Conception de Bases de Données Relationnelles

Marc Plantevit
Faculté de Sciences et Technologie - Laboratoire LIRIS -
Université Lyon 1

Pré-requis :

- Structure du modèle relationnel : attributs, relations, bases de données.
- Langages du relationnels : Algèbre et calcul relationnel, SQL
- Connaissance élémentaire du modèle E/A.
- Voir cours LIF04 d'E. Coquery (http://liris.cnrs.fr/ecoquery/dokuwiki/doku.php?id=enseignement:lif4:st ainsi que celui de N. Lumineau (http://www710.univ-lyon1.fr/~nluminea/index.php?ue=LIF4).
- **Présupposés théoriques :** théorie des ensembles (graphes, arbres, treillis), théorie de la complexité et des automates (automates finis, machines de Turing, théorie de la calculabilité, théorie de la complexité), logique mathématique élémentaire. La théorie des bases de données s'appuie sur ces différents concepts fondamentaux.

Objectif de l'enseignement : Approfondir les connaissances du modèle relationnel et les fondements de la conception de bases de données :

- Contraintes et dépendances,
- Notions de systèmes d'inférences et de bases de règles,
- Processus de normalisation d'une base de données à partir des contraintes.
- Conception à l'aide d'un modèle conceptuel de données.

Remarque : Ce document regroupe des éléments de cours et a pour but de faciliter le travail personnel des étudiants. Il ne remplace pas les cours et exercices effectués en classe. Ces notes de cours s'appuient sur les notes de F. De Marchi http://liris.cnrs.fr/fabien.demarchi/PageWebIF10/Cours.pdf.

Références :

- S. Abiteboul, R. Hull et V. Vianu. *Fondements des bases de données*. Addison-Wesley, 1995.
- R. Ramakrishnam et J. Gehrke. *Database Management Systems*. 2003 (troisième édition).
- Matériel disponible en ligne allant avec la référence précédente : http://pages.cs.wisc.edu/~dbbook/.

Table des matières

Chapitre 1

Introduction

Sommaire

Les ordinateurs sont désormais très largement utilisés dans presque toutes les activités humaines. Une de leur principale utilisation est la manipulation de l'information, ce qui signifie, dans certains cas, tout simplement que l'on stocke des données pour les reprendre plus tard, et dans d'autres cas, que cela sert de cadre pour manipuler le cycle de vie de processus financiers ou industriels complexes. Une grande quantité de données stockée dans un ordinateur est appelée *base de données*. Le logiciel qui permet la manipulation de ces données est appelé *système de gestion de bases de données* (SGBD). Dans ce chapitre, on présente les systèmes de bases de données afin d'indiquer le contexte le plus large qui a conduit à cette théorie.

1.1 Principes

Les systèmes de bases de données peuvent être considérés comme des *médiateurs* entre les utilisateurs qui veulent utiliser des données et les outils physiques qui contiennent les données. Les premières manipulations sur les bases de données étaient fondées sur l'usage explicite de systèmes de fichiers et les logiciels d'application étaient construits de façon *ad hoc*. Petit à petit, des principes et des mécanismes ont été développés et *détachèrent les utilisateurs des bases de données de l'implémentation physique*. A la fin des années 1960, la première étape importante a été le développement de l'*architecture 3 niveaux*. Cette architecture séparait les fonctionnalités des bases de donnée en niveaux physiques, logique et externe.

La séparation de la définition logique des données de son implémentation physique est fondamentale pour la description des bases de données. Des *modèles de données* logiques ont été définis, visant à permettre à l'utilisateur de se concentrer uniquement sur une représentation logique des données sans se soucier de la représentation physique des données. Plusieurs modèles ont été développés, comprenant les modèles *hiérarchiques, en réseaux, relationnels, et orientés objet*. Ils sont constitués d'un *langage de définition des données* (LDD), pour définir les aspects structurels des donneés, et d'un *langage de manipulation des données* (LMD) pour y accéder et les mettre-à-jour. La séparation des aspects logique et physique a ainsi permis un accroissement extraordinaire de l'utilisation des bases de données et de la productivité des programmeurs.

Un autre avantage non négligeable de cette séparation est que plusieurs aspects de l'implémentation physique peuvent être changés sans avoir à modifier la vision abstraite de la base de données.

La séparation des niveaux logique et physique est appelée *principe d'indépendance des données*. C'est la distinction la plus importante qui existe entre les systèmes de fichiers et les systèmes de bases de données.

La seconde séparation entre les niveaux physique et externe est aussi importante pour l'utilisateur puisqu'elle permet des points de vue différent sur les bases de données qui sont adaptés à des besoins spécifiques. Les vues cachent ainsi les informations sans intérêt et restructure les données qu'elles retiennent. De telles vues peuvent être simples comme les distributeurs automatiques ou plus sophistiquées comme le cas de systèmes de conception assistés par ordinateur.

Un problème important en relation avec les deux séparation dans l'architecture d'un SGBD est le compromis à trouver entre le confort humain et une performance raisonnable. La séparation entre les aspects logique et physique signifie par exemple que le système doit compiler les requêtes et les mises-à-jour destinées à la représentation logique dans les programmes « réels ». Ainsi, le modèle relationnel devint largement répandus lorsque des techniques d'optimisation de requêtes le rendirent faisable. Plus généralement, plus la discipline de l'optimisation des bases de données arrivait à maturité, plus les modèles logiques devenaient éloignés des moyens de stockages matériels. Les développements matériels (mémoire plus importante, plus rapide) ont également grandement influencé le domaine en changeant continuellement les limites des possibilités.

1.2 Fonctionnalités

Les SGBD incluent de nombreuses fonctionnalités, allant des fonctionnalités des plus physiques aux plus abstraites. Les fonctions principales d'un SGBD sont les suivantes :

Manipulation de la mémoire secondaire : Le but d'un SGBD est la manipulation d'une grande quantité de données partagées. Par grande, nous entendons qu'elles ne peuvent pas tenir en mémoire principale. Ainsi, une des tâches essentielle de ces systèmes est la ma-

nipulation des mémoires secondaires, ceci passe par de nombreuses techniques telles que l'indexation, le regroupement et l'allocation de ressources.

Persistance : Les données doivent survivre à la fin d'une application particulière sur la base de données pour qu'elles puissent être réutilisées plus tard.

Contrôle de concurrence : Les données sont partagées. Le SGBD doit ainsi permettre l'accès simultané aux informations partagées dans un environnement harmonieux qui contrôle les conflits et qui fournit un état cohérent de la base de données à chacun des utilisateurs. Cela a conduit à d'importantes notions telles que celles de *transaction* et de *séquentiabilité* (en anglais, serializibility) et à des techniques de verrouillage.

Protection des données : La base de données doit être protégée contre les erreurs humaines, contres les erreurs de programmes, et contre les défaillance de l'ordinateur. Des mécanismes de vérification de l'intégrité portent leur attention sur la prévention des incohérences entre les données stockées (par exemple pour les m.a.j.). La reprise sur panne et les protocoles de sauvegarde prémunissent contre les défaillances matérielles en conservant des instantanés d'états antérieurs de la base de données et des journaux de bords des transaction en train de se dérouler. Enfin, les mécanismes de contrôle de sécurité prémunissent contre l'accès et/ou le changement d'informations sensibles à l'encontre de certaines catégories d'utilisateurs.

Interface homme-machine : Cette interface concerne une grande variété de fonctions tournant autour de la représentation logique des données. Plus concrètement, cette interface concerne des LDD et LMD. Les outils graphiques pour l'installation et la conception de bases de données sont très populaires.

Distribution : Dans beaucoup d'applications, les informations résident dans des lieux distincts, réparties en plusieurs bases de données. Ces bases de données peuvent être accessibles par différents systèmes (*interopérabilité*) et elles peuvent être fondées sur des modèles distincts (*hétérogénéité*). Il est donc important de donner un accès transparent à des systèmes multiples.

Compilation et optimisation : Une tâche importante est la traduction des requêtes de niveaux logiques et externes en programmes exécutables. Cela nécessite en général une ou plusieurs étapes de compilation et une optimisation intensive de façon à ce que les performances ne soient pas dégradés par la commodité d'interfaces utilisateurs plus avenantes.

1.3 Complexité et diversité

Outre le fait d'avoir à faire face à plusieurs fonctions, le champ des bases de données doit être conçu pour une grande diversité d'utilisations, de styles et plate-formes physiques. Voici

quelques exemples de cette diversité :

Applications : finances, personnel, inventaire, ventes, dossiers, biologie, etc.

Utilisateurs : programmeurs d'applications, sav, secrétaires, administrateurs de bases de données, geek, etc.

Modes d'accès : LMD linéaires et graphiques, interfaces graphiques, entrée des données, génération de rapports, etc.

Modèles logiques : les plus diffusées sont les modèles en réseau, hiérarchique, relationnel, orienté objet. Il existe des variations pour chacun de ces modèles, implémentées par divers distributeurs.

1.4 Modèles de données

Tout SGBD est conçu autour d'un *modèle de données* bien défini. Comme dit précédemment, il est constitué d'un LDD et d'un LMD. Plus précisément, il s'agit de la combinaisons de 3 éléments :

- Une *structure*, qui correspond à l'organisation logique des données, la forme sous laquelle les utilisateurs vont les percevoir ou les représenter.
- Des *contraintes d'intégrité*, que l'on peut définir sur les données, afin d'en assurer l'intégrité et la cohérence avec le monde réel et les besoins des applications.
- Des langage de *manipulation des données*, pour les mises à jour et les interrogations.

