z$, alors $X \twoheadrightarrow Y \bigcap Z$
- différence : si $X \twoheadrightarrow Y$ et $X \twoheadrightarrow Z$, alors $X \twoheadrightarrow Y$
  Z

Les règles de décomposition et de transitivité ne sont pas valides pour les dépendances à valeurs multiples. Par exemple, pour le schéma CTHRSG, on a $C \twoheadrightarrow HR$ et $HR \twoheadrightarrow H$, mais pas $C \twoheadrightarrow H$.

De plus, comme pour les dépendances fonctionnelles, il est possible de déterminer si une DVM est une conséquence logique d'un ensemble de dépendances.

3.3.2 Décomposition sans perte

Pour toute relation r de schéma R, r satisfait la DVM $X \twoheadrightarrow Y$ si et seulement si la décomposition $\rho = (XY, X(R - XY))$ est sans perte pour r, c'est-à-dire $r = m_\rho(r)$

Preuve : soit r une relation quelconque de schéma R.

$\Leftarrow$ supposons que la décomposition est sans perte pour r, c'est-à-dire

$$r = \pi_{XY}(r) \bowtie \pi_{XZ}(r)$$

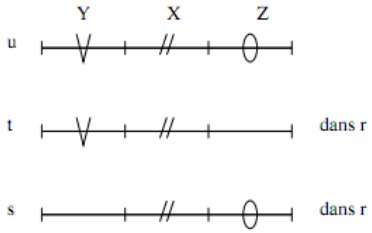
avec $Z = R - XY$. Considérons deux tuples $t, s \in r$ tels que $t[X] = s[X]$. On a $t[XY] \in \pi_{XY}(r)$ et $s[XZ] \in \pi_{XZ}(r)$.

Donc les tuples u, v tels que $u[X] = v[X] = t[X] = s[X]$, $u[Y] = t[Y]$, $u[Z] = s[Z]$, $v[Y] = s[Y]$ et $v[Z] = t[Z]$ appartiennent à $\pi_{XY}(r) \bowtie \pi_{XZ}(r) = r$.

$\Rightarrow$ supposons que r satisfait la dépendance $X \twoheadrightarrow Y$ et montrons que le joint est sans perte, c'est-à-dire que si $u \in \pi_{XY}(r) \bowtie \pi_{XZ}(r)$, alors $u \in r$.

Si $u \in \pi_{XY}(r) \bowtie \pi_{XZ}(r)$, alors $u \in r$, il existe $t, s \in r$ tels que $t[XY] = u[XY]$ et $s[XY] = y[XZ]$, ou encore $u[X] = t[X] = s[X]$, $u[Y] = t[Y]$, $u[Z] = s[Z]$.

Puisque r satisfait $X \twoheadrightarrow Y$, le tuple u doit être aussi dans r (par définition des DVM).



La propriété peut aussi s'écrire comme suit : soit R un schéma de relation et $\rho = (R_1, R_2)$ une décomposition de R.

Pour toute relation r de schéma R, r satisfait la DVM $(R_1 \bigcup R_2) \twoheadrightarrow (R_1 - R_2)$ (ou $(R_1 \bigcup R_2) \twoheadrightarrow (R_2 - R_1)$) par complémentation) si et seulement si la décomposition (R_1, R_2) de R est sans perte pour r, c'est-à-dire $r = m_\rho(r)$.

Il y a une équivalence entre la notion de dépendance à valeurs multiples et celle de décomposition sans perte.

Pour les dépendances fonctionnelles, la présence de la dépendance fonctionnelle permet la décomposition sans perte, mais une relation r peut être décomposable sans perte sans qu'elle ne satisfasse de décomposition fonctionnelle.

3.3.3 Critère de décomposition sans perte

Soit $\rho = (R_1, R_2)$ une décomposition de R. Cette dépendance est sans perte par rapport à un ensemble de dépendances (fonctionnelles et multiples) D si et seulement si

$$(R_1 \bigcup R_2) \twoheadrightarrow (R_1 - R_2) \in D^+$$

ou, par complémentation,

$$(R_1 \bigcup R_2) \twoheadrightarrow (R_2 - R_1) \in D^+$$

Preuve : c'est la conséquence directe de la propriété établissant l'équivalence entre la satisfaction d'une DVM par une relation et la décomposition sans perte pour cette relation.

3.3.4 Quatrième forme normale

Il peut être gênant, pour une DVM $X \twoheadrightarrow Y$, si l'association entre les valeurs de X et celles de Y est répétée, c'est-à-dire apparaît pour plusieurs valeurs des autres attributs $Z = R - XY$ de la relation. Cela peut se produire, sauf si

- $Z = \emptyset$, $X \twoheadrightarrow Y$ est triviale, ou
- $Y \subset X$, $X \twoheadrightarrow Y$ est triviale, ou
- $X \rightarrow R$, X est une super-clé.

On peut définir la quatrième forme normale : un schéma est en 4FN si, pour toute dépendance $X \twoheadrightarrow Y$,

- soit $Y \subseteq X$
- soit $XY = R$
- soit X est une super-clé de R.

La 4FN implique la BCNF. En effet,

- une dépendance $X \rightarrow Y$ est aussi une dépendance $X \twoheadrightarrow Y$
- si $X \subset X$, la dépendance $X \rightarrow Y$ est triviale
- si $XY = R$, alors $X \rightarrow R$ et X est une super-clé.

Pour obtenir la 4FN, on peut procéder par décomposition comme pour la BCNF.

3.3.5 Algorithme de décomposition en 4FN

Soit un schéma R et un ensemble de dépendances (fonctionnelles et à valeurs multiples) D.

L'algorithme procède par décomposition successives pour obtenir une décomposition de R sans perte par rapport à D, mais sans garnatie de conservation des dépendances.

- si R n'est pas en 4FN, soit une dépendance $X \twoheadrightarrow Y$ de D^+ telle que $Y \neq \emptyset$, $Y \not\subseteq X$, $XY \subset R$ et X n'est pas une super-clé
  On peut supposer que $X \bigcap Y = \emptyset$, puisque $X \twoheadrightarrow (Y - X)$ est impliqué par $X \twoheadrightarrow Y$
- on décompose R en $R_1 = XY$ et $R_2 = X(R - XY)$, ce qui est sans perte vu le critère
- on applique l'algorithme à R_1, $\pi_{R_1}(D)$ et R_2, $\pi_{R_2}(D)$.

Puisque les relations deviennent de plus en plus petites, la décomposition doit s'arrêter.

Exemple : Relation de schéma $CTHRSG$

C : *course*	R : *room*	$C \rightarrow T$	$CS \rightarrow G$
T : *teacher*	S : *student*	$HR \rightarrow C$	$HS \rightarrow R$
H : *hour*	G : *grade*	$HT \rightarrow R$	

$\boxed{C \twoheadrightarrow HR}$ est une dépendance à associer à ce schéma. Il y a une seule clé : HS

Toutes les MVD qui s'appliquent naturellement à ce schéma sont dérivables de l'ensemble de DF ci-dessus et de $C \twoheadrightarrow HR$.

Par exemple :

- De $C \twoheadrightarrow HR$ et $C \rightarrow T$, on peut dériver $C \twoheadrightarrow SG$:
  - de $C \rightarrow T$, on obtient $C \twoheadrightarrow T$, et ensuite $C \twoheadrightarrow HRT$ par union ;
  - la règle de complémentation donne alors $C \twoheadrightarrow SG$.
- De $HR \rightarrow CT$, on peut dériver $HR \twoheadrightarrow CT$.
- Par la règle de complémentation on obtient $HR \twoheadrightarrow SG$.

R_1 : $C\underline{HR}$ avec $C \twoheadrightarrow HR$ 4FN !

R_2 : $\underline{CT}SG$ avec $C \twoheadrightarrow T$

R_3 : $\underline{CT}$ avec $C \rightarrow T$ 4FN !

R_4 : $\underline{CS}G$ avec $CS \rightarrow G$ 4FN !

3.3.6 Autres types de dépendances

On a aussi les dépendances "joint" ; les DVN caractérisent les cas où une décomposition en 2 relations est sans perte. Il est parfois possible de décomposer une relation sans perte en trois sous-schémas sans qu'il y ait de décomposition en deux sous-schémas qui sont sans perte.

