

Introduction aux bases de données

Notes de cours

Stéphane Gañarski

13 janvier 2003

Table des matières

Préface et références bibliographiques	1
1 Généralités et modélisation conceptuelle	3
1.1 Généralités sur les systèmes de gestion de bases de données	4
1.1.1 Qu'est-ce qu'un SGBD?	4
1.1.2 Problèmes posés par l'absence de SGBD	4
1.1.3 Abstraction des données	4
1.1.4 Modèle de données	5
1.1.5 Les différents langages d'un SGBD	5
1.1.5.1 Langage de définition de données (LDD)	5
1.1.5.2 Langage de manipulation de données (LMD) et langage de requêtes	5
1.2 Le modèle Entité-Association	7
1.2.1 Entité - ensemble d'entité - attribut - identificateur	7
1.2.2 Association - ensemble d'associations	8
1.2.3 Entité forte - Entité faible	9
1.2.4 Généralisation - Spécialisation	9
1.2.5 Comment faire un bon schéma Entité-Association	10
2 Modèle et langages de requêtes relationnels	11
2.1 Le modèle relationnel	12
2.1.1 Structure d'une base de données relationnelle	12
2.1.2 Schéma et instance de BD relationnelle	12
2.1.3 Passage d'un schéma Entité-Association à un schéma Relationnel	13
2.1.4 Langages de requêtes relationnels	15
2.2 L'algèbre relationnelle	16
2.2.1 Opérateurs fondamentaux	16
2.2.1.1 Opérateurs fondamentaux unaires	16
2.2.1.2 Opérateurs fondamentaux binaires	17
2.2.2 Opérateurs dérivés	19
2.2.3 Conclusion sur l'algèbre relationnelle	20
2.3 Le calcul relationnel à variable nuplet	21
2.3.1 Introduction	21
2.3.2 Présentation intuitive	21
2.3.3 Définitions formelles	22
2.3.4 Formules "sûres"	22
2.3.5 Présentation des résultats	22
2.3.6 Équivalence entre algèbre et calcul	23
2.3.6.1 Problématique	23
2.3.6.2 Traduction des opérateurs algébriques en formules du calcul	23
2.3.7 Exemples	23
2.4 SQL	24
2.4.1 Requêtes simples	24

2.4.1.1	Forme principale	24
2.4.1.2	Cas particuliers	25
2.4.1.3	Présentation des résultats	25
2.4.2	Agrégats	26
2.4.2.1	Agrégats globaux	26
2.4.2.2	Agrégats partitionnés	26
2.4.3	Requêtes imbriquées	27
3	Manipulation et contraintes d'intégrité	29
3.1	Définition et manipulation de données en SQL	30
3.1.1	Gestion de schéma en SQL	30
3.1.2	Manipulation de données en SQL	30
3.1.2.1	Insertion de nuplets	30
3.1.2.2	Modification de nuplets	31
3.1.2.3	Suppression de nuplets	31
3.2	Contraintes d'intégrité en SQL	31
3.2.1	Généralités sur les contraintes d'intégrité	31
3.2.2	Typologie des contraintes dans le modèle relationnel	32
3.2.2.1	Contraintes de domaine	32
3.2.2.2	Contrainte de clé	32
3.2.2.3	Contraintes de table	32
3.2.2.4	Contraintes d'intégrité référentielle	32
3.2.2.5	Un exemple complet	33
3.2.2.6	Contraintes générales sur plusieurs tables	33
3.2.3	Maintenance des contraintes	34
4	Théorie de la conception de bases de données relationnelles	35
4.1	Conception d'un schéma de base de données relationnelle	36
4.1.1	Généralités	36
4.1.2	Anomalies d'un schéma relationnel	36
4.1.2.1	Redondance d'information	36
4.1.2.2	Différentes anomalies liées à la redondance	36
4.1.3	Changer de schéma	37
4.2	Dépendances fonctionnelles dans un schéma relationnel	37
4.2.1	Rappel	37
4.2.2	Notion de dépendance fonctionnelle (DF)	37
4.2.2.1	Notations	37
4.2.2.2	Définition	38
4.2.2.3	Cas particuliers	38
4.2.2.4	Évaluation d'une dépendance fonctionnelle	38
4.2.3	Calcul des dépendances fonctionnelles	38
4.2.3.1	Implication logique de DF	38
4.2.3.2	Fermeture transitive d'un ensemble d'attributs	39
4.2.3.3	Axiomes d'Armstrong [1974]	40
4.2.4	Equivalence et couverture minimale d'ensembles de DF	41
4.3	Décomposition de schéma et formes normales	44
4.3.1	Décomposition de schéma relationnel	44
4.3.1.1	Définition	44
4.3.1.2	Décomposition sans perte d'information (SPI)	44
4.3.1.3	Décomposition sans perte de dépendance (SPD)	46
4.3.2	Formes normales	47
4.3.2.1	Forme normale de Boyce-Codd (FNBC)	48
4.3.2.2	Troisième forme normale (3FN)	48
4.3.2.3	2e forme normale (2FN)	48

4.3.2.4	Algorithmes de passage aux formes normales FNBC et 3FN	49
5	Transactions et concurrence	51
5.1	Transactions	52
5.1.1	Problématique	52
5.1.2	Transactions	52
5.1.2.1	Définitions	52
5.1.2.2	Syntaxe des transactions	52
5.1.2.3	Propriétés ACID des transactions	53
5.2	Concurrence - Sériabilité	53
5.2.1	Exécution ou ordonnancement (d'un ensemble de transactions)	54
5.2.2	Sérialisation des transactions	54
5.2.2.1	Exécution en série	55
5.2.2.2	Actions permutable et exécutions équivalentes	55
5.2.2.3	Sériabilité	55
5.2.2.4	Relation de précédence et théorème	55
5.2.3	Contrôle de la concurrence	56
5.2.3.1	Définition	56
5.2.3.2	Les différentes stratégies	56
5.3	Le verrouillage deux phases	57
5.3.1	Deux types de verrou	57
5.3.2	Le protocole V2P	58
5.3.3	Interblocage	58
5.3.3.1	Problème	58
5.3.3.2	Solutions au problème d'interblocage	59
5.4	Sérialisation par estampillage (time-stamping)	59
5.4.1	Principe	59
5.4.2	Mécanisme	59
5.4.3	Règle de Thomas	60
5.5	Recouvrabilité	60

Bibliographie

- [Ullman88] Jeffrey D. Ullman. *Database and Knowledge-base systems. Vol I : Classical Database Systems*. Computer Science Press, Rockville, Maryland, 1988.
- [Korth Silberschatz91] Henry F. Korth et Abraham Silberschatz. *Database System Concepts*. Computer Science Series, Mac Graw-Hill, USA, 1991.

Préface

Ce document rassemble et met en forme les notes de cours accumulées depuis 1991. Il est basé sur l'expérience personnelle, les deux ouvrages cités précédemment et les collaborations avec d'autres enseignants en bases de données, notamment Geneviève Jomier, Professeur à l'Université Paris-Dauphine et Anne Doucet, Professeur à l'Université Pierre et Marie Curie (UPMC). Cette édition reprend et corrige l'édition précédente de janvier 2000.