Voici quelques exemples de modèles de données :

- *le modèle réseau*
  - Structure : graphe orienté, les noeuds sont des enregistrements
  - Contraintes d'intégrité : un identifiant pour chaque noeud, un mécanisme de référence entre des noeuds
  - Manipulation : parcours de réseaux.
- le modèle hiérarchique
  - Structure : arborescente (forêts)
  - Contraintes d'intégrité : un identifiant pour chaque noeud, un mécanisme de référence entre des noeuds
  - Manipulation : navigation hiérarchique
- le modèle relationnel
  - Structure : ensembles de relations, qui sont des ensembles de tuples. Graphiquement, une relation est un tableau.
  - Contraintes d'intégrité : principalement les clés primaires (identifiants de tuples) et les clés étrangères (liens entre les relations)
  - Manipulation : algèbre relationnel, calcul relationnel, SQL.

- le modèle déductif
  - Structure : quasiment la même que le modèle relationnel
  - Contraintes d'intégrité : les mêmes que le modèle relationnel
  - Manipulation : langages logiques, le principal est *Datalog*. Contrairement aux langages du modèle relationnel, il admet la récursivité.
- le modèle Objet
  - Structure : logique objet, soit des classes, des objets, des attributs et des méthodes. Peut être vu comme un graphe orienté.
  - Contraintes d'intégrité : identifiant pour les objets, référence entre objets.
  - Manipulation : extensions de SQL comme OSQL ou OQL.
- le modèle Entité/Association
  - Structure : Entités (avec des attributs) et associations entre des entités.
  - Contraintes d'intégrité : identifiants, cardinalités sur les associations
  - Manipulation : aucun.

Ajoutons les modèles semi-structurés, qui s'adaptent à la nature hétérogène des données, principalement trouvées sur le Web. Le principal exemple est XML. En pratique, le relationnel est, de loin, le plus utilisé de tous les modèles de données. Quelques SGBD objet existent, mais rencontrent des problèmes de performances pour les gros volumes de données, dus à l'exécution en cascade de méthodes lors des parcours de données. Des SGBD XML ont aussi plus récemment vu le jour, et sont utilisés, mais n'égalent pas les performances du modèle relationnel.

Les principaux SGBD relationnels intègrent de plus en plus des extentions permettant de "percevoir" les données sous la logique objet ou XML, mais utilisant derrière des bases de données relationnelles pour stocker les données.

Chapitre 2

Le modèle relationnel

Sommaire

Ce chapitre décrit *la structure* et *les langages* du modèle relationnel. L'objectif est de présenter le contexte et les notations. La partie sur les *contraintes* fera l'objet du chapitre suivant.

2.1 Introduction

Un modèle de base de données fournit les moyens de définir des structures de données particulières, de contraindre l'ensemble des données associées à ces structures et d'en manipuler les données. La définition de structure et de contraintes est faite en utilisant un LDD et la définition des manipulations en utilisant un LMD.

Le modèle relationnel a été défini en 1970 par Codd. Cette proposition a succédé aux modèles hiérarchiques et réseaux, avec pour but de définir un modèle simple à comprendre, fondé sur des bases mathématiques solides (logique et théorie des ensembles), et pouvant conduire à une amélioration des performances des outils. L'idée de base est simple : mettre les données « à plat » dans des tableaux, de façon à ce que chaque valeur apparaissant dans les « cases » du tableau soient atomiques. Cette simplification est la principale explication de la performance

du modèle relationnel, car elle évite les traitements récursifs (parcours de graphes et d'arbres complexes).

Après une dizaine d'année de développement de la théorie inhérente aux propositions de Codd, les premiers SGBD commerciaux font leur apparition dans les années 80, avec des outils comme ORACLE principalement. Les contributions techniques se combinent alors aux fondements théoriques et le modèle relationnel s'impose comme un standard. Dès le début des années 90, les progrès technologiques fulgurants sur le matériel (processeurs et mémoire disque) conduisent à des bases de plus en plus grosses, pour atteindre des proportions gigantesques de nos jours. La recherche dans le domaine des bases de données relationnelles est toujours active, placée sans cesse face à des nouveaux défis de taille, d'exigence de rapidité, d'hétérogénéité des données. Même si de nouveaux paradigmes pourraient être en passe de voir le jour dans le traitement des bases de données de production, les fondements théoriques présentés dans ce cours constituent toujours la base des techniques utilisées.

2.2 La structure du modèle relationnel

La structure d'une BD relationnelle est fondée sur la théorie des ensembles. Rappelons qu'un ensemble est une collection *non ordonnée* d'éléments *distincts*, à différencier de la *séquence* (lorsque les éléments sont ordonnés) et du *multi-ensemble* (lorsqu'on accepte les doublons).

Notations les notations suivantes sont communément admises lorsque le contexte le permet. Un ensemble $\{A_1; A_2; ...; A_n\}$ sera noté par la concaténation de ses éléments : $A_1A_2...A_n$. Si X et Y sont deux ensemble, leur union $X \cup Y$ sera notée XY. Une séquence d'éléments, est notée $< A_1, ..., A_n >$. Lorsque le contexte est clair, les séquences et les ensembles seront notés de la même façon.

Exemple 1. *L'ensemble $\{A; B; C\}$ sera noté ABC dans ce cours. Les notations ABC et BCA sont équivalentes puisque l'ordre n'a pas d'importance : par convention, on notera les ensembles théoriques en respectant l'ordre alphabétique. L'ensemble $AABC$ n'existe pas, puisque l'élément A apparaît deux fois. Soit les ensemble $X = ABC$ et $Y = BD$, alors leur union $X \cup Y$ sera noté $XY = ABCD$.*

Un exemple de bases de données relationnelles est reporté dans le figure 2.1. Intuitivement les données sont représentées sous forme de tableaux dans lesquels chaque ligne contient des données sur un objet ou un ensemble d'objets particuliers ; les lignes avec une structure uniforme et une signification attendue sont regroupées en tableaux. Les m.a.j. consistent à transformer les tableaux en ajoutant, en retranchant ou en modifiant des lignes. Les requêtes permettent l'extraction d'informations à partir de ces tableaux. Une caractéristique fondamentale de presque tous les langages de requêtes relationnels est que le résultat d'une requête est également un tableau ou une collection de tableaux.

Étudiants	NUM	NOM	$PRENOM$	AGE	$FORMATION$
	28	Codd	Edgar	20	3
	32	Armstrong	William	20	4
	53	Fagin	Ronald	19	3
	107	Bunneman	Peter	18	3

Enseignants	NUM	NOM	$PRENOM$	$GRADE$
	5050	Tarjan	Robert	Pr.
	2123	De Marchi	Fabien	MCF
	3434	Papadimitriou	Spiros	Pr.
	1470	Bagan	Guillaume	CR

TABLE 2.1 – Exemple de bases de données

Introduisons maintenant un peu de terminologie de façon informelle pour former l'intuition se trouvant derrière les définitions formelles qui suivent. Chaque tableau est appelé une *relation* et possède un *nom* (par exemple, *Étudiants*). Les colonnes possèdent également un nom appelé *attribut* (par exemple *Nom*). Chaque ligne d'un tableau est un *n-uplet* (ou enregistrement, en anglais tuple). Les coordonnées d'un n-uplet appartiennent à des ensembles de constantes appelés *domaines*, par exemple l'ensemble des entiers, des mots, ou des valeurs booléennes. Enfin, nous ferons une distinction entre le *schéma* de la base de données, qui définit la structure de la BD et l'*instance* de base de données, qui définit son contenu réel.

Venons-en maintenant aux définitions formelles.

2.2.1 Attributs, valeurs et domaines

Soit $\mathcal{U}$, un ensemble infini dénombrable de *noms d'attributs* ou simplement *attributs*, et $\mathcal{D}$ un ensemble infini dénombrable de *valeurs*. Soit $A \in \mathcal{U}$ un *attribut*, le *domaine* de A est un sous-ensemble de $\mathcal{D}$, noté $DOM(A)$.

Hypothèse d'unicité des noms : Soient deux valeurs v_1 et v_2 des éléments de $\mathcal{D}$. On dit que v_1 et v_2 sont égales si et seulement si elles sont syntaxiquement identiques, c'est à dire qu'elles ont le même nom. En d'autres termes, lorsque deux attributs ont la même valeur, ils désignent la même chose.

2.2.2 Schémas de relations, schémas de bases de données et première forme normale

Définition 1. *schémas de relations et de bases de données Un* schéma de relation R *est un*

ensemble fini d'attributs (donc $R \subseteq \mathcal{U}$). Un schéma de base de données $\mathbf{R}$ *est un ensemble fini de schémas de relation.*

Exemple 2. *L'exemple suivant sera utilisé tout au long de ce cours. On suppose une application gérant les cours dispensés au sein de l'UFR, regroupant des informations sur les étudiants, les formations, les enseignants, les modules, les UE, les salles de cours et les ressources matérielles (projecteurs, etc...).*

La modélisation des étudiants pourrait se faire à l'aide du schéma de relation

$$ETUDIANTS = \{NUM; NOM; PRENOM; AGE; FORMATION\}.$$

Si on représente les autres informations dans d'autres schémas de relations, alors l'union de tous les schémas de relation obtenus constituera le schéma de notre base de données, qu'on pourra appeler $FACULTE$.

Hypothèse de schéma de relation universel : Un schéma de base de données $\mathbf{R}$ satisfait l'hypothèse de schéma de relation universelle si, lorsque deux attributs portent le même nom, ils désignent le même concept.

Définition 2. *Première forme normale Un schéma de relation R est en première forme normale (notée $1FN$) si, pour tout attribut $A \in R$, $DOM(A)$ est composé de valeurs atomiques.*

La première forme normale est une condition fondamentale du modèle relationnel, garante de sa simplicité et efficacité, et l'hypothèse de schéma de relation universelle est toujours vérifiés, permettant de désigner de façon unique un attribut, sans aucune ambiguïté.