On généralise donc la notion de DVM en la notion de dépendance-joint.

Une relation r de schéma R satisfait la dépendance-joint $\star[R_1, \ldots, R_k]$ si $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \ldots \bowtie \pi_{R_k}(r)$

Cela conduit à la définition de 5FN, "project-join normal form".

3.3.7 La cinquième forme normale - 5FN

Une relation r de schéma R est en 5FN par rapport à un ensemble de dépendances fonctionnelles et joint F si, pour toute dépendance joint $\star[R_1, \ldots, R_k]$ impliquée par F,

- soit elle est triviale (vraie de toute relation de schéma R)
- soit chaque R_i est une super-clé de R

Chapitre 4

Langages d'interrogation

4.1 Multi-ensembles

Un multi-ensemble (multiset) est une collection d'éléments pour laquelle on tient compte de la multiplicité (le multi-ensemble $\{1, 1, 2, 3\}$ est différent de $\{1, 2, 3\}$).

Les systèmes de bases de données considèrent les relations comme des multi-ensembles, car

- travailler avec des multi-ensembles accélère certaines opérations, car il n'est pas nécessaire de rechercher les occurrences multiples de tuples (projection, union) ;
- pour certaines opérations (ex : calcul d'une moyenne), c'est nécessaire.

4.1.1 Algèbre relationnelle des multi-ensembles

Soient deux relations r et s de schéma R et S et un tuple t qui apparait m fois dans r et n fois dans s. Si $R = S$,

- dans $r \bigcup s$, t apparaît $m + n$ fois ;
- dans $r \bigcap s$, t apparaît $min(m, n)$ fois ;
- dans $r \backslash s$, t apparaît $max(0, m - n)$ fois.

Les autres opérations s'étendent aussi aux multi-ensembles :

- projection : la projection de chaque tuple est conservée, même si certains tuples deviennent identiques
- sélection : les tuples qui satisfont la condition de sélection sont conservés, y compris les occurrences multiples
- produit cartésien : toutes les combinaisons de tuples sont considérées avec leur multiplicité ; pour un tuple t_1 qui apparaît m fois dans r et un tuple t_2 n f ois dans s, le tuple $t_1 t_2$ apparaît mn fois dans $r \times s$
- joint : idem que pour le produit cartésien

r_1 :

A	B	C
a	b	c
d	a	f
d	a	f

r_2 :

A	B	C
b	g	a
d	a	f

r_3 :

C	D	E
c	d	e
c	d	e

$r_1 \cup r_2$:

A	B	C
a	b	c
d	a	f
d	a	f
d	a	f
b	g	a

$r_1 - r_2$:

A	B	C
a	b	c
d	a	f

$r_1 \bowtie r_3$:

A	B	C	D	E
a	b	c	d	e
a	b	c	d	e

$r_1 \cap r_2$:

A	B	C
d	a	f

$\pi_{AB} r_1$:

A	B
a	b
d	a
d	a

$r_1 \times r_3$:

A	B	C	C'	D	E
a	b	c	c	d	e
a	b	c	c	d	e
d	a	f	c	d	e
d	a	f	c	d	e
d	a	f	c	d	e
d	a	f	c	d	e

4.1.2 Valeur nulle

Il est souvent utile d'indiquer qu'un attribut d'un tuple n'a pas de valeur, car
- cette valeur n'est pas connue au moment de l'insertion du tuple
- la valeur n'existe pas

Une valeur nulle est représentée par $\perp$ ou `NULL`. On peut éviter les valeurs nulles en décomposant les relations au maximum, mais cela aura un impact négatif sur les performances et la lisibilité du schéma.

4.1.3 Extension de l'algèbre relationnelle

Elimination de tuples redondants $\Delta(r)$ élimine les occurrences multiples de la relation r

Opérateurs calculant des valeurs agrégées Sum, Avg, Min, Max, Count (compte le nombre d'occurrence d'un attribut en tenant compte des occurrences multiples, en fait calcule le nombre de tuples de la relation).

Opérateur de groupement Opérateur $\gamma_L(r)$ où L est une liste d'éléments qui sont
- un élément A du schéma R de r par rapport auquel le groupement est réalisé. Les tuples ayant la même valeur pour l'attribut A sont alors placés dans le même groupe ;
- une opération d'agrégation $OP(A) \to B$ pour indiquer que dans chaque groupe, l'opération OP est appliquée à l'attribut A et le résultat est attribué à l'attribut B.

Le calcul de $\gamma_L(r)$ se fait d'abord en groupant les tuples, puis en introduisant pour chaque groupe un tuple comportant les valeurs des attributs de groupement, et une valeur pour chaque attribut qui correspond à une opération d'agrégation définie dans L.

Opérateur de tri $\tau_L(r)$, où L est la liste ordonnée des attributs par rapport auxquels on trie.

Opérateur de projection étendu On peut vouloir renommer un attribut, on introduit un attribut dont la valeur est calculée à partir d'autres attributs. On a π_L, où les éléments de L sont
- un attribut
- une contrainte de la forme $A \to B$, indiquant que l'attribut A est renommé B
- une contrainte de la forme $expr \to A$, où $expr$ est une expression arithmétique dont les éléments sont des attributs. La valeur de l'expression est prise comme valeur pour l'attribut A dans les tuples de la relation, projetée.

Joint externe (outerjoin) Avec un joint $r \bowtie s$, les tuples de r (s) qui ne peuvent être combinés avec aucun tuple de s (r) sont dit sans appui (dangling), ils n'apparaîtront pas dans le résultat du joint.

Avec le join externe $\overset{\circ}{\bowtie}$, ces tuples sont complétés par des valeurs nulles. Il y a des variantes à gauche et à droite, pour ne compléter les tuples que de la relation à gauche ou à droite. Le θ-joint peut être aussi étendu de cette façon.

4.2 SQL

Une requête `SELECT ...FROM ...WHERE` se détermine ainsi :

1. on examine les combinaisons de tuples de relations $r_1, \ldots, r_m$ (spécifiées avec `FROM`)
2. pour chaque combinaison, on détermine si la clause `WHERE` est satisfaite
3. si oui, on produit un tuple qui comporte les attributs spécifiés par `SELECT`

4.2.1 Opérations booléennes

Mots-clés : union, intersect, except (différence)
Les relations auxquelles on applique ces opérations doivent être de même schéma.

4.2.2 Copies multiples

Par défaut, les copies multiples de tuples sont conservées, sauf avec l'opérateur `union`.
On peut forcer un autre comportement avec les mots clés `all` et `distinct` :
– `SELECT distinct` $A_1, \ldots, A_n$ `FROM` $r_1, \ldots, r_m$ `WHERE P`
– `SELECT ...union all SELECT ...`

4.2.3 Réutiliser une relation

Il est parfois nécessaire de comparer les tuples d'une même relation entre eux.
En algèbre relationnelle, on effectue le produit de la relation avec elle-même. En SQL, on mentionne deux fois la relation dans la clause `FROM`, en attribuant un indicateur à chacune des occurences.
On parle parfois de "variable tuple".
Ex : Quels sont le nom et l'adresse de tous les clients dont le solde dû est inférieur à celui de Dupont, A. ?

```
select C1.NOM, C1.ADRESSE from CLIENTS C1, CLIENTS C2 where C1.SOLDE DU < C2.SOLDE DU
and C2.NOM = 'Dupont, A.'
```

4.2.4 Requêtes imbriquées

La condition `WHERE` de `SELECT` peut être une condition relative à un ensemble lui-même définit par le résultat d'un autre `SELECT`.
Ex : Quels sont les fournisseurs qui fournissent au moins un combustible commandé par 'Dupont, A.' ?

```
select NOM F from FOURNISSEURS where COMB in select COMB from COMMANDES where NOM = 'Dupont,
A.'
```

On peut aussi tester le résultat d'une requête imbriquée et voir si c'est un ensemble vide ou non.
ex : Quel est le nom des clients qui ont émis au moins une commande de bois ?

```
select NOM from
CLIENTS where exists ( select * from COMMANDES where CLIENTS.NOM = COMMANDES.NOM and COMB
= 'bois' )
```

4.2.5 Test d'absence de valeur et valeurs nulles

On peut utiliser `not exists`, ou bien `is null` pour détecter l'absence de valeur.
Lorsqu'un attribut a la valeur nulle, le résultat n'est ni vrai ni faux, il est inconnu.