Ce document ne prétend pas être complet sur les sujets abordés, ni faire un panorama exhaustif sur les bases de données. Il est destiné à servir de support (poly) aux étudiants et enseignants dans le cadre du programme de bases de données enseigné en Licence, d'Informatique de l'UPMC. Il peut être utilisé pour d'autres formations académiques (à l'exception de toute prestation commerciale) et sous réserve de citation de son auteur et de l'UPMC.

Paris, 13 janvier 2003

Chapitre 1

Généralités et modélisation conceptuelle

1.1 Généralités sur les systèmes de gestion de bases de données

1.1.1 Qu'est-ce qu'un SGBD ?

Un système de gestion de bases de données (SGBD) est un ensemble de programmes (gros logiciel) permettant à plusieurs utilisateurs d'accéder à un ensemble volumineux de données. Il doit être efficace et assurer la sécurité des données. Il doit aussi permettre aux utilisateurs de développer des applications manipulant ces données, mais aussi d'interroger les données de manière ad hoc (non prédéfinie) et si possible déclarative.

1.1.2 Problèmes posés par l'absence de SGBD

Le développement d'une application sans SGBD pose de nombreux problèmes. Dans ce cas, les programmes d'application sont écrits directement au-dessus du système de gestion de fichiers. Imaginons le cas du système informatique d'une banque : celui-ci sera composé de nombreux programmes manipulant de nombreux fichiers.

- retrait/crédit d'un compte
- ajout d'un compte
- solde d'un compte
- relevés de compte
- etc.

Au fur et à mesure, de plus en plus de programmes et de plus en plus de fichiers (compte épargne, sicav...) vont être créés, par différents programmeurs, dans différents langages, utilisant des formats de fichiers différents.

Les inconvénients d'une telle approche sont nombreux :

- redondance : coût de stockage et d'accès. Par exemple, les coordonnées d'un client seront stockées dans plusieurs fichiers.
- incohérence : souvent liée à la redondance. Par exemple, lors du changement d'adresse d'un client, il faut vérifier que cette adresse est mise à jour dans tous les fichiers où elle apparaît (encore faut-il les connaître).
- difficulté d'accès : les utilisateurs ne peuvent pas faire des requêtes non prévues dans les programmes.
- isolation des données : un nouveau programme peut avoir à chercher des données dans des fichiers variés, aux différents formats (différents langages), ce qui est souvent très compliqué.
- anomalies dues à la concurrence : l'exécution simultanée de programmes modifiant les mêmes données peut créer des incohérences. Par exemple, si deux programmes lisent le même solde d'un compte et ajoutent chacun une certaine somme à ce compte, seul le dernier ajout sera pris en compte.
- manque de sécurité : tout le monde ne doit pas pouvoir voir ou modifier les mêmes données. (ex : un employé ne peut voir que son salaire, pas celui des autres, et il n'a pas le droit de le modifier). Or très souvent on ne peut pas protéger un fichier car certaines de ses données sont privées, d'autres ne le sont pas.
- gestion de l'intégrité : les données obéissent à des contraintes. Par exemple, un compte ne doit pas avoir un solde inférieur à -10 000. Il faut donc vérifier cette contrainte *dans tous les programmes qui peuvent modifier le solde d'un compte*. De plus, si la contrainte change (-15 000 au lieu de -10 000), il faut retrouver tous les programmes qui vérifient la contrainte, les modifier au bon endroit et tout recompiler. On a ainsi une absence de coordination entre applications, chacune d'entre elles faisant *ce qu'elle veut*.

Ce sont ces problèmes, parmi d'autres, qui ont mené au développement des SGBD.

1.1.3 Abstraction des données

L'un des principaux buts des SGBD pour pallier les problèmes évoqués ci-dessus est de fournir l'abstraction des données, c'est-à-dire l'indépendance entre *ce que voient les utilisateurs*, qui doit être clair et "non-informatique" et *ce qui est effectivement géré* qui doit être efficace.

On considère habituellement trois niveaux d'abstraction : physique, conceptuel et vue (référence ANSI/SPARC). Chaque niveau correspond à une catégorie d'utilisateurs différente :

- le niveau physique concerne l'organisation physique (en machine) des données, comment elle sont stockées et comment y accéder rapidement (grâce à des index). Ce niveau dépend de la plateforme sur laquelle est mise en œuvre la base de données. Seul l'administrateur de la base de données est principalement concerné par ce niveau.
- le niveau conceptuel décrit “quelles” données sont stockées, à l'aide de structures assez simples. C'est à ce niveau qu'on décrit par exemple qu'une personne possède un nom qui est une chaîne de caractères, une date de naissance, une adresse... Ce niveau est en général géré par les concepteurs de la base de données, éventuellement en collaboration avec l'administrateur de la base de données et les programmeurs d'application.
- le niveau vue (ou externe) permet à chaque application de disposer des données qu'elle peut manipuler (le plus souvent des sous-ensembles du niveau conceptuel). Par exemple, une agence de la banque n'aura accès qu'aux comptes de ses clients.

La distinction de plusieurs niveaux d'abstraction permet aux SGBD d'offrir 2 types d'indépendance :

- physique : si on modifie le stockage des données (ex : changement d'organisation ou de format des fichiers), tous les accès à la base de données restent valides.
- logique : si on modifie le niveau conceptuel, seules les vues concernées doivent être éventuellement modifiées, les programmes d'application restent inchangés.

Pour définir le niveau conceptuel et aussi les vues, il faut un “langage commun” à tous les utilisateurs, c'est le *modèle de données*, présenté dans la section suivante.

1.1.4 Modèle de données

Un modèle de données est un *ensemble de concepts* qui permet de *décrire les données*, les liens entre données, la sémantique des données (ce que représentent les données), les contraintes d'intégrité sur les données . . . Il en existe beaucoup. On peut néanmoins distinguer les modèles de type “conceptuel” (utilisés pour l'analyse des applications) et les modèles de type “logique” (associés au SGBD). Les premiers sont plus lisibles (souvent graphiques), les seconds plus facilement “implantables”. La démarche habituelle est donc d'utiliser un modèle de données conceptuel pour définir la base de données, entre les différents intervenants. Ensuite, le schéma conceptuel est traduit dans le modèle de données logique pour être implanté dans le SGBD.

Dans ce cours, on verra les deux standards : le modèle conceptuel *Entité-Association* (concepts : entité, association, spécialisation, attributs, identificateur . . .) et le modèle logique *Relationnel* (concepts : relation, attribut, domaine, clé, n-uplets, . . .).

Les modèles de données sont la *clé de voûte* des SGBD car :

1. ils servent de support aux outils de manipulation des données (langages de définition des données, de manipulation des données et de requêtes, interface)
2. ils conditionnent la façon d'implanter le niveau physique.

1.1.5 Les différents langages d'un SGBD

1.1.5.1 Langage de définition de données (LDD)

Le LDD permet de construire le *schéma* de la BD, (qui est stocké par le SGBD dans un catalogue ou dictionnaire). Par exemple, dans le cas du modèle relationnel : créer une relation, décrire le domaine d'un attribut . . .

Le schéma décrit le *contenu* de la BD, aussi appelé *instance*. L'instance doit donc respecter les spécifications du schéma.