Remarque 1. *Ces restrictions sont nécessaires à assurer les fondements théoriques justes du modèle relationnel. En pratique, on trouve plus de souplesse dans les outils :*

- *Les types complexes sont autorisés dans plusieurs SGBD pour peupler les relations : il s'agit en fait d'extensions de la théorie de base, en utilisant une logique objet. Le modèle utilisé est alors appelé relationnel-objet (Exemple : Oracle). Il permet une modélisation plus intuitive pour les concepteurs, mais le modèle utilisé en fond par le SGBD pour stocker ses données est toujours le relationnel.*
- *Dans tous les outils existants, on peut nommer deux fois le même attribut pour représenter des concepts différents (au sein d'une relation, un attribut n'apparaît toutefois qu'une seule fois. Par exemple, dans notre base de données on pourrait imaginer un attribut NOM qui désigne le nom d'une salle de cours, et le même attribut qui désigne le nom d'une personne ; ce serait une faute de conception, mais impossible à vérifier par les outils !*

Un schéma de relation, ou de bases de données, est donc une boîte vide, avec une structure bien particulière, destinée à contenir des ensembles d'éléments possédant la même structure et donc sémantiquement équivalents.

2.2.3 Tuples, relations et bases de données

Définition 3. *tuple, Relation et Base de Données Soit $R = A_1...A_n$ un schéma de relation. Un tuple sur R est un membre du produit cartésien $DOM(A_1) \times \ldots \times DOM(A_n)$. Une relation r sur R est un ensemble fini de tuples. Une base de données* **d** *sur un schéma de base de données* $\mathbf{R} = \{R_1, ..., R_n\}$ *est un ensemble de relations $\{r_1, ..., r_n\}$ définies sur les schéma de relation de* **R**.

De façon informelle, on peut donc voir un tuple comme une séquence de valeurs, prises par les attributs du schéma de relation. Si t est un tuple défini sur un schéma de relation R, et X un sous-ensemble de R, on peut restreindre t à X en utilisant l'opération de projection, notée $t[X]$.

Exemple 3. *Prenons la relation* Étudiants *illustrée dans la table 2.1. Si on nomme t_1 le premier tuple, alors $t_1[NOM] = Codd$, $t_1[NUM, AGE] = (28, 20)$. L'ensemble de toutes les relations "peuplées" par des tuples constitue la base de données.*

2.3 Langages de requêtes et de mises à jour

Plusieurs langages ont été définis pour l'interrogation et la manipulation des données dans le modèle relationnel. *Le résultat d'une requête est toujours une relation* : les langages diffèrent sur la façon de définir la relation en sortie. Ils sont de deux sortes :

- *les langages procéduraux* : cela signifie que, pour obtenir les réponses à sa requête, il faut déterminer où et comment aller rechercher l'information parmi les relations de la base. On définit la procédure devant être exécutée par le programme. Le langage procédural du modèle relationnel est l'algèbre relationnel. Il s'agit d'un ensemble d'opérateurs algébriques unaires et binaires applicables à des relations et retournant des relations : ils peuvent donc être composés pour exprimer le résultat souhaité.
- *déclaratif* : pour obtenir les réponses à sa requête, il faut caractériser précisément le résultat que l'on veut, sans se soucier de la méthode utilisée par le programme pour accéder aux données. C'est grâce à des expressions logiques qu'on détermine le résultat souhaité : c'est le cas du calcul relationnel de tuple (qui manipule des tuples dans des expressions logiques) ou de domaine (qui manipule des variables parcourant les domaines), ou encore du DATALOG (admettant la récursivité).

Le SQL correspond à l'implantation du calcul relationnel de tuple : c'est donc un langage déclaratif, possédant de nombreuses extensions pour faciliter l'usage et augmenter le pouvoir d'expression.

2.4 Notations

Nous utiliserons en général les symboles suivants, éventuellement avec des indices.

Constantes	a, b, c
Variables	x, y
Ensembles de variables	X, Y
Attributs	A, B, C
Ensembles d'attributs	U, V, W
(Schémas de) noms de relation	R, S ; $R[U], S[V]$
Schémas de bases de données	**R**,**S**
Tuples, Uplets	t, s
Uplets libres	u, v, w
Faits	$R(a_1, \ldots, a_n)$, $R(t)$
Instances de relation	I, J
Instance de bases de données	**r**

Chapitre 3

Contraintes d'intégrité dans le modèle relationnel

Sommaire

3.1 Généralités

La structure du modèle relationnel défini dans le chapitre précédent permet de modéliser la réalité à un certain niveau de granularité : les attributs vont décrire les objets, et on peut séparer les données dans différentes relations avec des noms explicites. Mais ceci est largement insuffisant pour représenter plus finement les différents aspects des données réelles. L'incapacité à représenter des *méta-données* conduit sans conteste à un certain nombre de problèmes.

Exemple 4. *Supposons le schéma de relation $ETUDIANT$ décrit plus haut. Rien n'empêche l'utilisateur :*

- *d'insérer plusieurs étudiants avec le même numéro mais des données différentes ;*
- *d'insérer deux fois le même étudiant mais avec deux âges différents ;*
- *de mettre un numéro farfelu dans la colonne $FORMATION$, ne référant aucune formation existante.*
- *As-t-on le droit d'insérer un étudiant dans plusieurs formations ? Il faut un mécanisme pour pouvoir le préciser.*

Rappel : en revanche, il est impossible d'avoir deux fois exactement le même tuple dans une relation, puisque une relation est un ensemble (donc sans doublons) de tuples.

En réponse à ces problèmes, un cadre a été développé pour ajouter une sémantique au modèle relationnel. On incorpore des *contraintes d'intégrité*, c'est à dire des propriétés supposées être satisfaites par toutes les instances d'un schéma de bases de données. Un exemple simple est celui des *dépendances de clé*. Nous nous attendons par exemple à ce que, dans une relation concernant une donnée sur une personne, le numéro de sécurité sociale serve de « clé » (c'est-à-dire que le numéro de sécurité sociale identifie de façon unique un tuple de cette relation). Le but des contraintes d'intégrité est de pouvoir interdire tous les cas d'incohérence pouvant être générés lors des mises à jour, pour avoir des données collant toujours à la "logique" de la réalité. En effet, les applications sont souvent développées en utilisant certains postulats sur les données (exemple : le numéro d'étudiant est unique) - mais si ces postulats ne sont pas vérifiés, les applications vont générer des erreurs parfois imprévisibles.

On distingue plusieurs types de contraintes :

- Contraintes statiques : sont vérifiées dans un état donné de la base
  - les dépendances de données
  - les dépendances de domaine
  - les contraintes de cardinalité
- Contraintes dynamiques : vérifiées sur deux états différents de la base

Exemple 5.

- *Chaque étudiant a un numéro unique : cette contrainte est statique, puisse à tout moment, il suffit d'observer la base pour savoir si elle est bien respectée.*
- *Un étudiant ne peux pas avoir un âge qui décroît : contrainte dynamique, car ne peut être vérifiée que dans deux états successifs de la base.*

Les contraintes les plus importantes en conception des bases de données sont les dépendances de données.

Films	*Titre*	*Metteur*	*Acteur*
	Les oiseaux	Hitchcock	Hedren
	Les oiseaux	Hitchcock	Taylor
	Bladerunner	Scott	Hannah
	Apocalypse Now	Coppola	Brando

Programme	*Ciné*	*Salle*	*Titre*	*Friandise*
	Rex	1	Les oiseaux	café
	Rex	1	Les oiseaux	popcorn
	Rex	2	Bladerunner	café
	Rex	2	Bladerunner	popcorn
	ArtC	1	Les oiseaux	thé
	ArtC	1	Les oiseaux	popcorn
	Cinoche	1	Les oiseaux	Coca Cola
	Cinoche	1	Les oiseaux	vin
	Cinoche	2	Bladerunner	Coca Cola
	Cinoche	2	Bladerunner	vin

TABLE 3.1 – Exemple de base de données illustrant des dépendances simples

3.2 Motivations et principales dépendances

Considérons la BD représentées dans la table 3.1. Bien que le schéma lui-même ne fasse aucune restriction sur les données qui peuvent être stockées, l'application attendue de ce schéma peut impliquer plusieurs de ces restrictions. Nous pouvons, par exemple, savoir qu'il y a un seul metteur en scène associé à chaque titre de film et que, dans *Programme*, un seul titre de film est associé à un couple cinéma-salle donné. De telles propriétés sont appelées des *dépendances fonctionnelles* (df) puisque les valeurs de certains des attributs d'un tuple déterminent de façon unique, ou *fonctionnellement*, les valeurs des autres attributs du tuple. Dans la syntaxe que nous présentons, la dépendance dans la relation *Films* s'écrit :

$$Films : \ Titre \rightarrow Metteur$$

et celle de la relation *Programme* :

$$Programme : \ Ciné \ Salle \rightarrow Titre.$$

Formellement, il existe des ensembles d'attributs de chaque côté de la flèche, mais nous continuerons à omettre les accolades ensemblistes lorsque le contexte permet de comprendre.

Lorsque le contexte ne prête pas à confusion, une dépendance $R : \ X \rightarrow Y$ sera tout simplement notée $X \rightarrow Y$.

Une relation I **satisfait** la dépendance fonctionnelle $X \rightarrow Y$ ssi pour tout couple s, t de tuples de I on a :

$$\pi_X(s) = \pi_X(t) \Rightarrow \pi_Y(s) = \pi_Y(t).$$

Une des notions importantes de la théorie des dépendances est celle d'implication. On peut observer que toute relation qui satisfait la dépendance :

$$Titre \rightarrow Metteur$$

satisfait également la dépendance :

$$Titre,\ Acteur \rightarrow Metteur.$$

On dit que la première df implique la seconde.

Une *dépendance de clé* est une dépendance fonctionnelle $X \rightarrow U$, où U est l'ensemble de tous les attributs de la relation. X est appelée *clé*. Par exemple,

$$Titre,\ Acteur \rightarrow Titre,\ Acteur,\ Metteur$$

est une dépendance de clé.