4.2.6 Joints explicites

- produit cartésien : `cross join`
- joint naturel : `natural join`
- θ - joint : `r join s on`
- joint-externe : mots clés `full outer`, `left outer` ou `right outer`

4.2.7 Agrégation et groupement

`avg, count, sum, min, max, stddev, variance`
On peut partitionner les tuples d'une relation en groupes sur lequels appliquer des fonctions d'agrégation.
`group by` $A_1, \ldots, A_k$

4.2.8 Comparaison de chaînes de caractères

Le caractère `%` dénote un nombre quelconque de caractères quelconques, tandis que _ dénote un caractère quelconque.

Pour les comparaisons, on utilise `like` et `not like`

Ex : Quel est le nom des fournisseurs dont l'adresse contient 'Laville' comme sous-chaîne ?

```
select NOM F from FOURNISSEURS where ADRESSE F like '%Laville%'
```

Donner le nom et le prénom des employés qui sont nés dans les années 60 (sachant qu'une date de naissance est une chaîne de caractères du type aaaammjj).

```
select FNAME, LNAME from EMPLOYEE where BDATE like '196____'
```

4.2.9 Tri

Le tri de fait avec `order by`, de manière ascendante (`asc`) ou descendante (`desc`)

Chapitre 5

Base de données relationnelle : mise en oeuvre et utilisation

5.1 Organisation d'un système de gestion de bases de données

Un système de gestion de bases de données est organisé en plusieurs programmes exécutés séparément :
- un serveur, qui gère les données et attend des requêtes, les exécute et renvoie le résultat
- une ou des applications qui interagissent avec le serveur, et qui peut être une simple interface où taper des commandes SQL, une interface orientée gestion de bases de données ou une interface permettant d'interroger et de modifier la base de données suivent des opérations prédéfinies.

Un utilisateur peut être en interaction directe avec l'application (architecture à deux niveaux), ou bien en interaction avec un troisième programme qui dialogue avec l'application (ex : client web avec une application exploitée dans le contexte d'un serveur web) (architecture à trois niveaux).

5.2 Définition de bases de données et de relations

5.2.1 Définition d'une base de données

Un système de gestion de bases de données peut généralement gérer plusieurs bases de données.

Chaque base de données est un ensemble de tables gérées de façon indépendante et ayant ses propres autorisations d'accès. Une requête ne peut pas utiliser des tables de bases de données différentes.

Pour définir une base de donnée, on peut utiliser une interface de gestion ou des commandes SQL (`create database` et `drop database`). On précise la base de données à utiliser avec la commande `use`.

5.2.2 Définition des tables/relations

Lorsqu'on crée une relation, l'information sur le schéma de la relation est conservé dans un dictionnaire de données (data dictionary).

La commande `create table r (..., ..., ...)` permet de créer des tables.

Ex : create table fournisseurs (NOM F varchar(20) not null, ADRESSE F varchar(50) not null, COMB varchar(10), PRIX int)

5.2.3 Domaines des attributs

Domaines prédéfinis :
- Nombres entiers : int ou integer et smallint
- Nombres en virgule flottante : float, real et double precision
- Nombres en virgule fixe : decimal(i,j) ou dec(i,j) ou numeric(i,j)
- Chaînes de caractères : char(n) ou character(n), et varchar(n) ou char varying(n) ou character varying(n)
- Chaînes de bits (de booléens) : bit(n) et bit varying(n)

Nouveaux domaines (SQL 2) :
- Dates : date (10 car. 'aaaa-mm-jj')
- Heures : time (8 car. 'hh :mm :ss') et time(i) (i+9 car. 'hh :mm :ss :f1. . .fi') et time with time zone
- Dates + heures : timestamp
- Intervalles de temps (que l'on peut ajouter ou soustraire à une date, un temps ou un timestamp) : interval

Ex : `create table DEPARTMENT ( DNO int not null, DNAME varchar(15) not null, MGR ID char(11), MGR START date, primary key (DNO), unique (DNAME), foreign key (MGR ID) references EMPLOYEE (EMP ID) ) ;`

5.3 Contraintes d'intégrité

Lors de la définition d'une relation, on peut spécifier des contraintes d'intégrité. On distingue
- les contraintes locales, qui ne concernent que la relation définie
- les contraintes entre relations, ou contraintes de référence, qui impliquent plus d'une relation

5.3.1 Contraintes d'intégrité locales

- `not null` : impose qu'un attribut ne puise pas être sans valeur
- `default` : spécifie la valeur par défaut
- `primary key` $A_1, \ldots, A_n$: spécifie un ensemble d'attributs constituant une clé, et plus précisément la clé préférentielle
- `unique` $A_1, \ldots, A_n$: spécifie un ensemble d'attributs constituant une clé (autre clé que primaire)

5.3.2 Contraintes d'intégrité référentielles et clés étrangères

Un tuple dans une relation qui fait référence à un tuple d'une autre relation doit faire référence à un tuple existant dans cette relation.

Soient R_1 et R_2 deux schémas de relation $K = \{A_1, \ldots, A_n\} \subseteq R_1$, avec K la clé (primaire) de R_1, et $FK = \{B_1, \ldots, B_n\} \subseteq R_2$, avec $dom(A_i) = dom(B_i)$ pour $1 \leq i \leq n$.

FK est une clé étrangère de R_2 si pour tout $t(R_2)$ de r_2
- soit il existe dans r_1 un tuple $s(R_1)$ tel que $t[FK] = s[K]$ (t fait référence à s)
- soit $t[FK]$ n'a pas de valeur (nulle)

5.3.3 Autres contraintes

On peut définir des contraintes comme des dépendances, des contraintes de cardinalité, contraintes sur les valeurs de un ou plusieurs attributs.

Ces contraintes peuvent être vérifiées au niveau de l'application ou de la base de données. Dans ce dernier cas, il faut que la base de données dispose des possibilités suivantes :
- définition d'assertions
- procédures de la base de données
- possibilité de déclencher ces procédures lorsque certains évènements surviennent (triggers)

5.3.4 Définition de vues

Une vue est une relation dérivée d'autres relations enregistrées dans la base de données : `create view v(`$A_1, \ldots, A_k$`) as Q`, où `Q` est une requête SQL.

Ex : create view tout-brule-vend(COMB, PRIX) as select COMB, PRIX from fournisseurs where NOM F='Tout-Brule'

Les vues fournissent une vue partielle des données et imposent des restrictions d'accès (les mises à jour s'effectuent rarement dans une vue).

Chapitre 6

Gestion des transactions

Pour maintenir la cohérence d'une base de données, on utilise les transactions : ce sont des suites d'opérations (lectures-écritures) effectuées sur une base de données.

Le concept s'utilise comme suit :

1. l'interaction avec la BDD se fait uniquement avec des transactions
2. le système de BDD gère les transactions de manière atomique ; si une transaction est exécutée, elle doit l'être entièrement et de façon indivisible en cas de parallélisme.

On peut définir les transactions dites ACID :
- Atomicity : l'exécution des transactions doit être atomique
- Consistency : l'exécution des transactions doit respecter les contraintes d'intégrité définies par la BDD
- Isolation : il n'y a pas d'interaction entre transactions exécutées en parallèle
- Durability : une fois la transaction effectuée, le résultat ne peut être perdu

6.1 Ordonnancement

On définit un ordonnancement (schedule) d'un ensemble de transactions $\{T_1, \ldots, T_k\}$ est un ordre d'exécution des opérations a_i^j des transactions T_i.

Un ordonnancement est séquentiel (serial) lorsqu'il existe une permutation π de $\{1, \ldots, k\}$ telle que l'ordonnancement est $T_{\pi(1)}, T_{\pi(2)}, \ldots, T_{\pi(k)}$

Un ordonnancement est séquentialisable (serializable) s'il existe un ordonnancement de $\{T_1, \ldots, T_k\}$ qui "donne le même résultat".