1.1.5.2 Langage de manipulation de données (LMD) et langage de requêtes

Une fois que le schéma (qui décrit les données) a été créé grâce au LDD, les utilisateurs peuvent manipuler les données (donc l'instance de la BD) grâce au LMD. Celui-ci remplit deux fonctions :

- Retrouver l’information (lecture) : par exemple, “liste des clients ayant plus de 1000F sur leur compte”
Assez souvent cette partie est distinguée du reste du LMD car c’est une grande particularité des SGBD. En effet, un SGBD peut contenir de très grandes quantités d’information et il est donc nécessaire de disposer d’un outil sophistiqué pour permettre d’en extraire les données désirées. C’est pourquoi, la fonction d’extraction de données fait souvent l’objet d’un sous-langage : le *langage de requêtes*.
- Mettre-à-jour l’information : c’est-à-dire, ajouter, supprimer ou modifier des données.

Le LMD peut être procédural, mais on préfère qu’il le soit le moins possible, notamment pour la partie langage de requêtes. L’utilisateur doit avoir à exprimer “quelles sont les données qu’il désire”, pas “comment les trouver”. En d’autres termes, un langage de requête doit être le plus *déclaratif* possible, tout en évitant qu’il soit trop complexe (et donc trop lent) à évaluer.

1.2 Le modèle Entité-Association

Introduction

Le modèle Entité-Association (E/R en anglais, pour Entity-Relationship), introduit par Chen en 1976, est le plus connu des modèles conceptuels pour le développement d'applications de bases de données. Il sert de base à de nombreuses méthodes de conception, comme Merise, ou même OMT. Il est particulièrement adapté pour la description de bases de données qui vont être ensuite implantées dans des SGBD relationnels.

Il est constitué de 3 notions (concepts) principales : Entité - Attribut - Association. D'autres concepts viennent compléter le modèle.

1.2.1 Entité - ensemble d'entité - attribut - identificateur

Définition 1.2.1 Une entité est un "objet", un élément du monde qui existe et se distingue des autres. Une entité est représentée par un ensemble d'attributs valués qui la décrivent. \square

Par exemple, le compte du client A est une entité décrite par son numéro, le nom et le prénom du client A, le solde du compte, ...

Définition 1.2.2 Un attribut porte un nom et est associé à un domaine (string, [1..100]..). Il peut être vu comme une fonction entité \rightarrow domaine \square

Par exemple : l'attribut solde donne le solde d'un compte client. Vu comme une fonction, on aurait $solde(compte_client_A) = -200$

Définition 1.2.3 Un ensemble (ou classe) d'entités regroupe des entités "homogènes" (de même "genre"), c'est-à-dire ayant les mêmes attributs. Une classe d'entités est donc définie par son nom et par ses attributs. Elle représente un "schéma" pour les entités qu'elle contient. \square

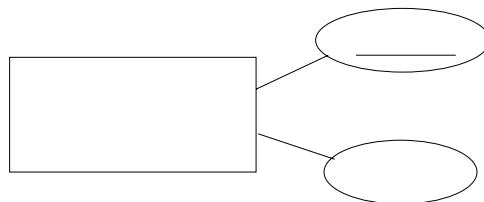
Par exemple, la classe Compte_clients regroupe les entités représentant des comptes clients. PAR ABUS DE LANGAGE, on appelle entité un ensemble d'entités homogènes.

Définition 1.2.4 Un identificateur d'une classe d'entités est un attribut ou un ensemble d'attributs de la classe permettant de distinguer une entité d'une autre du même ensemble \square

Par exemple, numéro_de_compte est un identificateur de la classe Compte_clients

Représentation graphique :

Une classe d'entités est représentée par un rectangle, avec le nom de la classe à l'intérieur du rectangle. Les attributs sont représentés par des ovales où figurent le nom de l'attribut et rattachés par une arête au rectangle de la classe d'entité. Le ou les attributs identificateurs sont soulignés.



1.2.2 Association - ensemble d'associations

Définition 1.2.5 Une **association** met en relation plusieurs entités. Comme les entités, une association peut avoir des attributs. □

Par exemple, le client Dupont est associé au compte numéro 1234, ou Dupont a_pour_compte #1234

Une association avec des attributs (ici l'attribut depuis) est par exemple l'association reliant l'employé Durand au service qui l'emploie, c'est-à-dire Durand est_au_service Boursier "depuis 1985"

Comme les entités, les associations sont regroupées en ensembles homogènes.

Définition 1.2.6 Un **ensemble (ou classe) d'associations** porte un nom et regroupe les associations de même type : ayant le même sens, les mêmes attributs, reliant des entités de même genre. □

Par exemple, l'association a_pour_compte relie les entités suivantes :

Dupond ↔ #1234

Momo ↔ #1254

...

PAR ABUS DE LANGAGE, on appelle "association" une classe d'associations, lorsqu'il n'y a pas d'ambiguïté.

- Les associations ne sont pas forcément binaires. Par exemple l'association a_pour_prof relie un étudiant, un professeur et une matière : Toto a Lulu comme professeur de Bases de données.
- Une association peut être réflexive, comme par exemple Personne époux_de Personne

Définition 1.2.7 Les **cardinalités** d'une classe d'associations indiquent, pour une entité, combien d'associations de cette classe sont possibles. □

On distingue les cardinalités *maximum* et *minimum* :

- maximum : 1 ou N (M, P, ...)

ex : un client peut avoir plusieurs compte, un client a au maximum un livret d'épargne.

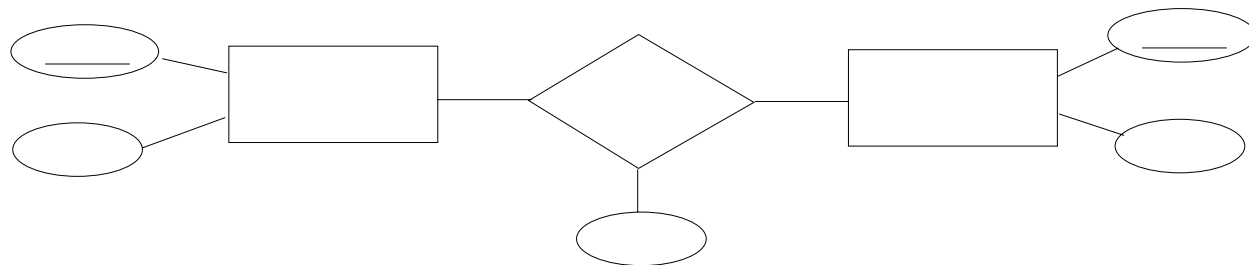
- minimum : 0 ou 1

ex : un compte est detenu par au moins un client, un client peut ne pas avoir de livret d'épargne.

Le couple (cardinalité maximum, cardinalité minimum) est représenté ainsi min :max.

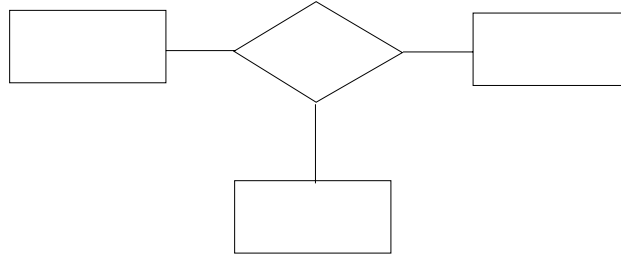
Représentation graphique :

Une association est représentée par un losange avec le nom à l'intérieur. Elle est reliée par des arêtes à chaque classe qu'elle relie. Les attributs éventuels de l'association sont représentés par des ovals où figurent le nom de l'attribut et rattachés par une arête au losange de l'association. Les cardinalités sont inscrites sur les arêtes allant de l'entité considérée vers le losange de l'association.