Un second type fondamental de dépendances est illustré par la relation *Programme*. Un tuple (th, sc, ti, sn) est dans *Programme* si le cinéma th passe le film ti dans la salle sc et si le cinéma th propose la friandise sn. Intuitivement, on pourra espérer une certaine indépendance entre les attributs $Salle,\ Titre$ d'une part et l'attribut $Friandise$, d'autre part, pour une valeur donnée de $Ciné$. Par exemple, puisque *(Cinoche, 1, Les oiseaux, Coca Cola)* et *(Cinoche, 2, Bladerunner, vin)* sont dans *Programme*, nous pourrions nous attendre à ce que *(Cinoche, 1, Les oiseaux, vin)* et *(Cinoche, 2, Bladerunner, Coca Cola)* soient également présents. Plus précisément si une relation$I$ possède cette propriété, alors on a :

$$I = \pi_{Ciné,Salle,Titre}(I) \bowtie \pi_{Ciné,Friandise}(I).$$

C'est donc un exemple simple de dépendances de jointure (dj), exprimée formellement par :

$$Programme : \bowtie [\{Ciné, Salle, Titre\}, \{Ciné, Friandise\}].$$

D'une façon générale, une dj peut faire intervenir plus de deux ensembles d'attributs. Une *dépendance multivaluée (dmv)* est un cas particulier de dj qui possède au plus deux ensembles d'attributs. Du fait de leur caractère naturel, les dmv ont été introduites avant les dj ; elles possèdent, de plus, plusieurs propriétés intéressantes, aussi valent-elles la peine d'être étudiées plus spécifiquement.

Le fait que la df $Titre \rightarrow Metteur$ soit satisfaite par la relation *Films* implique que la dj $\bowtie [\{Titre, Metteur\}, \{Titre, Acteur\}]$ est également satisfaite. Il existe de nombreuses interactions entre les df et les dj, nous y reviendront dans la suite.

Jusqu'à maintenant, nous avons considéré les dépendances s'appliquant à des relations individuelles. Ces dépendances sont habituellement utilisées dans le contexte d'un schéma de bases de données, cas dans lequel doit être spécifiée la relation concernée par chaque dépendance. Nous considérerons un troisième type de dépendances, appelées *dépendances d'inclusion (di)* ou « contraintes référentielles ». Dans notre exemple, nous pouvons nous attendre à ce que tout titre projeté actuellement (c'est-à-dire présent dans la relation *Programme*) soit le titre d'un film (c'est-à-dire apparaissant dans la relation *Films*). Cela est noté :

$$Programme[Titre] \subseteq Films[Titre].$$

En général, les di peuvent faire intervenir des *séquences* d'attributs de chaque côté.

Les dépendances de données telles que nous venons de les présenter donnent un mécanisme formel pour exprimer des propriétés attendues des données stockées. Si on sait que la BD satisfait un ensemble de dépendances, alors cette information peut être utilisée :

1. pour améliorer la conception d'un schéma ;
2. pour protéger les données en se prémunissant contre certaines mise à jour erronées ;
3. pour améliorer les performances.

3.2.1 Conception du schéma et anomalies de mise à jour

La tâche consistant à concevoir le schéma dans le cas d'un BD concrète et importante est loin d'être aisée, aussi le concepteur doit-il recevoir une aide du système. Les dépendances sont utilisées pour fournir des informations sur la sémantique de l'application afin que le système puisse aider l'utilisateur à choisir, parmi tous les schémas possibles, le plus approprié. Il existe plusieurs façons, pour un schéma, de ne pas être approprié. Les relations *Films* et *Programme* illustrent les types les plus importants relatifs aux df et dj.

Informations incomplètes : Supposons qu'il faille insérer le titre d'un nouveau film et son metteur en scène sans encore connaître les acteurs. C'est impossible avec le schéma décrit précédemment et il y a une *anomalie d'insertion*. Un phénomène analogue se produit pour l'élimination ; une *anomalie d'élimination* apparaît si l'acteur Marlon Brando n'est plus associé au film Apocalypse Now. Ainsi le triplet *(Apocalypse Now, Coppola, Brando)* doit-il être supprimé de la base de données. Mais cela a également pour effet supplémentaire d'éliminer l'association entre le film (Apocalypse Now) et le metteur en scène (Coppola) dans la BD, information qui peut être encore valide.

Redondance : Le fait que *Coca Cola* puisse être trouvé au *Cinoche* est enregistré plusieurs fois. Supposons de plus que le directeur du *Cinoche* décide de vendre du *Pepsi Cola* au lieu du *Coca Cola*. Il n'est pas suffisant de transformer le quadruplet *(Cinoche, 1, Les oiseaux, Coca Cola)* en tuple *(Cinoche, 1, Les oiseaux, Pepsi Cola)* puisque cela conduirait à la violation de la dj. Nous devons donc modifier plusieurs tuples. Il s'agit d'une *anomalie de modification*. Des anomalies d'insertion et de suppression sont également causées par la redondance.

Ainsi à cause d'un mauvais choix de schéma, les m.a.j peuvent conduire à une perte d'informations, à une incohérence des données et à une écriture correcte plus difficile des m.a.j. On peut se prémunir de ces problèmes en optant pour un schéma plus approprié. Dans notre exemple, la relation *Films* pourrait être « décomposée » en deux relations *M-Metteur[Titre, Metteur] et M-Acteur[Titre, Acteur]* où *M-Metteur* satisferait la df $Titre \rightarrow Metteur$. De même, la relation *Programme* pourrait être remplacée par les deux relations *ST-Programme[Ciné, Salle, Titre]* et *S-Programme[Ciné, Friandise]*, où *ST-Programme* satisferait la df $Ciné, Salle \rightarrow Titre$. La conception de tels schémas sera étudiée dans la suite du cours.

3.2.2 Intégrité des données

Les dépendances des données servent également de filtres aux m.a.j proposées d'une façon naturelle : si on s'attend à ce qu'une BD satisfasse une dépendance σ et qu'une proposition de m.a.j conduise à une violation de σ, alors la m.a.j est rejetée. En fait, le système supporte les transactions. Durant une transaction, la BD peut être dans un état inconsistant mais, à la fin de celle-ci, le système doit vérifier l'intégrité de la BD. Si les dépendances sont violées, alors la transaction entière est rejetée, sinon elle est acceptée (validée).

3.2.3 Implémentation efficace et optimisation de requêtes

On s'attend naturellement à ce que la connaissance des propriétés structurelles des données stockées soit utile à l'amélioration des performances d'un système pour une application particulière.

Au niveau physique, la satisfaction des dépendances conduit à une grande variété d'alternatives pour les structures de stockage et d'accès. Par exemple, la satisfaction d'une df ou d'une dj implique qu'une relation peut être stockée physiquement sous forme décomposée. De plus, la satisfaction d'une dépendance de clé peut être utilisée pour réduire l'espace d'indexation.

Un développement théorique particulièrement frappant de la théorie des dépendances a fourni une méthode qui optimise les requêtes conjonctives en présence d'une grande classe de dépendances. Comme exemple simple, considérons la requête :

$$ans(d, a) \leftarrow Films(t, d, a'), Films(t, d', a),$$

qui renvoie le couple (d, a), où un acteur a joue dans un film mis en scène par d. Une implémentation naïve de cette requête exige une jointure. Puisque *Films* satisfait $Titre \rightarrow Metteur$, cette requête peut être simplifiée en :

$$ans(d, a) \leftarrow Films(t, d, a),$$

qui peut être évaluée sans jointure. Lorsque $\{(t, d, a'), (t, d', a)\}$ est trouvé dans la relation *Films*, on doit avoir $d = d'$, aussi peut-on aussi bien n'utiliser que $\{(t, d, a)\}$, ce qui conduit à la requête simplifiée.

3.2.4 Suite

Les dépendances que nous avons cité jusqu'à présent sont :
- Les dépendances fonctionnelles
- Les dépendances d'inclusion
- Les dépendances de jointure
- Les dépendances multi-valuées

Les deux premières permettent de représenter les situations les plus communes, et sont largement les plus connues, étudiées et utilisées en pratique. Nous allons nous restreindre à elles dans un premier temps - les dépendances de jointure et les dépendances multi-valuées seront présentées plus loin.

3.3 Dépendances fonctionnelles et clés

Les dépendances fonctionnelles constituent la forme la plus fréquemment rencontrée de dépendances. Les dépendances de clé sont un cas particulier de dépendances fonctionnelles. La plupart des questions théoriques sur les dépendances ont de belles solutions dans le contexte des dépendances fonctionnelles. Celles-ci sont à l'origine de l'approche par décomposition des schémas.

Pour définir une classe de dépendances, on doit définir la syntaxe et la sémantique des dépendances concernées.
- La syntaxe : c'est le langage logique autorisé pour définir une contrainte. C'est la "*forme*" de la contrainte.
- La sémantique : ensemble de conditions devant être remplies pour que la contrainte soit *satisfaite* (c'est à dire juste) dans les données. C'est le *sens* de la contrainte.

La syntaxe des dépendances fonctionnelles est définie ainsi :

Définition 4. *Dépendance fonctionnelle (DF) Une* dépendance fonctionnelle (DF) *sur un schéma de relation R est une expression de la forme $R : X \rightarrow Y$ (ou simplement $X \rightarrow Y$ lorsque R est implicite), où $X, Y \subseteq R$. Une DF $X \rightarrow Y$ est dite* triviale *si $Y \subseteq X$ et* standard *si $X \neq \varnothing$.*

Une DF est donc une contrainte définie sur un schéma. Une DF est dite *valide* sur un schéma R si elle est satisfaite dans toute relation r sur R, selon de la définition suivante pour la satisfaction :

Définition 5. *Satisfaction d'une DF dans une relation Soit r une relation sur R. Une DF $R : X \rightarrow Y$ est* satisfaite *dans r si $\forall t_1, t_2 \in r$ on a $t_1[X] = t_2[X] \Longrightarrow t_1[Y] = t_2[Y]$.*

Si $X \rightarrow Y$ est satisfaite dans une relation r, on dit aussi que X *détermine* Y dans r. De façon intuitive, on définit une DF $X \rightarrow Y$ pour exprimer le fait que lorsqu'on connaît la valeur de X, alors on peut déterminer (en parcourant la relation) de façon certaine la valeur de Y.