Il faut un critère précis pour déterminer quand un ordonnancement est séquentialisable et un protocole d'exécution des transactions pour garantir la séquentialisabilité.

6.1.1 Critère de séquentialisabilité basé sur les conflits

Deux actions a_1 et a_2 sont en conflit si l'exécution de $a_1; a_2$ peut donner un résultat différent de $a_2; a_1$.

Critère Un ordonnancement est séquentialisable par rapport aux conflits s'il peut être transformé en un ordonnancement séquentiel par permutations successives d'actions qui ne sont pas en conflit.

Un ordonnancement peut être séquentialisable sans l'être par rapport aux conflits, l'intérêt du critère "par rapport aux conflits" est qu'il est plus facile à exploiter.

Déterminer la séquentialisabilité Une transaction T_i précède une transaction T_j dans un ordonnancement donné s'il existe une action a_i de T_i qui précède une action a_j de T_j avec laquelle elle est en conflit.

Le graphe de précédence d'un ordonnancement d'un ensemble de transactions $T = \{T_1, \ldots, T_k\}$ est le graphe (V, E) tel que

– l'ensemble des sommets V est l'ensemble des transactions T
– l'ensemble des arcs E est l'ensemble des paires (T_i, T_j) telles que T_i précède T_j dans l'ordonnancement donné

Théorème : un ordonnancement est séquentialisable par rapport aux conflits ssi son graphe de précédence est sans cycle.

Preuve

$\Leftarrow$ si le graphe est sans cycle, il contient au moins un noeud dans lequel n'entre aucun arc.
Les actions de la transaction correspondant à ce noeud ne sont donc en conflit avec aucune action qui les précède dans l'ordonnancement.
Les actions peuvent être donc déplacées vers le début de l'ordonnancement en ne les permutant qu'avec des actions avec lesquelles elles ne sont pas en conflit
En répétant le même raisonnement avec le graphe dont le noeud traité a été éliminé, on peut construire de proche en proche un ordonnancement séquentiel correspondant à l'ordonnancement de départ

$\Rightarrow$ Supposons donc que dans le graphe de précédence, il y a un cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$.
Vu que la permutation d'actions qui ne sont pas en conflit ne change par le graphe de précédence, dans un ordonnancement séquentiel correspondant, les actions de T_1 doivent précéder celles de T_2,...qui précèdent celles de T_n, qui précèdent celles de T_1, ce qui est impossible.

6.2 Algorithme d'ordonnancement : les verrous

Pour garantir la séquentialisabilité, on peut utiliser l'exclusion mutuelle des transactions (par exemple avec un sémaphore), mais cette contrainte est souvent trop restrictive.

On va supposer que l'on n'assure l'exclusion mutuelle qu'au niveau des lectures/écritures. On utilise deux opérations :
– $lock(D, mode)$ ou $mode$ est une lecture r où une écriture w
– $unlock(D)$

La règle est de verrouiller une donnée lors de sa première utilisation et de la déverrouiller après sa dernière utilisation.

6.2.1 Règle des 2 phases (2 phase locking)

Chaque donnée est verrouillée avant sa première utilisation et déverrouillée après sa dernière utilisation.

Aucune opération de verrouillage ne peut suivre une opération de déverrouillage de la même transaction. On distingue donc deux phases : verrouillage et déverrouillage.

Théorème : si un ordonnancement est obtenu en respectant la règle des deux phases, il est séquentialisable.

Preuve

1. Supposons que ce ne soit pas le cas. Le graphe de précédence de l'ordonnancement possède donc un cycle
   $T_{i_1} \rightarrow T_{i_2} \rightarrow T_{i_3} \rightarrow \cdots \rightarrow T_{i_n} \rightarrow T_{i_1}$
2. Le fait qu'il existe un arc dans la graphe de précédence entre T_{i_1} et T_{i_2} indique qu'il y a une ressource pour laquelle les deux transactions sont en conflit et qui est utilisée par T_{i_1} avant T_{i_2}.
3. Cette ressource doit donc être déverrouillée par T_{i_1} avant d'être verrouillée par T_{i_2}. Par conséquent, la phase "lock" de T_{i_2} s'étend au delà la phase "lock" de T_{i_1} (il y a au moins un "unlock" T_{i_1} avant la fin de la phase "lock" de T_{i_2}.
4. Similairement, la phase "lock" de T_{i_3} s'étend au delà de la phase "lock" de T_{i_2} (et donc de T_{i_1}), ..., la phase lock de T_{i_n} s'étend au delà de la phase "lock" de T_{i_1}.
5. Par conséquent la phase "lock" de T_{i_1} s'étend au delà de la phase "lock" de T_{i_1}, ce qui signifie que la règle des deux phases n'est pas respectée.

Le problème est que la règle des deux phases garantit la séquentialisabilité, mais elle n'exclut pas la présence de blocages (deadlocks) ou d'exclusion d'un processus (starvation).

Blocage :

T_0	T_1
lock(A,w)	
	lock(B,w)
lock(B,w) - fail	
	lock(A,w) - fail

Exclusion :

T_{0i}	T_{1i}	T_2
lock(A,r)		
	lock(A,r)	
		lock(A,w) - fail
unlock(A)		
lock(A,r)		
	unlock(A)	
		lock(A,w) - fail
	lock(A,r)	
unlock(A)		
lock(A,r)		
⋮	⋮	

6.2.2 Prévention des blocages

On impose un ordre sur les transactions : $T_i < T_j$ si T_i a démarré avant T_j.

On applique la politique suivante : si une transaction T_i dit avoir accès à une donnée verrouillée par T_j, alors

- si $T_i > T_j$, T_i attend
- si $T_i < T_j$, la transaction T_j est annulée (rolled back) et T_i poursuit son exécution

La transaction la plus ancienne poursuit toujours son opération et il n'y a pas de blocage possible.

6.2.3 Récupération en cas d'erreurs

Le problème est de gérer les transactions qui ont été interrompues.

Les erreurs possibles :

- annulstion d'une transaction
- arrêt du système
- perte d'une partie du contenu du disque

Modèle utilisé :
- les données sont transférées de et vers le disque bloc par bloc (page par page) par des opérations atomiques
- chaque transaction T se termine par
  - soit un $commit(T)$: les modifications sont appliquées à la BDD
  - soit un $abort(T)$: la transaction est annulée, les modifications ne sont pas appliquées

6.2.4 Fichier historique

On utilise un fichier log pour pouvoir récupérer des erreurs ; on y sauve
- le début de chaque transaction
- chaque modification effectuée par une transaction
- les opérations $commit(T)$ effectuées
- les points de synchronisation

Lorsqu'une transaction modifie une page, cela est directement logé, mais les pages ne sont pas modifiées (on retarde donc les opérations unlock).

Si la transaction se termine normalement, $commit(T)$ est ajouté au log, qui est sauvé sur le disque.

Les modifications effectuées par la transaction sont effectivement réalisées avec unlock, les pages correspondantes sont sauvées sur le disque ou non.

On impose périodiquement un point de synchronisation (checkpoint) : les nouvelles transactions sont interdites et on attend que toutes les transactions en cours soient terminées. On sauve ensuite toutes les pages modifiées sur le disque.

Méthodes de récupération :
- arrêt d'une transaction : ne rien faire, car les modifications ne sont pas effectuées
- arrêt du système : avec le fichier historique, on réexécute tous les transactions dont le commit se trouve après le dernier point de synchronisation
- perte de contenu, idem, mais à partir du dernier point de synchronisation correspondant à une sauvegarde que l'on peut restaurer

Le log ne peut être perdu, c'est pour ça qu'il est dupliqué.

6.3 Transactions en SQL

Il y a une gestion complète des transactions ; la fin d'une transaction est signalée par la commande `commmit` ou `rollback`. Le début est signalé par `start transaction`.

Il y a également un verrouillage de tables et un sauvetage immédiat sur disque des modifications. Les verrouillage et déverrouillage se font avec `lock tables` et `unlock tables`.