La figure ne montre pas tous les attributs des entités. On voit grâce aux cardinalités qu'un client peut ne pas avoir de compte (0) ou en avoir plusieurs (M). En revanche, un compte a forcément un titulaire (1) mais peut en avoir plusieurs (N).

Attention, les associations ternaires ne sont pas équivalentes à trois associations binaires. Ainsi, le modèle de la figure ci-dessous permet de modéliser qui enseigne les bases de données à Toto, ce que ne permettrait pas trois associations binaires entre chaque couple d'entité (on saurait quelle matière étudie Toto, quels professeurs enseignent les bases de données et quels professeurs enseignent à Toto, mais sans savoir lequel d'entre eux est celui qui lui enseigne les bases de données)



1.2.3 Entité forte - Entité faible

Jusqu'à présent, nous n'avons considéré que des *entités fortes*, c'est-à-dire des entités qu'on peut identifier indépendamment des autres. Ce n'est pas toujours le cas, c'est pourquoi il est nécessaire de définir des *entités faibles*.

Définition 1.2.8 *Entités fortes et faibles peuvent être définies ainsi :*

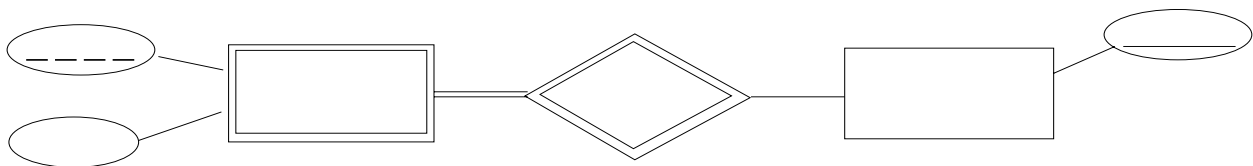
- Une **entité forte** est une entité entièrement identifiable à l'aide de ses attributs propres.
- Une **entité faible** ne peut être identifiée que par rapport à une autre entité, appelée **entité dominante** à laquelle elle se réfère. Par conséquent, l'identificateur d'une entité faible est formé d'un **identificateur partiel**, formé par un ou plusieurs attributs de l'entité faible, auquel on rajoute l'identificateur de l'entité dominante référencée.

□

Par exemple, l'entité faible *salle de cours* a pour entité dominante le *bâtiment* dans lequel elle se trouve. Son identificateur partiel est donc le numéro de salle, son identificateur est la concaténation du numéro de salle et du numéro de bâtiment (qui est l'identificateur de l'entité dominante dans ce cas).

Représentation graphique :

L'entité faible est représentée par un double cadre. L'association avec l'entité dominante est elle aussi en double trait. L'identificateur partiel de l'entité faible est souligné en pointillés.



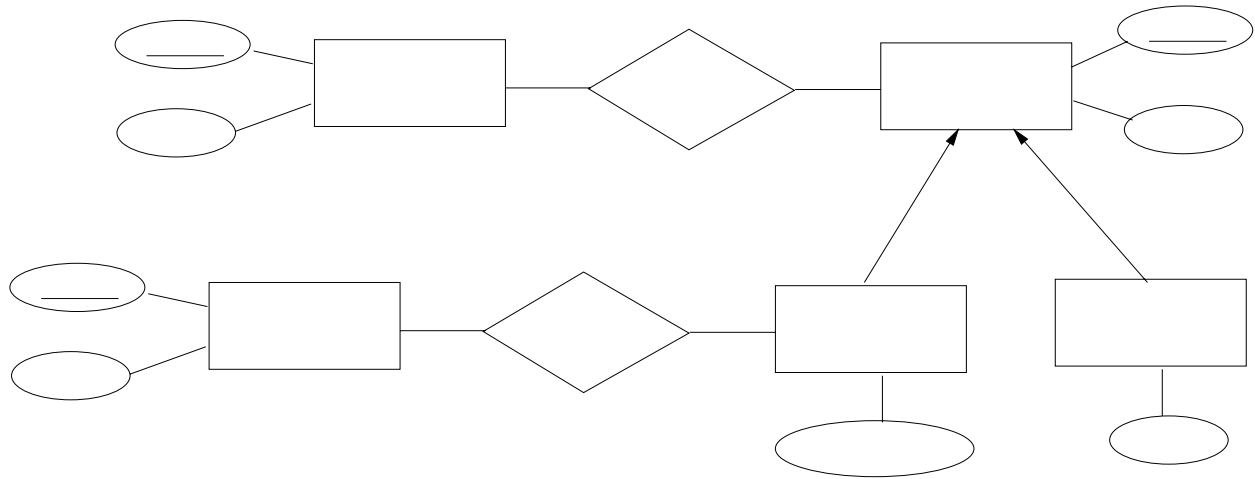
1.2.4 Généralisation - Spécialisation

On remarque que souvent, plusieurs classes d'entités se "ressemblent" sémantiquement et structurellement. Afin d'obtenir un schéma plus compact qui montre ces ressemblances, on crée une nouvelle entité qui rassemble les points communs des classes qui se ressemblent. C'est le processus de **généralisation**. Ensuite, on rajoute une association particulière, appelée "est_un" entre la nouvelle classe générale et les classes qui se ressemblent, qui sont souvent appelées "sous-classes". Ces sous-classes ayant par défaut les attributs et les associations de la classe générale (appelée "super-classe"), on ne représente pour ces sous-classes que les attributs et les associations qui leur sont spécifiques. On dit que ces sous-classes **spécialisent** la classe générale.

Par exemple, on peut créer une classe générale `Compte` qui regroupe les caractéristiques communes à tous les comptes (`numero_de_compte`, `solde`), et des sous-classes `Compte_chèque` (attribut spécifique `decouvert_autorise`, association spécifique avec les cartes de crédits du compte) et `Compte_epargne` (attribut spécifique `taux_d_epargne`).

Représentation graphique :

Les sous-classes sont reliées à la super-classe par une simple flèche pointant vers la super-classe.



1.2.5 Comment faire un bon schéma Entité-Association

Les règles suivantes peuvent être contradictoires, prendre dans ce cas la plus restrictive.

1. déterminer les entités et attributs
 - si une information décrit un “objet” celui-ci doit être modélisé par une entité
 - pas d’attribut multivalué. Il faut donc le transformer en une entité associée par une association de cardinalité maximum N. Par exemple, si une personne peut posséder plusieurs voitures, il faut créer une entité `voiture` et l’associer à l’entité `personne` par une association `possède`
2. attacher les attributs aux entités qu’ils décrivent “le plus directement”.
3. à compléter avec l’expérience

Chapitre 2

Modèle et langages de requêtes relationnels

2.1 Le modèle relationnel

Introduction

Le modèle relationnel a été introduit par E.F. Codd en 1970. Il offrait à l'époque un niveau d'abstraction plus élevé que les modèles précédents (hiérarchique, réseau). Il est actuellement le plus répandu dans les SGBD existants, quelle que soit la plate-forme matérielle et logicielle.