La notion de *clé d'une relation* est très connue par les utilisateurs du modèle relationnel. Une clé peut-être définie de deux manières :

- une clé est un ensemble d'attributs qui ne prend jamais deux fois la même valeur (pas de doublons dans les relations) ;
- A l'aide des DF : une clé est un ensemble d'attributs qui détermine (au sens des DF) tous les autres attributs de la relation.

Ces deux définitions sont rigoureusement équivalentes.

Démonstration. Soit X un ensemble d'attributs sur un schéma R sur lequel on interdit les doublons. Ainsi, la DF $X \rightarrow R - X$ ne peut être ""violée"" dans aucune relation possible sur R, puisqu'il est impossible d'avoir deux tuples qui ont la même valeur sur X. Cela implique donc que la contraite $X \rightarrow R - X$ est valide sur R.

Inversement, supposons que la DF $X \rightarrow R - X$ est valide sur R. Supposons une relation r sur R dans laquelle deux tuples t_1 et t_2 soient tels que $t_1[X] = t_2[X]$. Dans ce cas, on a également $t_1[R - X] = t_2[R - X]$ par application de la DF, et par conséquence $t_1 = t_2$ - ce qui est impossible puisque r est un ensemble de tuples (donc sans doublon). Donc quelque soit la relation r sur R, il n'y a pas de doublon sur X. □

Une clé primaire est simplement une clé parmi les autres, choisie par le concepteur pour sa simplicité ou son aspect naturel.

3.4 Dépendances d'inclusion

Les DI se différencient des DF sur plusieurs points. D'abord, elles peuvent être définies entre attributs de relations différentes, et possèdent ainsi un caractère plus "global" que les DF dont la définition est locale à une relation (bien que l'on parle parfois de *DF globale*, que nous n'abordons pas ici). Ensuite, les DI sont définies non pas entre deux ensembles quelconques d'attributs, mais *entre deux séquences d'attributs de même taille*. Cette fois, l'ordre des attributs est donc important.

La syntaxe (forme) des DI est la suivante.

Définition 6. *Dépendance d'inclusion Soit* $\mathbf{R}$ *un schéma de base de données. Une dépendance d'inclusion sur* $\mathbf{R}$ *est une expression de la forme* $R[X] \subseteq S[Y]$*, où* $R, S \in \mathbf{R}$*,* X *et* Y *sont des séquences d'attributs distincts respectivement de* R *et de* S*, et* $|X| = |Y|$*.*

Pour ce qui est de la sémantique des DI, l'intuition est la suivante : une DI est satisfaite dans une base de données si toutes les valeurs prises par la partie gauche apparaissent dans la partie droite.

Définition 7. *Satisfaction d'une DI Soit* $d = \{r_1, r_2, \ldots, r_n\}$ *une base de données sur un schéma* $\mathbf{R} = \{R_1, \ldots, R_n\}$*. Une dépendance d'inclusion* $R_i[X] \subseteq R_j[Y]$ *sur* $\mathbf{R}$ *est* satisfaite *dans* d*, noté* $d \models R_i[X] \subseteq R_j[Y]$*, si* $\forall t_i \in r_i, \exists t_j \in r_j \mid t_i[X] = t_j[Y]$ *(de manière équivalente,* $\pi_X(r_i) \subseteq \pi_Y(r_j)$*).*

Exemple 6. *Supposons des schémas de relation pour décrire les modules :*

$$MODULE = \{\underline{NUMMODULE}; INTITULE; DESC\}$$

et un schéma de relation pour décrire les séances de cours :

$$SEANCE = \{\underline{DATE; NUMMODULE}; NUMSALLE\}$$

Pour forcer que les numéros de modules dans les séances soient bien des modules qui existent, on devra alors définir la contrainte :

$$SEANCE[NUMMODULE] \subseteq MODULE[NUMMODULE]$$

Supposons maintenant un schéma de relation EMPRUNTVIDEO *modélisant les réservations de vidéos pour les séances, sous la forme :*

$$EMPRUNTVIDEO = \{DATE, NUMMODULE; NUMPROF; NUMVIDEO\}$$

Pour assurer la cohérence de la base, on doit préciser que les vidéos doivent être réservés pour des séances existantes dans la base ; on définira alors la DI :

$$EMPRUNTVIDEO[DATE, NUMMODULE] \subseteq SEANCE[DATE, NUMMODULE]$$

On peut remarque l'importance de l'ordre dans les attributs (ce sont des séquences et non des ensembles) : si on inverse les attributs à gauche par exemple, la DI change complètement de signification !

Une *contrainte d'intégrité référentielle* est une DI dont la partie droite est une clé ; la partie gauche d'une contrainte d'intégrité référentielle est appelée *clé étrangère*.

Exemple 7. *Les deux DI données dans l'exemple précédent sont des contraintes d'intégrité référentielle, puisqu'on peut supposer que les parties droites sont des clés de leur relation. Les parties gauches sont donc des clés étrangères des relations* $SEANCE$ *et* $EMPRUNTVIDEO$*.*

En revanche, les DI ne définissent pas toujours des clés étrangères. Pour s'en convaincre, il suffit d'imaginer qu'on souhaite imposer que tous les cours possèdent au moins une séance dans l'année : on définira alors une DI

$$COURS[NUMCOURS] \subseteq SEANCE[NUMCOURS]$$

Ainsi, tous les cours apparaîtront au moins une fois dans la relation des séances ; toutefois, $NUMCOURS$ *n'est pas une clé de* $SEANCE$ *(on imagine difficilement que tous les cours n'aient qu'une seule séance !) et donc ce n'est pas une contrainte d'intégrité référentielle.*

3.5 Inférence, fermetures et couvertures de contraintes

Une *classe de contraintes d'intégrité* correspond à un type particulier de contraintes d'intégrité sur un schéma de relation ou de base de données. Par exemple, l'ensemble de toutes les dépendances fonctionnelles sur un schéma de relation R est une classe de contrainte d'intégrité.

L'*inférence de contraintes* est un concept fondamental pour la théorie des bases de données ; cela consiste à considérer les contraintes étant impliquées par d'autres contraintes. Les contraintes impliquées sont tout aussi valides que les contraintes de départ, mais souvent de façon implicite : elles échappent alors à la connaissance du concepteur. Le paragraphe suivant introduit ces notions importantes en prenant comme exemple les DF.

3.5.1 Inférence de dépendances fonctionnelles

Définition 8. *Implication de DF Soit* F *un ensemble de DF sur un schéma de relation* R*, et* f *une DF sur* R*. On dit que* F *implique* f*, noté* $F \models f$ *si pour toute relation* r *sur* R *telle que* $r \models F$*, on a* $r \models f$*.*

Exemple 8. *Soit le schéma*

$$ETUDIANT(NUMETUD, NOMETUD, NOMVILLE, REGION, CP)$$

et l'ensemble de DF sur ce schéma :

$$\{NUMETUD \rightarrow NOMETUD, NOMVILLE, REGION, CP; NOMVILLE, REGION \rightarrow CP\}$$

On comprend intuitivement que les deux DF suivantes seront également valides, par implication :

$$\{NUMETUD \rightarrow NOMVILLE, REGION; NUMETUD \rightarrow CP\}$$

Si ces implications sont intuitives, il est toutefois difficile et fastidieux de prouver que chaque intuition est bien juste ! Et surtout il est impossible d'être certain qu'on a bien pensé à toutes les DF impliquées par notre ensemble de DF de départ. Pour palier ces difficultés, et permettre un traitement automatique des contraintes, on définit la notion de *règles d'inférences.*

Définition 9. *Règle et système d'inférence Une règle d'inférence est une expression de la forme* $F \Rightarrow f$*. Un système d'inférence est un ensemble de règles d'inférence. Si* S *est un système d'inférence, on note* $F \vdash_S f$ *le fait qu'on puisse dériver* f *à partir de* F *par des applications successives de règles de* S*.*

Le système d'inférence des DF est le *système d'Armstrong*.

Définition 10. *Système d'inférence d'Armstrong Soit* F *un ensemble de DF sur un schéma de relation* R*. Les règles d'inférence de DF suivantes sont appelées* système d'inférence d'Armstrong *:*

- *Réflexivité : si* $Y \subseteq X \subseteq R$ *alors* $F \vdash X \rightarrow Y$*.*
- *Augmentation : si* $F \vdash X \rightarrow Y$ *et* $F \vdash W \subseteq R$ *alors* $F \vdash WX \rightarrow WY$*.*
- *Transitivité : si* $F \vdash X \rightarrow Y$ *et* $F \vdash Y \rightarrow Z$ *alors* $F \vdash X \rightarrow Z$

Si ce système d'inférence est important pour les DF, c'est qu'il possède les propriétés suivantes :

- Il est juste, soit $F \vdash f \Longleftrightarrow F \models f$;
- Il est complet, soit $F \models f \Longleftrightarrow F \vdash f$;
- Il est minimal en nombre de règles d'inférence.

Une conséquence de ce résultat est que $F \models \alpha \Leftrightarrow F \vdash \alpha$ **; nous utiliserons dans la suite la notation** $F \models \alpha$**.**

Exemple 9. *Considérons l'ensemble* $\Sigma = \{A \rightarrow C, B \rightarrow C, CD \rightarrow E\}$ *de df sur* $\{A, B, C, D, E\}$*. Montrons que* $\Sigma \models AD \rightarrow E$ *:*

σ_1 :	$A \rightarrow C$	$\in \Sigma$ *;*
σ_2 :	$AD \rightarrow CD$	*à partir de* **augmentation** *et* σ_1 *;*
σ_3 :	$CD \rightarrow E$	$\in \Sigma$ *;*
σ_4 :	$AD \rightarrow E$	*à partir de* σ_2 *et* σ_3 *en utilisant la* **transitivité**.