Il y a plusieurs modes de gestion des transaction : `set transaction isolation level ...`
- `serializable` : le plus contraignant, les transactions sont effectuées de façon séquentialisables
- `read uncommited` : la lecture de données modifiées par d'autres actions qui ne sont pas achevées (commit exécuté) est possible
- `read commited` : on ne lit que des données modifiées par des transactions terminées, mais des lectures multiples des mêmes données peuvent reproduire un résultat différent
- `reapatable read` : on ne voit que des données modifiées par des transactions terminées et les lectures multiples donnent le même résultat, si ce n'est que des nouveaux tuples peuvent apparaître entre deux lectures.

Le choix de niveau d'isolation est un compromis entre fiabilité, facilité d'utilisation et performances. La définition exacte des niveaux d'isolation dépend du système utilisé.

Chapitre 7

Implémentation du modèle relationnel

7.1 Les disques et leurs caractéristiques

Un disque comporte un ensemble de surfaces. Sur chaque surface, on trouve un ensemble de pistes (track) concentriques.

Chaque piste est divisée en secteurs de capacité fixe. Ce sont les plus petits groupes de données adressables et transférables sur un disque.

Un cylindre est l'ensemble des pistes se trouvant à une position identique sur les différentes surfaces.

Les secteurs sont parfois aussi appelés blocs, mais ce terme désigne aussi des groupes de secteurs traités logiquement comme une seule entité au niveau du système de gestion du disque.

Pour accéder à un secteur, il faut positionner les têtes de lecture du disque sur le bon cycle (seek time, le temps de recherche) et attendre que le bon secteur passe sous les têtes de lecture (latency, temps de latence). La durée moyenne est de l'ordre de 10ms, mais pour des secteurs consécutifs on a des temps de l'ordre de 10 μs par secteur.

7.2 Technologie RAID

Cette technologie permet d'obtenir des grosses capacités et de meilleurs performances, en utilisant un ensemble de disques qui seront gérés comme un seul disque virtuel.

Il y a plusieurs niveaux RAID :

- striping (niveau 0) : écriture par bande ; on distribue les données entre les disques du RAID, pour paralléliser les accès.
  On distingue deux types de striping :
  - le striping au niveau du bit : on crée des groupes de 8 disques et les bits de chaque octet sont répartis entre ces disques
  - le striping au niveau du bloc : on utilise n disques et on écrit le bloc i sur le bloc $(i \mod n) + 1$
- mirroring (niveau 1) : partitionnement des disques en deux groupes, et maintient sur chacun des groupes d'une copie complète des données. Chaque opération d'écriture est réalisée simultanément et symétriquement sur chacun des 2 groupes.

Les autres niveaux RAID prévoient une redondance par l'utilisation de codes de correction d'erreur.

L'écriture par bandes peut être combinée à l'écritoire miroir ou avec les codes de correction d'erreur.

Avec une redondance suffisante, on arrive à une excellente fiabilité et on peut même remplacer les disques défaillants sans arrêt du système (échange à chaud, hot swapping).

7.3 Implémentation des relations par les fichiers ISAM

Pour permettre l'utilisation des disques, les systèmes d'exploitation fournissent un système de gestion de fichiers, des ensembles de données auquel on attribue un nom. Les noms des fichiers sont gérés dans un

catalogue (directory) hiérarchique qui permet de trouver rapidement les fichiers conservés sur un disque.

Les fichiers de base disponibles dans un système d'exploitation sont vu comme une séquence d'octets destinée à être lue et écrite séquentiellement.

Pour implémenter une base de données, il faut des fichiers organisés sous la forme d'un ensemble de tuples ou enregistrements (records) et des technique permettant de retrouver rapidement un enregistrement à partir de sa clé (ou à partir d'autres attributs)

Les systèmes de BD utilisent des fichiers ISAM (Indexed Sequential Access Method) pour conserver les relations. Un parcours séquentiel est possible, mais on ajoute des index (indexes) permettant de retrouver rapidement le bloc dans lequel se trouve un enregistrement.

Habituellement, on construit un index pour la clé des enregistrements (et suivant les besoins pour d'autres attributs).

7.3.1 Les B-trees

Un B-tree est utilisé pour les index ; c'est un arbre équilibré et optimisé pour l'utilisation sur disque.

L'équilibre est maintenu en le réorganisant périodiquement, mais en laissant varier le nombre de successeurs de chaque noeud dans certains limites. L'implémentation sera optimale si les noeuds du b-tree ont la taille d'un secteur.

Pour n enregistrement, la recherche se fait en $\mathcal{O}(n)$ sans index, en $\mathcal{O}(\log n)$ avec un index. Par contre, plus il y a d'index pour un fichier, plus les opérations de modification sont longues.

7.3.2 Les tables de hash

Deuxième forme d'index plutôt utilisée pour des relations temporaires conservées en mémoire.

Les enregistrements sont répartis entre des bacs regroupés dans un tableau. Un bac peut contenir un ou plusieurs enregistrements (chaînage).

On prévoit un nombre de bacs supérieur au nombre d'enregistrements prévus, ce qui donne un nombre moyen d'enregistrements par bac inférieur à 1.

Le numéro de bac dans lequel un enregistrement est placé est calculé à partir de la clé de l'enregistrement par une fonction de hash, qui idéalement répartit les enregistrements uniformément entre les bacs. Pour accéder à un enregistrement, il suffit de calculer son numéro de bac avec la fonction de hash.

Les fonctions de hash parfaites n'existent pas, mais il est possible de trouver des fonctions avec d'excellents résultats. Pour n enregistrements, le temps d'accès est de $\mathcal{O}(1)$.

7.3.3 Index en SQL

Les index sont spécifiés au moment de créer la table. Par défaut, un index est créé pour la clé primaire ; il est possible d'ajouter ou de supprimer un index d'une table (`create index`, `drop index`).

Il est parfois possible de préciser si un index doit être réalisé à l'aide d'un B-tree ou d'une table de hachage.

7.4 Implémentation des opérations de l'algèbre relationnelle

On va poser

- n_r le nombre de tuples de r
- $V(X, r)$: le nombre de valeurs distinctes qui apparaissent dans r pour les attributs X ; nombre de tuples distincts dans $\pi_X r$.
  Le nombre moyen d'occurences de tuples de même valeurs pour les attributs X est donc $\frac{n_r}{V(X,r)}$
- s_r la taille des tuples, pour par exemple déterminer le nombre d'enregistrements par bloc.

Opérations booléennes

$r_1 \bigcup r_2$ Avec une implémentation directe, on a $\mathcal{O}(n_{r_1} + n_{r_2})$.

Si on peut éliminer les occurrences multiples, on peut

- trier le résultat : $\mathcal{O}((n_{r_1} + n_{r_2}) \log(n_{r_1} + n_{r_2}))$

– utiliser un index existant, par exemple pour r_1 et n'ajouter à r_1 que les tuples de r_2 qui ne figurent pas dans r_1 : $\mathcal{O}(n_{r_1} + n_{r_2} \log(n_{r_1})$

$r_1 - r_2$ on peut
– trier les deux relations et les parcourir ensuite séquentiellement, en éliminant de r_1 les tuples de r_2 : $\mathcal{O}(n_{r_1} \log(n_{r_1}) + n_{r_2} \log(n_{r_2}))$
– utiliser un index existant pour r_2 : considérer chaque tuple de r_1 et le chercher dans r_2 en utilisant l'index : $\mathcal{O}(n_{r_1} \log(n_{r_2}))$

Sélection et projection

$\sigma_F r_1$
– au pire considérer chaque tuple de r_1 et tester la condition F : $\mathcal{O}(n_{r_1})$
– meilleur complexité s'il y a des index adaptés à la condition de sélection

$\pi_X r_1$
– en général : parcours de r_1 avec extraction des composantes nécessaires de chaque tuple : $\mathcal{O}(n_{r_1})$
– si on veut éliminer les occurrences multiples, il faut trier la relation r_1 : $\mathcal{O}(n_{r_1} \log(n_{r_1}))$

Produit cartésien et joint

$r_1 \times r_2$ Par définition, le produit cartésien contient $n_{r_1} \times n_{r_2}$ tuples, et nécessite un temps de calcul $\mathcal{O}(n_{r_1} \times n_{r_2})$