2.1.1 Structure d'une base de données relationnelle

Intuitivement, une BD relationnelle est un ensemble de tables. Chaque ligne d'une table est un *n-uplet*. Chaque colonne représente un *attribut*. Donc une case de la table est la valeur d'un n-uplet pour un attribut.

Nom(Char[15])	Prénom(Char[10])	Numtél(Int[10])	→ schéma de relation
Dupond	Toto	0123428978	
Burr	Raymond	0321843939	→ Instance ou relation
Rominet	Titi	0293493900	

Définition 2.1.1 Plus formellement, à chaque attribut on associe un domaine. La définition de la table, c'est-à-dire son nom et la liste de ses attributs+domaine, est appelé **schéma de relation S**. L'ensemble des lignes ou n-uplets est appelé **instance de S** ou **relation de schéma S**. □

Pourquoi $\{relation\}$? En mathématique, une relation entre plusieurs domaines de valeur est un sous-ensemble du produit cartésien. Or ici c'est bien ce que l'on fait. En fait, l'ensemble des n-uplets d'une relation est bien un sous-ensemble du produit cartésien des domaines du schéma de la relation.

Par exemple, dans notre cas, les différents domaines sont :

- char[15] = (A,AA, ...,BURR,BURRA ..., DUPOND, ...,DURAND, ..., ROMINET, ...,ZZZZZZZZZZZZZZ)
- char[10] = (A,AA, ..., Raymond, ..., Titi, ..., Toto, ..., ZZZZZZZZZZ)
- Int[10] = (0000000000, ..., 9999999999)

Le produit cartésien de ces trois domaines donne :

char[15] X char[10] X int[10] =
 ((A, A, 0000000000), (A, A, 0000000001), ..., (Dupond, Toto, 0123428978), ..., (Durand, Titi, 0143288989), ..., (Burr, Raymond, 0321843939), ..., (Rominet, Titi, 0293493900), ..., (ZZZZZZZZZZZZZZ, ZZZZZZZZZZ, 9999999999))

On vérifie bien que l'instance de notre relation, qui est égale à :

((Dupond, Toto, 0123428978), (Burr, Raymond, 0321843939), (Rominet, Titi, 0293493900))

est bien un sous-ensemble du produit cartésien précédent.

2.1.2 Schéma et instance de BD relationnelle

Comme on l'a vu au chapitre 1.1, une base de données est formée d'un schéma qui décrit les données et d'une instance qui contient les données. Le schéma est manipulé par le langage de définition des données alors que l'instance est manipulée par le langage de manipulation des données.

Définition 2.1.2 Dans le cas relationnel, le schéma de la base de données est défini à l'aide des concepts suivants :

- Un **schéma de BD relationnelle** est un ensemble de schémas de relation.
- Un **schéma de relation** est défini par son nom et par un ensemble d'attributs (dans certains livres, il peut être défini sous forme de liste, mais cette précision formelle ne nous intéressera pas dans ce cours).
- Un **attribut** est donné par son nom et par son domaine.

- Un **domaine** est un ensemble de valeurs, ex. *Int* pour les entiers, *CHAR[10]* pour les chaînes de 10 caractères. Il est toujours de type atomique (pas de domaine de type liste, structure ou ensemble). Il peut être défini et nommé par l'utilisateur. Par exemple, on peut définir le domaine *Age* comme l'ensemble des entiers positifs inférieurs à 130.

On dit que deux domaines sont compatibles s'ils sont définis à partir du même type de base. Par exemple, le domaine *Age* est compatible avec celui des entiers, pas avec celui des chaînes de caractères. □

L'instance d'une telle base de données sera donc un ensemble de relations associées chacune à un schéma de relation.

Définition 2.1.3 Une **relation** est un ensemble de *n*-uplets dont les attributs correspondent à ceux du schéma de la relation. C'est un *jjvrai* ensemble : il ne peut donc y avoir de doublons ni d'ordre entre les *n*-uplets. □

Bien que nous y reviendrons plus tard, on peut déjà introduire les notions de clé et de valeur nulle :

- Une *surclé* d'un schéma de relation est un sous-ensemble des attributs de ce schéma tel que deux *n*-uplets d'une relation sur ce schéma ne peuvent avoir la même valeur pour la surclé. Notons que l'ensemble des attributs du schéma de relation est toujours une surclé de ce schéma.
- Une *clé ou clé minimale* d'un schéma de relation est une surclé de ce schéma possédant la propriété de minimalité au sens de l'inclusion : si on retire un quelconque des attributs de la clé, alors le reste ne forme plus une surclé.
- Dans les systèmes commerciaux, il est possible d'attribuer une *jjvaleur nulle* pour un attribut d'un *n*-uplet, lorsqu'on ne connaît pas cette valeur. Cette possibilité pratique pose néanmoins des problèmes d'interprétation de la relation qui font qu'une *jjbonne* base de données ne doit pas contenir de valeurs nulles.

Lorsque c'est nécessaire, on distingue un schéma de relation *R* d'une instance *r* de ce schéma qu'on notera *r(R)*. Lorsqu'il y a une seule instance par schéma de relation, on pourra confondre les deux.

2.1.3 Passage d'un schéma Entité-Association à un schéma Relationnel

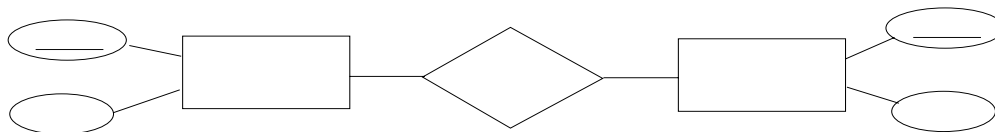
On décrit dans cette section les différentes étapes pour passer d'un schéma E/A (utilisé pour concevoir la base de données) à un schéma relationnel (qui sera implanté dans le SGBD).

1. Pour chaque classe d'entités, créer un schéma de relation avec les mêmes attributs. La clé de ce schéma de relation est l'identificateur de la classe d'entités.

Par exemple, l'entité *Compte_courant* ayant comme attributs *Numéro-compte* (identificateur) et *Solde* sera représentée par la relation *CompteCourant* dont les attributs sont *Numéro-compte* (clé) et *Solde*.

2. Pour chaque association dont au moins une classe d'entité associée a une cardinalité 1 :1, rajouter dans le schéma de relation correspondant à cette entité les clés des schémas de relation correspondant aux autres classes d'entités associées et, s'il y en a, les attributs de l'association.

Par exemple, si un client a toujours son compte domicilié dans une et une seule agence, alors l'association *domicilié* liant un client à l'agence qui gère son compte sera représentée en rajoutant, dans la relation *Client* la clé *NuméroAgence* de la relation qui représente les agences. Si un client peut être dans plusieurs agences ou dans aucune, alors on passe au cas suivant.