Ainsi, le système d'Armstrong permet de systématiser la recherche de toutes les DF impliquées par d'autres DF. Rigoureusement, on appelle *problème d'inférence des DF* le problème suivant : Soit F un ensemble de DF, et f une DF, a-t-on $F \models f$? La résolution de ce problème est "facile" pour les DF, et plus précisément linéaire en la taille de F et de f (c'est à dire l'espace nécessaire pour les écrire).

Le concept fondamental utilisé dans cet algorithme est celui de *fermeture d'un ensemble d'attributs*.

Soit F un ensemble de DF, on note F^+ (fermeture de F) l'ensemble de toutes les contraintes f telles que $F \models f$.

Enfin, si X est un ensemble d'attribut et F un ensemble de DF, on appelle fermeture de X par rapport à F l'ensemble de tous les attributs qu'on peut "déduire" de X par des dépendances fonctionnelles. Naturellement, il faut aussi tenir compte des DF qui ne sont pas dans F, mais qui sont dérivables à partir de F. Donc :

$$X^+ = \{A \mid F \models X \rightarrow A\}$$

ou encore

$$X^+ = \{A \mid X \rightarrow A \in F^+\}$$

Remarque 2. *On peut donner une définition alternative de la notion de clé d'une relation : X est clé d'un schéma R si et seulement si $X^+ = R$*

On peut facilement vérifier le lemme suivant :

Lemme 1. *Soient F un ensemble de DF et $X \rightarrow Y$ une DF. Alors on a $F \models X \rightarrow Y$ ssi on a $Y \subseteq X^+$.*

Ainsi, pour tester si on a $F \models X \rightarrow Y$, on calcule X^+. L'algorithme suivant peut être utilisé pour calculer cet ensemble.

Algorithm 1: Fermeture d'un ensemble d'attributs

Data: F un ensemble de DF, et X un ensemble d'attributs.
Result: X^+, la fermeture de X par F.

```
1 unused := F;
2 closure := X;
3 repeat
4     if W → Z ∈ unused and W ⊆ closure then
5         unused := unused − {W → Z};
6         closure := closure ∪ Z;
7 until jusqu'à ce qu'il n'y ait plus de changement ;
8 Print(closure);
```

Cet algorithme nous donne les moyens de vérifier si un ensemble de df implique une dépendance. Pour tester, l'implication d'un ensemble de dépendances, il suffit de tester indépendamment, l'implication de chaque dépendance de l'ensemble. On peut vérifier de plus que l'algorithme précédent tourne en temps $O(n^2)$, où n est la longueur de F et de X. On peut améliorer cet algorithme pour obtenir un temps linéaire. Pour toute df, $X \rightarrow Y$ de F non

utilisée, il faut stocker le nombre d'attributs de X non encore dans $closure$. Pour le faire efficacement, il faut maintenir à jour une liste pour chaque attribut A des df de F non utilisées pour lesquelles A apparait en partie gauche.

3.5.2 Inférence de dépendances d'inclusion

Le système d'inférence pour les DI est le suivant :

Définition 11. *Système d'inférence de Casanova et al. Soit I un ensemble de DI sur un schéma de base de données $\mathbf{R}$. Les règles d'inférence de DI suivantes sont appelées* système d'inférence de Casanova et al. *:*

- *Reflexivité : $I \vdash R[X] \subseteq R[X]$.*
- *Projection et permutation :*
  si $I \vdash R[A_1...A_n] \subseteq S[B_1...B_n]$ alors $I \vdash R[A_{\sigma 1}...A_{\sigma m}] \subseteq S[B_{\sigma 1}...B_{\sigma m}]$ pour chaque séquence $\sigma 1...\sigma m$ d'entier distincts dans $\{1...n\}$
- *transitivité : si $I \vdash R[X] \subseteq S[Y]$ et $I \vdash S[Y] \subseteq T[Z]$ alors $I \vdash R[X] \subseteq T[Z]$*

Ce système d'inférence est lui aussi juste, complet et minimal pour les dépendances d'inclusion. Contrairement aux DF, le problème d'inférence des DI est PSPACE-complet, donc infaisable dans le cas général.

La notion I^+ s'applique aussi pour noter la fermeture d'un ensemble de DI. En revanche, la notion de fermeture d'un ensemble d'attributs par rapport à un ensemble de DI ne s'applique pas, car les DI manipulent des séquences et non des ensembles.

3.5.3 Couvertures des DF et des DI

La notion de couverture est une relation d'équivalence entre des ensembles de contraintes.

Définition 12. *Couverture d'un ensemble de DF Soit Σ et Γ deux ensembles de contraintes. Γ est une couverture de Σ si $\Gamma^+ = \Sigma^+$.*

Une couverture d'un ensemble de DF est donc une représentation alternative, avec d'autres DF, mais véhiculant une sémantique rigoureusement équivalente : au final, c'est exactement le même ensemble de DF qui est implicite. C'est exactement la même chose pour les DI.

Trois propriétés sont importantes pour les couvertures des DF - peu d'études existent pour les DI.

- un ensemble F de DF est dit *non redondant* s'il n'existe pas de couverture G de F telle que $G \subset F$.
- un ensemble F de DF est dit *minimum* s'il n'existe pas de couverture G de F tel que $|G| \leqslant |F|$.

– F est dit *optimum* s'il n'existe pas de couverture G de F avec moins d'attributs que dans F.

Remarque : une couverture minimum est non redondante, une couverture optimum est minimum.

L'algorithme suivant calcule une couverture minimum pour un ensemble F de DF.

Algorithm 2: Couverture minimum d'un ensemble de DF

Data: F un ensemble de DF
Result: G une couverture minimum de F

```
1 G := ∅;
2 for X → Y ∈ F do
3     G := G ∪ {X → X+};
4 for X → X+ ∈ G do
5     if G − {X → X+} ⊢ X+ then
6         G := G − {X → X+};
7 return G;
```

Cet algorithme est polynomial dans le nombre de DF dans F et le nombre d'attributs dans F. La couverture minimum calculée par l'algorithme n'est pas forcément unique : d'autres couvertures peuvent avoir le même nombre de DF, mais être différentes. Parmi celles-ci, certaines sont optimum ; malheureusement, leur calcul est un problème "NP-Complet" dans le cas général.

3.5.4 Relation d'Armstrong

Etant donné un ensemble de df F sur U, existe t-il une instance I qui satisfasse F et qui viole toute df qui n'est pas dans F^+ ? Il se trouve que, pour tout ensemble de df, il existe une telle instance ; c'est ce qu'on appelle les *relations d'Armstrong*.

Dans certaines applications, les domaines de certains attributs peuvent être finis (par exemple, $Sexe$ a classiquement deux valeurs). Dans de tels cas, la construction d'une relation d'Armstrong peut être impossible. Les relations d'Armstrong peuvent en pratique être utilisées pour aider l'utilisateur à définir les df d'une application particulière. Un processus de spécification itératif et interactif débute par la spécification par l'utilisateur d'un premier ensemble de df. Le système génère alors une relation d'Armstrong pour les df, qui viole toutes les df non incluses dans la spécification. Cela sert de contre-exemple du pire cas, et ce qui en résulte est la détection de df supplémentaires dont la satisfaction est voulue.

Les relations d'Armstrong ne seront pas plus développées en cours mais feront l'objet d'une séance de TD.

Chapitre 4

Conception de bases de données

Sommaire

Nous entendons par conception, dans ce cours, le fait de choisir une modélisation des données réelles à partir du cahier des charges des applications. La modélisation est en relationnel : il s'agit donc de déterminer l'ensemble des attributs, des relations et des contraintes qui constitueront le modèle.

Nous allons voir dans la suite quelles sont les propriété attendues d'une bonne modélisation en relationnel, et comment les obtenir.

4.1 Anomalies de mise à jour et redondance

Une anomalie de mise à jour à lieu lorsque, à la suite d'une modification de la base, des contraintes sémantiques valides se trouvent violées. Bien sûr, des mécanismes de contrôle sont intégrés aux SGBDR pour éviter ce genre de problèmes. Mais celà suppose :

– une perte de temps dans la gestion de la base, certains contrôles pouvant être assez lourds ;
– une implémentation rigoureuse de toutes les contraintes par le concepteur de la base. Sous Oracle, celà passe bien souvent par la mise en place de ”Trigger” en PL/SQL.

Le compromis est alors atteint en faisant l'hypothèse suivante : le concepteur n'implémente que les clés et les clés étrangères. Le contrôle automatique de ces contraintes est peu coûteux par le SGBD, et leur implémentation est toujours intégrée dans les SGBDR. Ainsi, on considère que toute mise à jour respecte les clés.

R a une anomalie de mise à jour par rapport à F s'il est possible d'insérer ou de modifier un tuple t tel que :
– $r \cup t \models CLE(F)$, où $CLE(F)$ est l'ensemble des clés induites par F.
– $r \cup t \not\models F$.

Exemple 10. *Supposons le schéma de relation*

$$ETUDIANT(NUMETUD - A, NOM - B, VILLE - C, CP - D, DPT - E)$$

muni de l'ensemble de DF

$$A \rightarrow BCD, CD \rightarrow E$$

La seule clé minimale de la relation est donc A (Toutes les autres clés sont des sur-ensembles de A).

Supposons qu'un tuple soit inséré pour un nouvel étudiant, avec une ville et un CP déjà présent mais un département différent. La clé ne sera pas violée (pas de doublon sur A) mais la DF $CD \rightarrow E$ ne sera plus satisfaite. Puisqu'un tel cas de figure est possible, la relation $ETUDIANT$ possède une anomalie de mise à jour.

Une suppression ne peut entraîner aucune anomalie proprement dite ; toutefois, elle peut engendrer une perte involontaire d'information, lié à la redondance dans les données. Dans l'exemple précédent, si on supprime tous les étudiants de LYON, on aura également perdu le code postal et le département de Lyon, même si on souhaitais garder cette information.

La notion de *redondance* est une autre façon de considérer les problèmes de mises à jour. Elle se définit sur les relations, alors les problèmes de mise à jour portent sur des schémas.