$r_1 \bowtie r_2$ les stratégies diffèrent selon qu'il existe un index ou non pour l'une ou l'autre relation.
– Joint sans index : soient X les attributs communs de R_1 et R_2.
Naïvement, on peut examiner toutes les paires de tuples et comparer les valeurs et les attributs communs : $\mathcal{O}(n_{r_1} \times n_{r_2})$
Ou bien on trie les deux relations par rapport à leurs attributs communs, et on fusionne les résultats. On a $\mathcal{O}(n_{r_1} \log(n_{r_1}) + n_{r_2} \log(n_{r_2}))$ pour autant que $\frac{n_{r_i}}{V(X,r_i)}$ soit borné par les attributs communs X.
– Joint avec une table de hash : on construit une table de hash pour r_2 sur les attributs communs X. On parcourt r_1 et pour chaque tuple de r_1 (et sa valeur pour X), on va chercher dans la table de hash les tuples de r_2 qui ont la même valeur pour X.
Complexité : $\mathcal{O}(n_{r_1} + n_{r_2})$, pour autant que $\frac{n_{r_2}}{V(X,r_2)}$ soit borné pour les attributs communs X.
– Joint en présence d'un index : on suppose que r_2 possède un index pour les attributs communs X. On parcourt r_1 et pour chaque tuple de r_1 (et sa valeur pour X), on va chercher dans r_2 les tuples qui ont la même valeur pour X.
Complexité : $\mathcal{O}(n_{r_1}(1 + \log(n_{r_2})))$

7.5 Optimisation des requêtes

Principes généraux de l'optimisation :
– générer différentes stratégies d'évaluation et en estimer le coût. Choisir la meilleure.
– règle heuristique : effectuer les sélections et projections dès que possible
– il est utile d'avoir des informations statistiques sur le contenu de la base de données pour bien estimer le coût de différentes stratégies d'évaluation

Soient des relations r de schéma R, s de schéma S et u de schéma U.

7.5.1 Associativité et commutativité du joint

$$r \bowtie s = s \bowtie r$$

$$r \bowtie (s \bowtie u) = (r \bowtie s) \bowtie u$$

Cela peut optimiser le calcul de joints multiples.

7.5.2 Groupement des conditions de sélection

$$\sigma_{F_1}\sigma_{F_2}r = \sigma_{F_1 \wedge F_2}r$$

Utilité :
- remplacer plusieurs sélections par une seule
- décomposer une sélection pour exploiter un index
- décomposer une sélection pour qu'elle puisse en partir être permutée avec une autre opération

7.5.3 Sélection et projection

$$\pi_X\sigma_{A=a}r = \pi_X\sigma_{A=a}\pi_{X \bigcup A^r}$$

Cela permet d'appliquer la sélection à une relation plus petite.

7.5.4 Sélection et joint

Soit un attribut $A \in R$

$$\sigma_{A=a}(r \bowtie s) = (\sigma_{A=a}r) \bowtie s$$

Cela permet de réduire la taille des relations avant le calcul d'un joint.

7.5.5 Sélection et union ou différence

$$\sigma_{A=a}(r \bigcup s) = (\sigma_{A=a}r) \bigcup (\sigma_{A=a}s)$$
$$\sigma_{A=a}(r - s) = (\sigma_{A=a}r) - (\sigma_{A=a}s)$$

On réduit la taille des relations dès que possible.

7.5.6 Projection et joint

$$\pi_R(r \bowtie s) = \pi_X(\pi_{R \bigcap (X \bigcup S)}r \bowtie \pi_{S \bigcap (X \bigcup R)}s)$$

7.5.7 Projection et union

$$\pi_X(r \bigcup s) = (\pi_X r) \bigcup (\pi_X s)$$

Minimiser la taille des relations avant le calcul de l'union.

Chapitre 8

Les bases de données déductives

L'idée est de coupler une bdd à un ensemble de règles logiques pour en déduire de l'information, et sans passer par un langage de programmation.

Une base de données déductive est constituée
- de prédicats de base (ou extensionnels), dont l'extension est conservée explicitement dans la base de données (base de données extensionnelle)
- de prédicats dérivés (ou intentionnels), dont l'extension est définie par des règles déductives (base de données intentionnelle)

8.1 Datalog

8.1.1 Syntaxe

Un terme est soit une variable, soit une constante. Une formule atomique ou un atome est un symbole prédicatif appliqué à une liste d'arguments qui sont soit des variables, soit des constantes.

Un symbole prédicatif peut être
- un nom de prédicat de base (extensionnel) (ex : parent(X,Y))
- un prédicat dérivé (intentionnel) (ex : grandparent(X,Y))
- une relation arithmétique

Ex : $grandparent(X, Y) \leftarrow parent(X, Z) and parent(Z, Y)$

Restrictions :
- les prédicats apparaissant dans la tête (membre de gauche) de règles sont uniquement des prédicats dérivés, les règles ne définissent donc que des prédicats dérivés.
- conditions de sûreté : toute variable utilisée dans une règle doit apparaître dans au moins un atome dont le prédicat représente une relation, et qui n'est pas précédé d'une négation.
  Cela permet de garantir que l'évaluation des règles est possible.

Ex :
- Non sûr : $q(X, Y) \leftarrow p(X, Z)$ AND NOT $r(X, Y, Z) AND X \geq Y$
- Sûr : $q(X) \leftarrow p(X)$ AND $X \geq 0$

8.1.2 Interprétation des règles datalog

Calcul de l'extension des prédicats intentionnels définis par des règles datalog :

1. si un prédicat intentionnel est la tête de plusieurs règles, son extension est l'union des extensions données par les différentes règles
2. pour chaque règle, on considère tous les tuples des relations extensionnelles non précédées d'une négation figurant dans le corps de la règle.
   Pour chaque combinaison de tuples, si
   - les tuples sont compatibles avec les occurrences des prédicats extensionnels (attributs pour lesquels une valeur constante est donnée ayant cette valeur)
   - les prédicats relationnels niés et les prédicats arithmétiques sont satisfaits

   alors on ajoute le tuple défini par la règle.

La sureté garantit que l'on a ainsi pris en compte toutes les valeurs possibles.

8.1.3 Conversion de l'algèbre relationnelle

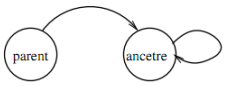
- $r \bigcup s : rus(\overline{t_1}) \leftarrow r(t1)$ et $rus(\overline{t_1}) \leftarrow s(\overline{t_1})$
- $r - s : rms(\overline{t_1}) \leftarrow r(\overline{t_1})$ AND NOT $s(\overline{t_1})$
- $r \times s : rts(\overline{t_1}, \overline{t_2}) \leftarrow r(\overline{t_1})$ AND $s(\overline{t_2})$
- $r \bowtie s : rjs(\overline{t_1} \bigcup \overline{t_2}) \leftarrow r(\overline{t_1})$ AND $s(\overline{t_2})$
- $\pi_X r : pr(\overline{t_1} \bigcap X) \leftarrow r(\overline{t_1})$
- $\sigma_C r : sr(\overline{t_1}) \leftarrow r(\overline{t_1})$ AND $C(\overline{t_1})$

8.1.4 Structure des règles et récursivité

Pour un ensemble de règles donné, le graphe des règles est défini comme suit :
- il y a un noeud dans le graphe pour chaque symbole prédicatif
- il existe un arc d'un noeud p à un noeud q ssi il existe une règle de la forme $q(\dots) \leftarrow \dots p(\dots) \dots$

Un ensemble de règles est récursif si son graphe comporte un cycle.

```
ancetre(X,Y)  ←  parent(X,Y)
ancetre(X,Y)  ←  parent(X,Z) AND ancetre(Z,Y)
```



Pour évaluer des règles récursives, le principe est de ne mettre dans l'extension des prédicats intentionels récursifs que les tuples qui doivent y figurer.

On commence par une extension vide pour ces prédicats, puis on applique les règles jusqu'à ce qu'on obtienne une extension stable, ce qu'on appelle un point fixe (fixpoint).

On peut utiliser cette approche tant que les prédicats définis récursivement ne sont pas dans la portée d'un `not`.