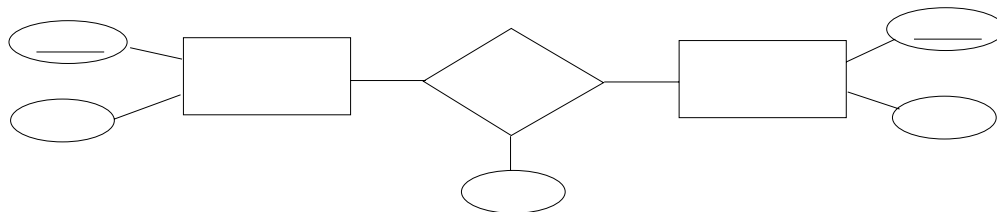


se traduit en *Agence*(#agence, adresse), *Client*(#client, nom, #agence)

3. Pour tous les autres cas d'association, créer un schéma de relation comportant les clés des relations représentant les classes d'entités associées et, s'il y en a, les attributs de l'association. La clé d'une telle relation est formée par l'ensemble des clés des relations représentant les classes d'entités associées

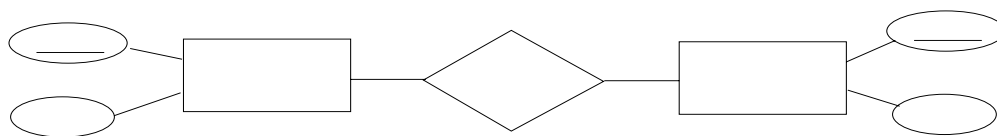
Par exemple, l'association `a_pour_compte` qui lie les clients à leurs comptes sera représenté par une relation `ACompte` qui comporte le `NuméroClient`, clé de la relation `Clients`, le `NuméroCompte`, clé de la relation `CompteClient` et l'attribut `Depuis`, attribut de l'association `a_pour_compte` qui indique depuis quand le client est associé à ce compte.

- (a) Le schéma E/A suivant :



se traduit en `CompteClient(#compte,solde)`, `Client(#client,nom)`, `ACompte(#client,#compte, depuis)`

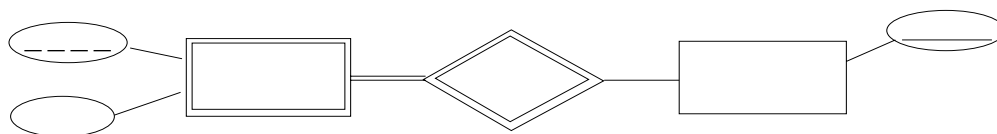
- (b) Le schéma E/A suivant :



se traduit en `Agence(#agence, adresse)`, `Client(#client, nom)`, `EstDans(#client, #agence)`

Reste à traiter le cas de entités faibles et de la généralisation.

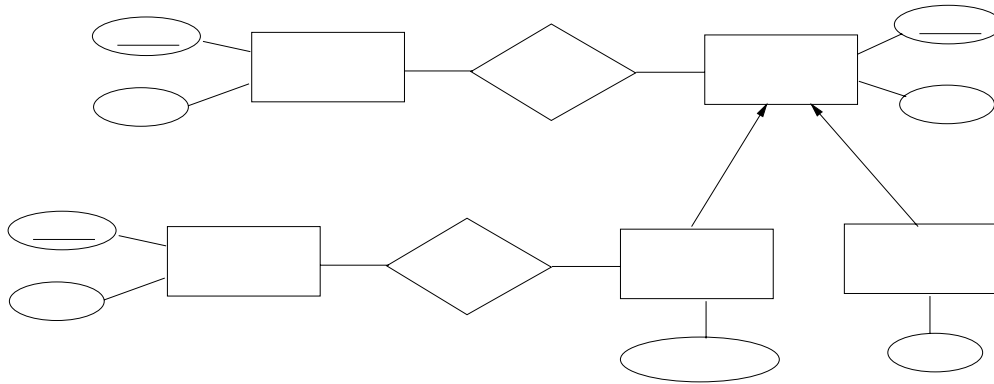
4. Pour représenter une entité faible et son association vers l'entité dominante, la seule particularité est que la clé de l'entité dominante fait partie de la clé de l'entité faible.



se traduit en `Salle(#salle, #batiment, nbplaces)`, `Batiment(#batiment,...)`

5. Pour représenter des sous-classes d'une super classe générale, on peut créer un schéma de relation pour la classe générale qui comporte les attributs communs et ensuite un schéma de relation pour chaque sous-classe qui comporte la clé de la relation qui représente la classe générale, ainsi que les attributs spécifiques à la sous-classe.

Dans le cas où la classe générale est virtuelle (elle ne contient pas d'entité mais sert uniquement à factoriser les propriétés), on peut, afin de condenser le schéma relationnel, ne pas créer de schéma de relation pour la super-classe. Dans ce cas, chaque schéma de relation des sous-classes devra comporter aussi les attributs communs.



se traduit en `Compte(#compte, solde)`, `Compte-cheque(decouvert_autorise', #compte)`,
`Compte-epargne(taux, #compte)`

On ne représente pas la traduction des autres entités et associations car elles se ramènent aux cas précédents.

Si `Compte` est virtuelle (il n'y a pas de `Compte`, il n'y a que des comptes chèque ou des comptes épargne), alors on peut se passer de la relation `Compte`, en `;;descendant;;` ses attributs dans les sous-classes. On obtient :

`Compte-cheque(decouvert_autorise', #compte, solde)`, `Compte-epargne(taux, #compte, solde)`

Notons que, pour éviter d'avoir un schéma trop "éclaté", on peut regrouper les deux tables précédentes, mais se pose alors le problème des valeurs nulles pour les attributs exclusifs à l'une ou l'autre des sous-classes.

2.1.4 Langages de requêtes relationnels

Dans la suite de cette partie, on s'intéressera aux divers langages associés au modèle relationnel. On mettra l'accent sur les langages de requêtes car ce sont eux qui font la spécificité des langages de manipulation de données.

On distingue deux catégories de langages relationnels :

- les langages théoriques qui permettent d'étudier le modèle, de prouver des propriétés et d'obtenir une mise en œuvre efficace.
 - Comme langage procédural, on étudiera l'algèbre relationnelle
 - Comme langage déclaratif, on étudiera le calcul relationnel à variables n-uplet, mais il en existe d'autres comme le calcul à variable domaine.
- les langages `;;commerciaux;;` qui sont proposés aux utilisateurs dans les SGBD. Ils ont une syntaxe plus conviviale et des `;;rajouts;;` pratiques. Pour être conviviaux, ils doivent être déclaratifs, ils sont donc basés sur les langages théoriques déclaratifs. On verra principalement SQL, qui est basé sur le calcul relationnel à variables n-uplet et qui est le standard des bases de données relationnelles. On peut citer aussi QBE, basé sur le calcul à variables domaine.

2.2 L'algèbre relationnelle

Introduction

Le terme “algèbre” désigne un ensemble d'opérateurs qui manipulent des expressions et dont le résultat est une expression qui peut ensuite être manipulée par les mêmes opérateurs. L'algèbre relationnelle manipule des *expressions relationnelles*, ses opérateurs prennent donc toujours en entrée une (ou deux) expressions relationnelles et produisent une expression relationnelle en sortie. On peut ainsi combiner (ou composer) les opérateurs relationnels autant qu'on veut, jusqu'à avoir obtenu en sortie du dernier opérateur appliqué le résultat désiré. La composition des opérateurs s'appelle une *requête d'algèbre relationnelle* et le résultat *l'évaluation de la requête*.