Définition 13. *Une relation r sur R est redondante par rapport à un ensemble F de DF sur R si :*

1. $r \models F$ *et*
2. *il existe* $X \rightarrow A \in F$ *et* $t_1, t_2 \in r$ *tels que* $t_1[XA] = t_2[XA]$.

Exemple 11. *Reprenons l'exemple précédent, et considérons cette fois la relation suivante sur le schéma $ETUDIANT$:*

NUMETUD	*NOM*	*VILLE*	*CP*	*DPT*
1	*Fagin*	*Lyon*	*69003*	*Rhône*
2	*Armstrong*	*Lyon*	*69001*	*Rhône*
3	*Bunneman*	*Clermont*	*63000*	*Puys-de-Dôme*
4	*Codd*	*Lyon*	*69001*	*Rhône*

Cette relation est bien correcte car elle respecte les DF. Néanmoins, elle est redondante car il existe un doublon sur $(Ville, CP)$ *: l'information du département de LYON 1er apparaît deux fois.*

On voit bien que les notions d'anomalie de mise à jour et de redondance sont très liées - elles sont en fait équivalentes, selon le résultat suivant.

Théorème 1. *Il y a équivalence entre :*
- R *a une anomalie de mise à jour par rapport à* F*,*
- *Il existe une relation* r *sur* R *qui est redondante par rapport à* F*.*

4.2 Les pertes d'information

Le principe de base est alors de décomposer les relations de telle sorte d'éviter d'avoir ces anomalies, c'est à dire de transformer une relation en plusieurs relations. Avec des schémas de relation à un seul attribut, il n'y a plus aucun problème de redondance ! Le risque en décomposant est de perdre de l'information.

4.2.1 Perte de données

Tout d'abord, il faut que toutes les informations de la base de donnée initiale puissent être retrouvées en effectuant des jointures sur les relations issues de la décomposition.

Soit R un schéma de relation (c'est à dire un ensemble d'attributs), que l'on décompose en un schéma de base de données (un ensemble de relations) $\mathbf{R} = \{R_1, \ldots, R_n\}$. On dira alors que $\mathbf{R}$ est *sans perte de jointures* par rapport à un ensemble F de dépendances fonctionnelles si, pour toute relation r sur R telle que $r \models F$ on a :

$$r = \pi_{R_1}(r) \ldots \pi_{R_n}(r)$$

Exemple 12. *Reprenons la relation de l'exemple précédent. Supposons que pour régler le problème de redondance, on découpe le schéma en deux de façon à obtenir les relations suivantes :*

NUMETUD	*NOM*
1	*Fagin*
2	*Armstrong*
3	*Bunneman*
4	*Codd*

VILLE	*CP*	*DPT*
Lyon	*69003*	*Rhône*
Lyon	*69001*	*Rhône*
Clermont	*63000*	*Puys-de-Dôme*

4.2.2 Perte de DF

Ensuite, il ne faut pas que la décomposition ait ”coupé” des dépendances fonctionnelles, ce qui conduirai à une perte inévitable de sémantique. Pour cela, on définit la notion de projection d'un ensemble de dépendances fonctionnelles :

Définition 14. *projection d'un ensemble de DF Soit F un ensemble de DF sur R, et S un schéma de relation tel que $S \subseteq R$.*

$$F[S] = \{X \rightarrow Y \mid X \rightarrow Y \in F \text{ et } XY \subseteq S\}$$

La projection sur un schéma de bases de données est l'union des projection sur chaque relation du schéma.

Soit R un schéma de relation et F un ensemble de DF sur R. Un schéma de relation $\mathbf{R}$ est une *décomposition qui préserve les dépendances* de R par rapport à F si :

$$F[\mathbf{R}]^+ = F^+$$

4.3 Les principales formes normales

Les formes normales sont des propriétés que doivent vérifier les schémas pour éviter les anomalies de mises à jour. Plusieurs formes normales ont été définies. Une forme normale s'applique à un schéma de relation, en fonction d'un certain ensemble de contraintes d'une classe donnée. Concernant les DF, l'idée générale est de n'avoir que les clés à vérifier, et d'éliminer au maximum des DF qui ne définissent pas des clés. Dans la suite, soit R un schéma de relations et F un ensemble de DF définies sur R.

Rappel : en relationnel, on est toujours en première forme normale, soit toutes les valeurs des attributs sont atomiques.

Définition 15. *2FN R est en deuxième forme normale par rapport à F si, pour chaque DF $X \rightarrow A$ de F, l'une des deux conditions suivantes est remplies :*
- *A appartient à une clé de R,*
- *X n'est pas un sous-ensemble propre d'une clé de R*

Donc, aucun attribut (en dehors de ceux qui forment les clés), ne sont déterminés par des sous-ensembles des clés.

Par contre, même en dehors des clés, certains attributs peuvent être déterminés par des ensembles qui ne sont pas des sous-ensembles des clés. Ce qui est exclu dans la troisième forme normale.

Définition 16. *3FN R est en troisième forme normale par rapport à F si, pour chaque DF $X \rightarrow A$ de F, l'une des deux conditions suivantes est remplies :*
- *A appartient à une clé de R,*
- *X est une clé (ou une superclé)*

Ainsi, les seules DF "parasites" autorisées dans la troisième forme normale sont celles dont la partie droite est un attribut membre d'une clé.

Enfin, la forme normale de Boyce-Codd impose que toutes les parties gauches des DF sont des clés.

Définition 17. *FNBC R est en forme normale de Boyce-Codd par rapport à F si, pour chaque DF $X \rightarrow A$ de F, X est une superclé de R.*

Remarques :
- Si toutes les clés d'une relation sont composées d'un seul attribut, alors la troisième forme normale est équivalente à la forme normale de Boyce-Codd.
- Si une relation n'est composée que d'un ou deux attributs, elle est en FNBC.

La FNBC est, en ce qui concerne les DF, la forme idéale d'un schéma de bases de données. En effet, les trois propriétés suivantes sont équivalentes :
- R est en FNBC par rapport à F ;
- R n'a pas de problème de redondances par rapport à F ;
- R n'a pas de problème de mise à jour par rapport à F ;

Ajoutons que la troisième forme normale est toujours possible, quelque soit les DF considérées - la deuxième forme normale n'a donc qu'un intérêt "historique". En revanche, il existe des situations où la FNBC n'est pas possible.

Exemple 13. *Un exemple type : une rencontre sportive entre équipes, dans laquelle chaque équipe présente un seul athlète dans chaque épreuve ; le même athlète peut faire plusieurs épreuves. On a donc $(Equipe, Epreuve, Athlete)$ avec les DF :*

$$(Equipe, Epreuve \rightarrow Athlete; Athlete \rightarrow Equipe)$$

Quelle que soit la décomposition sans perte réalisée, elle n'est pas en FNBC (essayer toutes les décompositions imaginables à partir de ces trois attributs)

Dans un tel cas, il faut soit :
- Accepter de ne pas être en BCNF, et donc accepter d'avoir une anomalie de mise à jour. En d'autre termes, on ne pourra pas se contenter de déclarer les clés, mais il faudra implémenter les DF (Trigger sous un SGBD, ou au niveau de l'application) qui ne sont pas des clés.
- Accepter de relâcher des contraintes, c'est à dire enlever quelques DF et rendre l'application "plus souple".
- On peut aussi rajouter des attributs et des DF. Dans l'exemple, on peut rajouter un attribut qui "regroupe" les couples Equipe/Epreuve. On obtiendrait :

$$(Equipe, Epreuve, N^{\circ}Equipe/Epreuve, Athlete)$$

ainsi que les DF

$$(Equipe, Epreuve \rightarrow N^{\circ}Equipe/Epreuve)$$

$$(N^{\circ}Equipe/Epreuve \rightarrow Equipe, Epreuve, Athlete)$$

et

$$Athlete \rightarrow Equipe$$

La décomposition en FNBC est alors possible, et aucune sémantique n'est perdue.

4.3.1 Au delà des dépendances fonctionnelles

Les dépendances fonctionnelles nous ont permis jusque-là de mettre en évidence une forme de redondance, que les formes normales cherchent à faire disparaître. Mais des cas de redondance ne sont pas capturés par les DF.

Exemple 14. *Supposons l'énoncé suivant : "les étudiants suivent des parcours, et sont inscrits dans des transversales indépendantes du parcours. Chaque étudiant peut être inscrit à plusieurs parcours". Soit la modélisation suivante :*

$$R(etudiants, parcours, transversale)$$

On ne peut dégager aucune DF dans ce schéma, donc la seule clé est la combinaison des trois attributs.

La relation de l'exemple précédent est donc en FNBC, puisqu'aucune DF n'est valide. On se rend bien compte pourtant de la redondance présente dans ce schéma, car il faudra, pour

chaque étudiant, répéter tous les parcours à chaque transversale suivie. En effet, si on ne le fait, on risque d'induire une dépendance entre les parcours et les transversales qui n'est pas vraie.

La notion qui permet de modéliser cette redondance est celle de dépendance multivaluée. De façon intuitive, une dépendance multivaluée sert à modéliser que deux informations sont liées "indépendamment" des autres informations.

On notera :

$$etudiants->->parcours|transversale$$

pour signifier que "une étudiant est associé à plusieurs parcours et plusieurs transversales, indépendamment". Une telle contrainte est satisfaite dans la relation r si $xyz\ xy'z' \in r$ implique $xy'z$ et $xyz' \in r$.

Ainsi, on peut décomposer la relation est deux relations

$$R_1(etudiants, parcours); R_2(etudiantstransversale)$$

L'information pourra être retrouvée par jointure : en d'autres termes, on peut décomposer car la jointure reconstruira exactement la relation initiale (sans tuple supplémentaires).

On dira alors que R est en Quatrième forme normale, ce qui correspond à

- R est en troisième forme normale
- les seules dépendances multivaluées sont du type $X->->R-X$. Ou encore, R ne doit pas être décomposable en deux relations sans perte de jointure.

Remarque : 4FN $\Rightarrow$ 3FN $\Rightarrow$ 2FN.