Exemple pour le prédicat ancêtre.

ancêtre	
anna	bob
bob	chris
bob	dan
dan	eric

ancêtre	
anna	bob
bob	chris
bob	dan
dan	eric
anna	chris
anna	dan
bob	eric

ancêtre	
anna	bob
bob	chris
bob	dan
dan	eric
anna	chris
anna	dan
bob	eric
anna	eric

On peut utiliser la négation et la récursion en même temps, pour autant qu'elles n'interagissent pas. On impose une restriction : les règles sont stratifiées.

Si une règle comporte une négation $q(\dots) \leftarrow \dots$ NOT $p(\dots)\dots$, il n'y a pas de chemin de q à p dans le graphe des règles.

Le graphe des règles peut alors être stratifié, divisé en couches : il n'y a pas de récursion entre couches et dans une règle définissant un prédicat d'une couche, la négation n'est appliquée qu'à des prédicats des couches inférieures.

On peut alors évaluer l'extension des prédicats dérivés par couche.

8.1.5 Expressivité

Les prédicats définis par des règles non récursives peuvent être exprimés en algèbre relationnelle.

Les prédicats définis par des règles récursives ne sont pas toujours exprimables. Par exemple, la fermeture transitive d'une relation (comme la relation ancetre) : son calcul nécessite l'application d'un nombre d'opérations qui dépend du contenu de la base de données.

Chapitre 9

Bases de données orientées objet

Certaines applications ont des besoins nouveaux, notamment
- des structures d'objets plus complexes
- des transactions de durée plus longue
- de nouveaux types de données

Les BDD OO sont une tentative de réponse à ces nouveaux besoins. Une caractéristique importante est qu'elles donnent au concepteur de la BDD de spécifier
- la structure d'objets complexes
- les opérations à appliquer à ces objets

Il y a actuellement plusieurs prototypes et quelques produits commerciaux de BDD OO.

Ces systèmes reprennent les concepts adoptés par la programmation OO, avec les spécificités des BDD (persistance des données, transactions, etc). Il permettent donc de décrire des objets complexes dans une base de données, qui sont caractérisés par une ensemble structuré de valeurs (et pas une seule). On peut faire référence à un objet complexe, par exemple si le domaine d'un attribut de relation est un ensemble d'objets complexes, il faut que les objets aient un identificateur.

9.1 Objets complexes

On dénode $\{D_i\}$ un ensemble fini de domaines, A un ensemble dénombrable d'attributs et ID un ensemble dénombrable d'identificateurs.

On définit l'ensemble des valeurs V subdivisé en
- valeurs de base, qui sont soit *nil* (valeur non définie) soit des éléments $d \in D$
- valeurs-ensembles : des sous-ensembles de ID
- valeurs-tuples : des fonctions (partielles) de A vers ID

Un objet est une paire (id, v), où $id \in ID$ est l'identificateur de l'objet et $v \in V$ la valeur de l'objet.

Dans tout ensemble d'objets considérés, deux objets distincts ne peuvent avoir le même identificateur.

9.1.1 Types d'objets

On distingue les objets selon leur type (schéma, structure) :
- les objets de base : ce sont des paires (id, v) où v est une valeur de base. Le type d'un objet de base (id, v) (où $v \in D_i$) est le domaine D_i. Le type de *nil* est NIL.
- les objets-ensembles : ce sont des paires (id, v) où v est une valeur-ensemble. Le type d'un objet-ensemble (id, v) est $2T$, où T est l'union des types des objets (id', v') tels que $id' \in v$
- les objets-tuples : ce sont les paires (id, v) où v est une valeur-tuple. Le type d'un objet tuple (id, v) est $\{(a_i, T_i)\}$ tel que $v(a_i)$ est défini et tel que T_i est le type de l'objet (id', v') pour lequel $id' = v(a_i)$.

Exemple.

Pour définir les routes en tant qu'objets complexes, on considère un ensemble de domaines de base contenant
- D_{lat} : le domaine des latitudes,
- D_{long} : le domaine des longitudes,
- D_{id} : le domaine des identifiants de routes,
- D_{cat} : le domaine des catégories,
- D_{aut} : le domaine des autorités.

Un réseau routier est représenté par des *objets* de la manière suivante.
- Un point d'intersection est un objet tuple de type $T_{point} = \{(\mathit{LATITUDE}, D_{lat}), (\mathit{LONGITUDE}, D_{long})\}$.
- Un segment est un objet tuple de type $T_{seg} = \{(\mathit{POINT1}, T_{point}), (\mathit{POINT2}, T_{point}), (\mathit{CATEGORIE}, D_{cat})\}$.
- Une liste de segments est un objet ensemble de type $2^{T_{seg}}$.
- Une route est un objet tuple de de type $T_{route} = \{(\mathit{ID\_R}, D_{id}), (\mathit{LISTE\_SEG}, 2^{T_{seg}}), (\mathit{AUTORITE}, D_{aut})\}$.

9.1.2 Manipulation et encapsulation

Les méthodes sont des procédures de manipulation d'objets. Principe d'encapsulation : seules les méthodes associées à un objet peuvent être utilisées pour le manipuler.

Une classe est un ensemble d'objets de même type, elle correspond à un ensemble d'entités. Elle peut être elle-même un objet.

Les objets de certaines classes peuvent être des particularisation d'objets d'autres classes : c'est l'héritage.

Dans un langage de programmation classique, le lien entre un appel de procédure et le texte du programme correspondant se fait à la compilation. En OO, cela se fait au moment de l'exécution : c'est la liaison tardive (late binding). Il n'est donc pas nécessaire de connaître le type d'un objet dans un programme qui le manipule. On peut donc écrire des programmes génériques qui peuvent s'appliquer à différentes classe d'objets.

En plus de ces fonctionnalités, les bases de données OO doivent avoir les possibilités suivantes :
- complétude au niveau du calcul ; toutes les requêtes calculables doivent être expressibles
- le système doit être extensible, soit permettre la définition de nouvelles classes
- traiter des grandes quantités de données
- gérer des transactions (parallélisme, récupération après erreur)
- possibilité de poser aisément des requêtes simples

9.1.3 Langage d'interrogation

Il n'y a pas de solution largement répandue. On trouve des langages de programmation orienté-objets et des langages spécifiques permettant l'interrogation directe de la base de données de façon plus déclarative.

9.1.4 Le modèle objet-relationnel

Le modèle objet permet de traiter des objets complexes et donne de la flexibilité au niveau des types de données possibles. Le modèle relationnel est associé à un langage d'interrogation de haut niveau et permet l'exécution efficace de requêtes. Combiner les deux permettrait d'avoir les avantages des deux.

Le modèle relationnel-objet est vu comme le modèle relationnel avec des extensions orientées-objets. On a les principes suivants :
- la possibilité de définir des types (UTD : user defined types), autrement dit définir une classe et des méthodes, avec une hiérarchisation si nécessaire
- l'utilisation des références REF, qui identifient les objets, mais qui restent visibles
- la définition de tables dont les lignes sont des objets plutôt que des tuples. La définition de valeurs "ensemble" pour un attribut est possible
- la possibilité d'avoir des relations comme valeurs pour certains attributs ; on parle de nested tables.

Le langage SQL a été étendu pour permettre la définition et l'interrogation de BDD objet-relationnel. Il y a utilisation de méthodes sur les types définis, on peut suivre les références (DEREF) et l'interrogation des tables imbriquées est possibles avec THE(), qui permet de traiter une table imbriquée comme une table ordinaire.

Chapitre 10

L'intégration de données

Le problème de l'intégration de données est de permettre un accès cohérent à des données d'origine, de structuration et de représentation différentes.

Deux techniques existent :

- les entrepôts de données (data warehouses), qui regroupent des données pour les analyser statistiquement et les exploiter en gestion stratégique
- le langage XML (eXtended Markup Language), qui est un cadre générique de représentation des données (semi-)structurées

10.1 Entrepôts de données

Les informations d'une BDD opérationnelle (exploitées pour la gestion quotidienne) peuvent être aussi utiles pour une gestion plus "stratégique". Le type de requêtes nécessite un traitement lourd (donc avec un impact sur les performances de la BDD opérationnelle), pas toujours facilement expressible en SQL, et qui implique parfois des données se trouvent dans des BDD distinctes.

On effectue donc ce type de traitement dans un entrepôt de données, une BDD spécialement conçue pour cette utilisation.