Définition 2.2.1 Une *expression relationnelle* se définit récursivement :

- Un schéma de relation est une expression relationnelle.
- Si $Op1$ est un opérateur relationnel unaire et Exp est une expression relationnelle, alors $Op1(Exp)$ est une expression relationnelle.
- Si $Op2$ est un opérateur relationnel binaire et $Exp1$ et $Exp2$ sont des expressions relationnelles, alors $Op2(Exp1, Exp2)$ est une expression relationnelle.

□

Dans la suite, nous présentons les différents opérateurs de l'algèbre relationnelle, tout d'abord les opérateurs fondamentaux (2.2.1), puis les opérateurs dérivés (2.2.2), qui peuvent s'exprimer en fonction des opérateurs fondamentaux mais qui sont d'une grande utilité pratique. Pour cela, nous nous appuyons sur la base de données exemple suivante :

Schéma de l'exemple :

Le schéma contient les trois schémas de relation suivants. Les clés sont soulignées. Les attributs Parent, Enfant et Personne ont même domaine.

Parenté(Parent, Enfant), Descriptif(Personne, Age, Sexe), Scolarité(Enfant, Ecole)

Instance de l'exemple :

Parenté		Descriptif			Scolarité	
Parent	Enfant	Personne	Age	Sexe	Enfant	Ecole
Toto	Titi	Toto	40	M	Zoe	A
Toto	Tata	Titi	20	M	Titi	B
Raymond	Zoe	Tata	18	F	Tata	A
Clara	Zoe	Zoe	2	F		
Marcel	Raymond	Clara	27	F		
		Marcel	60	M		
		Raymond	40	M		
		Johnny	65	M		

2.2.1 Opérateurs fondamentaux

Les opérateurs fondamentaux de l'algèbre sont au nombre de cinq. Deux opérateurs unaires : la *sélection* et la *projection*. Trois opérateurs binaires : le *produit cartésien*, l'*union* et la *différence*.

2.2.1.1 Opérateurs fondamentaux unaires

- **Sélection :** $\sigma_F(Exp)$

Définition 2.2.2 La *sélection* consiste à filtrer les n -uplets de l'expression donnée en paramètre selon une condition de sélection F . F est une formule de logique du premier ordre formée uniquement à partir des éléments suivant :

- constantes
- attributs figurant dans Exp
- comparateurs : =, <, >, ≥, ≤, ≠
- connecteurs logiques : ∨ (“ou”), ∧ (“et”), ¬ (“non”)

Le résultat est une expression de même schéma que Exp et qui contient tous les n -uplets de Exp tels que F est vraie □

Par exemple, pour obtenir la liste des filles de notre base de données exemple, il suffit de poser la requête :

$$\sigma_{Sexe='F'}(Descriptif)$$

Résultat :

Personne	Age	Sexe
Tata	18	F
Zoe	2	F
Clara	27	F

• **Projection** $\prod_{Att_1, Att_2, \dots}(Exp)$

Définition 2.2.3 La **projection** consiste à ne garder d' Exp que les attributs figurant dans Att_1, Att_2, \dots . Ceux-ci doivent donc être des attributs de Exp .

Le résultat a pour schéma Att_1, Att_2, \dots et contient les mêmes n -uplets que Exp , mais tronqués des attributs qui ne sont pas dans Att_1, Att_2, \dots □

ATTENTION : le résultat d'une projection n'a pas forcément le même nombre de n -uplets que Exp , car il se peut qu'on produise deux fois le même n -uplet en tronquant. Et comme le résultat est une expression relationnelle, c'est un “vrai ensemble” qui ne contient pas de doublons.

Par exemple, pour obtenir la liste des parents, on pose la requête : $\prod_{Parent}(Parenté)$

Résultat :

Parent
Toto
Marcel
Raymond
Clara

On peut maintenant composer sélection et projection, pour obtenir la liste des enfants de Raymond :

$$\prod_{Enfant}(\sigma_{Parent='Raymond'}(Parenté))$$

Résultat :

Enfant
Zoe

2.2.1.2 Opérateurs fondamentaux binaires

Les opérateurs fondamentaux binaires sont des opérateurs classiques de la théorie des ensembles.

• **Produit cartésien** $Exp1 * Exp2$

Définition 2.2.4 Le **produit cartésien**, aussi noté $Exp1 \times Exp2$, permet de “mettre ensemble” les données de plusieurs relations n’ayant pas le même schéma.

Le résultat est une relation de schéma $schéma(Exp1) \cup schéma(Exp2)$ (en cas d’attributs commun, on renomme $Exp1.att, Exp2.att$) qui contient tous les n -uplets qu’on peut construire en associant un n -uplet de $Exp1$ et un n -uplet de $Exp2$. Le résultat contient donc $Card(Exp1) \times Card(Exp2)$ n -uplets. \square

Le produit cartésien est rarement utilisé en tant que tel car il produit des n -uplets qui n’ont souvent aucun sens (on lui préférera une forme dérivée, la *jointure* définie plus loin). Par exemple, si on fait le produit cartésien des relations Parenté et Scolarité, on obtient :

Parent	Parenté.Enfant	Scolarité.Enfant	Ecole
Toto	Titi	Zoe	A
Toto	Titi	Titi	B
Toto	Titi	Tata	A
Clara	Zoe	Zoe	A
Clara	Zoe	Titi	B
Clara	Zoe	Tata	A
Marcel	Raymond	Zoe	A
Marcel	Raymond	Titi	B
Marcel	Raymond	Tata	A
Toto	Tata	Zoe	A
Toto	Tata	Titi	B
Toto	Tata	Tata	A
Raymond	Zoe	Zoe	A
Raymond	Zoe	Titi	B
Raymond	Zoe	Tata	A

• **Union**

$Exp1 \cup Exp2$

et **Différence**

$Exp1 - Exp2$

Définition 2.2.5 L’**union** et la **différence** sont les opérateurs ensemblistes bien connus, considérant les expressions relationnelles comme des ensembles de n -uplets.

Dans les deux cas, $Exp1$ et $Exp2$ doivent avoir le même schéma, ou du moins des schémas compatibles (attributs 2 à 2 de domaine compatible). Le résultat a donc le même schéma que $Exp1$ et $Exp2$. Il contient les n -uplets de $Exp1$ et les n -uplets de $Exp2$ (cas de l’union), ou les n -uplets de $Exp1$ qui ne sont pas dans $Exp2$ (cas de la différence). \square

Par exemple, on obtient les enfants de Raymond ou Toto par la requête :

$$\Pi_{Enfant}(\sigma_{Parent="Raymond"}(Parenté)) \cup \Pi_{Enfant}(\sigma_{Parent="Toto"}(Parenté))$$

Résultat :

Enfant
Zoe
Titi
Tata

On obtient la liste des enfants non-scolarisés par la requête : $\Pi_{Enfant}(Parenté) - \Pi_{Enfant}(Scolarité)$

Résultat :

Enfant
Raymond

2.2.2 Opérateurs dérivés

Les opérateurs dérivés peuvent se déduire des cinq opérateurs fondamentaux.