4.4 Comment normaliser une relation

Un algorithme de normalisation prend en entrée une relation et des contraintes, et construit un schéma de BD normalisé, en 3FN ou en BCNF. Il existe deux grandes catégories d'algorithmes de normalisation : les algorithmes *de décomposition* et les algorithmes *de synthèse*. Nous ne verrons ici que les algorithmes de synthèse.

4.4.1 Algorithmes de synthèse

Entrée : une relation universelle (l'ensemble de tous les attributs du problème) et un ensemble de contraintes : DF, DI, DMV.

On commence par répertorier tous les attributs et construire ainsi la relation universelle de notre application. Puis on dresse l'inventaire des contraintes DF, DI, DMV.

Principe général

- Construire une couverture minimum de F, et réduire les parties gauches et droites au maximum

– Générer une relations XY pour chaque DF $X \rightarrow Y$;
– Générer une relations XY' pour chaque DMV $X->->Y$ avec $F \models Y' \rightarrow Y$;
– On supprime les schémas de relation qui ne sont pas maximaux par inclusion.
– S'il y a perte de jointure, alors on rajoute une relation composée d'une clé de F.

A partir de cette base, de nombreuses variantes incluent des heuristiques pour diminuer la redondance en sortie.

Propriétés : l'algorithme finit toujours, en donnant la meilleure forme normale (FNBC si elle existe, 4FN sinon)

L'algorithme suivant est utilisé pour réduire les parties gauches et droites des DF d'une couverture minimum. Attention, cet algorithme ne calcule pas une couverture *optimum*, qui contiendrait le nombre minimal d'attributs (qui est un problème NP-Complet). Ici, le nombre d'attribut est simplement minimal pour l'ensemble de DF pris en entrée.

Algorithm 3: Réduction du nombre d'attribut pour un ensemble de DF

Data: Un ensemble *minimum* de DF F sur R.
Result: F avec un nombre minimal d'attributs

```
1  Min := F;
   /* Réduction des parties gauches                                  */
2  for X → Y ∈ Min do
3      W := X;
4      for A ∈ X do
5          if Min ⊨ (W − A) → X then
6              W := W − {A};
7      Min := (Min − {X → Y}) ∪ {W → Y};
   /* Réduction des parties droites                                  */
8  for X → Y ∈ Min do
9      W := Y;
10     for A ∈ Y do
11         G := (Min − {X → Y}) ∪ {X → (W − A)};
12         if G ⊨ X → Y then
13             W := W − {A};
14     Min := (Min − {X → Y}) ∪ {X → W};
15 return Min;
```

Exemple 15. *On gère une liste de projets. Chaque projet a un responsable, un ensemble d'employés, et utilise certains produits en une quantité donnée. Pour un produit et un projet, plu-*

sieurs fournisseurs à des cout différents sont concernés. Un fournisseur a plusieurs adresses. Finalement, un employé a une date d'embauche et un salaire.

Soit $(Projets, produits, fournisseur, adresse, cout, responsable, employés, salaire, dateEmbauche)$.

Les DF sont :

$$Projet, produit \rightarrow quantite; Projet, fournisseur, produit \rightarrow coût$$

$$projet \rightarrow responsable; employe \rightarrow salaire, dateEmbauche$$

et on a deux MVD :

$$Fournisseur->->Location; projet->->employe, salaire, dateEmbauche$$

Après application de l'algorithme, on obtient les relations suivantes :

- $(Projet, Produit, Qute)$,
- $(Projet, Fournisseur, produit, cout)$,
- $(Projet, responsable)$,
- $(employe, salaire, dateEmbauche)$,
- $(fournisseur, location)$,
- $(projet, employe)$.

4.5 Convertir un schéma Entité/Association en schéma relationnel

4.5.1 Rappel : Le modèle Entité-Association

Le modèle Entité-Association est un modèle de données incomplet, dans le sens où il ne fournit aucun moyen de manipuler ou d'interroger les données. Sa popularité provient du fait qu'il permet de représenter *la sémantique* des données d'une application d'une manière relativement simple, intuitive pour un non spécialiste. Il facilite alors la tâche du concepteur d'une base de données, et possède l'avantage de parfaitement distinguer la spécification de la sémantique des données de l'implantation physique de la base, et plus généralement du modèle de données qui sera choisi par la suite pour la gestion effective de la base.

La part la plus importante et la plus populaire fournie par le modèle EA est le *diagramme entité-association*, qui permet une représentation visuelle (relativement...) intuitive des données. On distingue deux types d'*objets* différents : les entités et les associations entre entités. Plusieurs propositions ont été faites concernant les représentations graphiques des entités et associations ; celles utilisées ici sont une possibilité parmi d'autres, l'essentiel étant de comprendre les concepts abordés.

FIGURE 4.1 – EXEA

TODO

Les entités

Une **entité** est une ”chose” qui peuple les données à représenter. Dans l'exemple, une entité peut être un employé donné, ou un service, ou encore un auteur. Une entité appartient à un ou plusieurs **types d'entité**. Par exemple, une personne de l'entreprise peut être à la fois un employé et un auteur.

Un type d'entité est donc une collection d'entités ; il est décrit par un nom unique, et par une série d'*attributs*. Parmi les attributs, on distingue :

- Les attributs mono-valués : chaque entité du type entité concerné prend au plus une valeur pour cet attribut. Par exemple, un document possède un seul titre, un seul numéro d'identification, une seule date de parution.
- Les attributs multi-valués : pour une entité donné, un tel attribut peut prendre plusieurs valeurs. Par exemple, un document peut avoir plusieurs mots-clés, ou un employé peut avoir un ou plusieurs prénoms.

Certains attributs peuvent être des *clés* de leur entité, c'est à dire qu'ils identifient de façon unique chaque entité d'un type d'entité donné. Par exemple, le nom d'un service, ou le numéro NSS d'un employé sont des clés. Chaque type d'entité doit avoir au moins une *clé primaire*, qui une clé choisie parmi les autres. Graphiquement, la clé primaire sera soulignée pour la distinguer des autres attributs.

Les associations

Un *type d'association* est une association entre plusieurs types d'entités. nous nous limiterons ici aux types d'associations binaires, c'est à dire des types d'associations entre deux types d'entités. Par exemple, ECRIT est un type d'association entre les types d'entité DOCUMENT et AUTEUR. Un type d'association correspond donc à un ensemble d'*associations* qui peuplent les données. Une association de type ECRIT est donc une paire particulière composée d'une entité de type document et d'une entité de type auteur.

Un type d'association peut être décrite par des attributs. Sur le type d'association ECRIT, l'attribut *rang* dénote, pour une association composée d'un auteur et d'un document, quelle est la place de l'auteur parmi l'ensemble des auteurs de ce document (premier auteur, deuxième auteur, etc...). Une association est identifiée à partir des valeurs de clés primaires des entités participantes.

Les types d'associations peuvent être classés en trois catégories, suivant le nombre de fois qu'une entité donnée peut participer à une association.

- *Les types d'associations* $n-n$(ou plusieurs à plusieurs). Une entité peut alors participer à plusieurs associations de ce type. Exemple : l'association ECRIT entre AUTEUR et DOCUMENT est une association n-n ; un auteur particulier peut écrire plusieurs documents, et un document peut-être écrit par plusieurs auteurs.
- Les types d'association $1-n$ (ou un à plusieurs). Les deux types d'entités impliqués ne jouent pas alors des rôles symétriques. Une entité particulière du premier type pourra participer à plusieurs associations. Une entité particulière du deuxième type ne pourra participer qu'à au plus une seule association.
  Exemple 16. *Sur l'exemple, le type d'association MEMBRE, qui associe des entités de type SERVICE avec des entités de type EMPLOYEE, est une association* $1-n$*. Un employé ne peut être associé qu'à un seul service, alors qu'un service peut être associé à plusieurs employés.*
- *Les types d'association* $1-1$. Toute entité ne peut participer qu'une seule fois à une association de ce type.

Remarque - les cardinalités 1 et n se lisent "au plus 1" ou "au plus n". Il s'agit de cardinalités *maximales* ; une entité peut ne participer à aucune association. Par exemple, on peut avoir des documents qui n'ont pas d'auteurs, des employés qui n'ont pas de service. Notons qu'il est toutefois possible de contraindre des entités à participer à certaines associations, ce que nous ne détaillerons pas ici.

Les associations existentielles, ou entités faibles

Un type d'entité faible est un type d'entité particulier : l'existence d'une entité de type faible dépend de l'existence d'une autre entité. Le lien entre une entité faible et l'entité forte correspondante est appelé association existentielle. Nous en distinguerons deux sortes :

Associations ISA Elles permettent de coder la notion d'héritage entre types d'entités. Chaque entité du type faible est complètement identifié par une entité du type fort correspondant. Par exemple, un employé est une personne particulière. Il en possède toutes les caractéristiques, ainsi que des propriétés supplémentaires comme le numéro de bureau ou la date d'embauche. Notons que l'entité faible et l'entité forte apportent des informations différentes sur un seul et même objet conceptuel.

Graphiquement, ce lien est représenté par une flèche étiquetée "ISA".

Associations ID Dans ce cas, l'entité faible et l'entité forte ne correspondent pas au même objet, mais il existe un lien de "subordination" entre les deux. L'entité faible a tout simplement besoin de l'existence de l'entité forte. Sur l'exemple, le type d'entité EMPRUNT est faible. L'existence d'un emprunt particulier dans la base de données est dépendante de l'existence du document correspondant. Ce lien de dépendance se traduit par le fait que la valeur de la

clé de l'entité "dominante" se retrouve parmi la clé de l'entité faible. Dans l'exemple, la clé d'un emprunt est donnée par la clé du document correspondant, à laquelle on ajoute la date de l'emprunt. Muni de ces deux caractéristiques, un emprunt est parfaitement identifié.

Graphiquement, le lien entre un type d'entité faible et le type d'entité auquel il est rattaché est noté "ID".

4.5.2 Traduction E/A vers relationnel