Un entrepôt de données est une base de données

- consolidant les données de bases de données opérationnelles
- utilisée en consultation et mis à jour périodiquement
- organisée pour permettre le traitement de requêtes analytiques (OLAP) plutôt que transactionnelles (OLTP).

On a

- OLTP (on-line transaction processing) : petites transactions consistant en une recherche d'informations, et souvent une mise à jour.
- OLAP (online analytical processing) : grosses transactions impliquant une fraction importante des données (par exemple un calcul statistique)

10.1.1 BDD opérationnelle vers l'entrepôt de données

Les informations d'un entrepôt de données sont issues de BDD opérationnelles, mais leur schéma est en général différent, et les mises à jour ne sont que périodiques.

On appelle *extraction* le processus par lequel les données opérationnelles sont transférées vers l'entrepôt. Le schéma de l'entrepôt est considéré comme une vue sur les données opérationnelles, elle est dite matérialisée vu que les relations correspondant à la vue sont effectivement créées.

Il est utile (mais difficile) de pouvoir mettre à jour les données de l'entrepôt sans devoir régénérer toute son entièreté : c'est une mise à jour incrémentale.

10.1.2 Organisation d'un entrepôt de données

On utilise des schémas en étoile ; on distingue deux types de données :

- les faits, une grosse accumulation de données reprenant des faits simples

– les données dimensionnelles, en quantité réduite et souvent statiques, qui précisent des informations sur les éléments apparaissant dans les faits.

Exemple : on a les faits dans une relation `vente(bar, bière, consommateur, date, heure, prix)` et les données dimensionnelles `bars(bar, adresse, gérant)`, `bieres(biere, taux_alcool, fabriquant)` et `consommateurs(consommateur, adresse, date_naissance)`

On distingue deux types d'attributs dans la table des faits :
– les attributs de dimension, les clés des relations dimensionnelles
– les attributs dépendants, les valeurs déterminées par les attributs de dimension

Par exemple, dans la relation `vente`, l'attribut `prix` est un attribut dépendant déterminé par les attributs `bar, biere, consommateur, date, heure`.

10.1.3 Extraction d'informations

Il y a deux approches :
– ROLAP (relational on-line analytical processing) : expression des requêtes en SQL
– MOLAP (multidimensional on-line analytical processing) : on voit l'entrepôt comme une BDD multidimensionnelles, dont le modèle est un cube à n-dimensions. Il y a une dimension du cube par attribut dimensionnel, et les éléments du cube sont les valeurs des attributs dépendants.

L'approche ROLAP

Une requête ROLAP est généralement exprimée ainsi :

1. calcul du joint de la table des faits et des relations dimensionnelles
2. sélection des tuples en fonction des données dimensionnelles
3. groupement de ces données suivant certains dimensions
4. calcul d'une valeur agrégée

Exemple :

Pour chaque bar d'Angleur, trouver la somme des ventes de chaque bière produite par Interbrew.

2. On filtre le joint de la table des faits et des tables dimensionnelles par `fabriquant` = Interbrew et `adresse` = Angleur ;
3. On groupe le résultat par `bar` et `biere` ;
4. On agrège en calculant la somme de `prix`.

Les requêtes ROLAP peuvent être coûteuses. Il y a des techniques pour les accélérer :
– les index "bitmaps" : on accélère la sélection suivant les dimensions en créant pour chaque valeurs des clés de dimension un vecteur de bits indiquant quels tuples de la table des faits ont cette valeur.
– les vues matérialisées : précalcul de certains joints

L'approche MOLAP

Les clés des tables de dimension sont considérées comme les axes d'un hypercube. Les attributs dépendants apparaissent comme les points du cube. Ce dernier peut aussi contenir des valeurs agrégées suivant les axes, plans, hyperplans, etc.

Pour exploiter un cube de données, on a les opérations suivantes :
– roll-up : sélectionner et agréger suivant certains axes (ex : grouper les bars par région). On parle de slicing (sélectionner suivant une valeur) et de dicing (sélectionner selon un domaine de valeurs)
– drill-down : désagréger certains groupements (ex : grouper les bières par tuax d'alcool et les séparer par fabriquant)
– éliminer certains dimensions (projeter) en remplaçant les points du cube par les agrégats correspondants (pivoting).

10.1.4 La fouille des données (data mining)

L'idée est de trouver des informations dans un entrepôt en allant plus loin qu'une requête simple. Ex : pour une BDD de ventes `ventes(id_panier, produit)`, on cherche les éléments apparaissant souvent simultanément.

Une approche naïve est de fixer un seuil de support (pourcentage de paniers dans lesquels une paire de produits apparaît) pour sélectionner les paires intéressantes. On calcule pour cela le joint de la relation `ventes` avec elle-même. Puis on groupe par paires de produits et on sélectionne sur la fréquence.

Le coût en performances est énorme. Pour y remédier, on peut sélectionner a priori des articles isolés vendus fréquemment. Seuls ceux là peuvent apparaître dans les paires fréquentes.

10.2 Le langage XML

L'origine se trouve dans les langages d'annotation de textes comme HTML. SGML (Standard Generalized Markup Language) et XML (une révision simplifiée et conçue pour le web) permettent d'utiliser un ensemble définissable d'annotations.

Contrairement à une BDD où le schéma est fixé et conservé séparément des données, en XML, le schéma est incorporé dans les données. Cela permet beaucoup de flexibilité ; le schéma et le contenu d'une BDD peuvent être représentés facilement en XML, indépendamment du type de BDD utilisé.

Le XML permet donc l'exportation d'information d'une BDD. Il permet aussi d'extraire des informations d'une BDD et les fournir à une application.

10.2.1 Structure d'un document

```
<?xml version="1.0" encoding="UTF-8"?>
<note date="15/04/2004">
<a>Latour</a>
<de>Wolper</de>
</note>
```

La première ligne déclare qu'il s'agit d'un document XML et donne le codage des caractères. Le reste sont des éléments compris entre une annotation (tag) d'ouverture et de fermeture et sous forme d'arbre (un élément pouvant en inclure d'autres).

Une annotation peut avoir des attributs. Quand il s'agit de données, il est préférable d'utiliser l'élément, tandis que l'attribut est naturel pour des méta-données (informations au sujet des données).

Les annotations n'ont pas de signification par elles-mêmes, mais elles peuvent être interprétées par une application. Chaque document comporte néanmoins un élément de départ, appelé la racine (root element).

10.2.2 Définition de types de documents

Un document XML est bien formé s'il est syntaxiquement correct. Il est valide s'il est conforme à une définition de type de document (DTD - data type definition). Cette définition se trouve dans l'entête, ou dans un fichier auxiliaire mentionné dans l'entête.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note [
  <!ELEMENT note (a,de,sujet,texte)>
  <!ELEMENT a      (#PCDATA)>
  <!ELEMENT de     (#PCDATA)>
  <!ELEMENT sujet  (#PCDATA)>
  <!ELEMENT texte  (#PCDATA)>
]>
<note>
...
</note>
```

La définition spécifie les éléments qui peuvent apparaître dans le document et les annotations correspondantes. `#PCDATA` indique qu'un élément est constitué de texte (Parsed Character Data). Le "parsed" indique que le texte sera analysé pour y détecter des annotations éventuelles. `#CDATA` indiquerait que le texte ne doit pas être analysé.

Lorsqu'on écrit la liste des constituants d'un élément, on peut préciser la répétition de certains éléments : ? (zéro ou une fois), * (zéro fois ou plus), + (une fois au plus). On a par exemple `<!ELEMENT note(message+)>`.

Un choix est indiqué par une barre verticale, par exemple `<!ELEMENT note(a, de,header,(message|texte))>`.

On peut aussi préciser les attributs des différents éléments et leur nature, par exemple `<!ATTLIST note date type CDATA #REQUIRED>`.

Le dernier élément est la valeur par défaut ; `#REQUIRED` indique qu'il n'y en a pas et qu'une valeur doit figurer dans le document pour l'attribut.

Différents types sont possibles pour les attributs, certains sont particuliers :

– ID : identificateur unique
– IDREF : une référence à un identificateur d'élément
– IRREFS : liste d'indicateurs d'éléments

Cela permet de faire référence au même élément à plusieurs endroits d'un document.