- **Jointure** $Exp1 \bowtie Exp2$

Définition 2.2.6 La jointure s'exprime en fonction du produit cartésien et de la sélection :

- La **jointure** $Exp1 \bowtie Exp2$, où F est habituellement une condition d'égalité entre un (des) attribut(s) de $Exp1$ et un (des) attribut(s) de $Exp2$ permet dans le produit cartésien de ne retenir que les "mariages" de n -uplets qui ont un sens par rapport à une certaine condition.

$$Exp1 \bowtie Exp2 = \sigma_F(Exp1 * Exp2)$$

- Si on ne précise pas F , celle-ci est construite à partir de tous les attributs communs à $Exp1$ et $Exp2$. On parle alors de **jointure naturelle** et, dans le résultat, on ne duplique pas les attributs communs. \square

Par exemple, la liste des parents et de l'école de leurs enfants s'obtient par :

$$\prod_{Parent, Ecole} Parenté \bowtie Scolarite$$

Résultat :

Parent	Ecole
Toto	B
Toto	A
Clara	A
Raymond	A

Notons que pour faire une *auto-jointure* (jointure d'une relation avec elle-même), il est nécessaire d'effectuer un **renommage d'attribut** pour lever l'ambiguïté. Par exemple, pour avoir la liste des GrandsParents et PetitsEnfants, on effectue la requête

$$\begin{array}{c} Parenté \\ Parent \rightarrow GP \end{array} \bowtie \begin{array}{c} \\ Enfant = Parent \end{array} \begin{array}{c} Parenté \\ Enfant \rightarrow PE \end{array} \quad \text{Résultat : } \begin{array}{|c|c|} \hline \mathbf{GP} & \mathbf{PE} \\ \hline Marcel & Zoe \\ \hline \end{array}$$

- **Intersection** $Exp1 \cap Exp2$

Définition 2.2.7 L'intersection $Exp1 \cap Exp2$ s'exprime classiquement à partir de la différence :

$Exp1 \cap Exp2 = Exp1 - (Exp1 - Exp2)$. Elle contient les n -uplets qui sont à la fois dans $Exp1$ et $Exp2$. \square

Par exemple, pour trouver les enfants de Raymond et Clara, on pose la requête :

$$\prod_{Enfant} (\sigma_{Parent="Raymond"}(Parenté)) \cap \prod_{Enfant} (\sigma_{Parent="Clara"}(Parenté))$$

Résultat :

Enfant
Zoe

- **Division** $Exp1 \div Exp2$

Définition 2.2.8 Pour effectuer la **division** $Exp1 \div Exp2$, il faut que les attributs de $Exp2$ forment un sous-ensemble des attributs de $Exp1$. Le résultat a pour schéma le reste des attributs de $Exp1$ et contient des n -uplets sur ces attributs restant qui sont, dans $Exp1$, associés à tous les n -uplets de $Exp2$

□

Par exemple, pour savoir quels sont les parents qui ont un enfant dans chaque école, on obtient d'abord la liste des parents et des écoles de leurs enfants par la requête $\prod_{Parent, Ecole}(Parenté \bowtie Scolarité)$. Il suffit ensuite de diviser le résultat par la liste des écoles qu'on obtient par la requête $\prod_{Ecole}(Scolarité)$

La requête complète est donc :

$$\prod_{Parent, Ecole}(Parenté \bowtie Scolarité) \div \prod_{Ecole}(Scolarité)$$

Résultat :

Parent
Toto

Expression de la division à l'aide des autres opérateurs à faire en exercice.

2.2.3 Conclusion sur l'algèbre relationnelle

- Grâce aux opérateurs unaires on peut extraire toute sous-relation d'une relation.
 - Grâce aux opérateurs de produit cartésien et de jointure on peut associer des données de plusieurs tables.
 - Grâce à l'union, l'intersection et la différence on peut sélectionner des n -uplets selon les valeurs dans d'autres tables.
- ⇒ On peut "presque tout" faire. Mais on ne peut pas "tout faire". Par exemple, on ne peut pas former une requête qui nous donnerait tous les ancêtres d'une personne.

2.3 Le calcul relationnel à variable nuplet

2.3.1 Introduction

L'algèbre relationnelle est un langage procédural. On doit indiquer "comment" obtenir la réponse. Or, du point de vue de l'utilisateur, ce qu'on voudrait, c'est exprimer uniquement "ce qu'on veut".

⇒ Problème pratique : on veut que des non-informaticiens puissent faire des requêtes sans savoir ce qu'est une jointure, une division . . .

⇒ Problème théorique : Pour obtenir un langage déclaratif, il faut faire le lien avec la logique. Ici, le mieux adapté (simple mais suffisamment expressif) est la logique des prédicats (ou logique du premier ordre).

Avec cette logique, on exprime des formules, qui peuvent être vraies ou fausses :

- $A \geq B$

- Pierre est le père de Paul

⇒ on voit ici que la deuxième formule est vraie si dans la relation Parenté(Parent, Enfant) on a le nuplet (Pierre,Paul). Mais si il n'y est pas ? 2 hypothèses :

- Hypothèse de "Monde fermé" : tout ce qui n'est pas dans la base de données est faux. Pierre n'est donc pas le père de Paul.

- Hypothèse de "Monde ouvert" : pour ce qui n'est pas dans la BD, on ne sait pas.

Évidemment, l'hypothèse de monde ouvert est plus réaliste, mais l'hypothèse de monde fermé est plus facile à employer, c'est celle qui est généralement utilisée.

NB : pour l'algèbre relationnelle, le fait qu'on utilise l'hypothèse de monde fermé est plus évidente car elle est implicite.

Il existe deux sortes de calcul relationnels : le calcul à variables n-uplets, qui est à la base de SQL, et le calcul à variables domaine qui est à la base de QBE. Ici nous ne verrons que le premier.

2.3.2 Présentation intuitive

On reprend l'exemple utilisé pour l'algèbre relationnelle.

- les variables du calcul représentent des nuplets des différentes relations, d'où le nom de "calcul à variables nuplets"

- Une requête exprime le fait qu'on veut en sortie (résultat) des nuplets qui vérifient une certaine formule logique. Une requête est donc de la forme :

- $\{t \mid F(t)\}$, on veut les nuplets t tels que $F(t)$ est vraie.

- t est la seule variable libre de F .

- Pour exprimer la projection, on utilise la notation pointée $t.att$ qui désigne la valeur de l'attribut att pour le nuplet t , bien qu'elle soit jugée comme un abus de langage par certains auteurs.

- Que peut-on exprimer dans $F(t)$?

- $R(t)$, exprime que le nuplet variable t appartient à R (se note aussi $t \in R$)

- $u.att = t.att$, $t.att1 = t'.att2$, et aussi avec d'autres comparateurs tels que $<$, $>$, \leq , \geq , \neq

- $F1(t) \vee F2(t)$ (OU), et aussi $\neg F(t)$, $F1(t) \wedge F2(t)$, $F1(t) \rightarrow F2(t)$ (ET, NON, IMPLIQUE)

- Exemple : $\{t \mid Descriptif(t) \wedge (t.age > 10) \wedge (t.sexe = 'f')\}$ renvoie les personnes (qui sont donc dans la relation Descriptif) de plus de dix ans et de sexe féminin.

- Introduction d'autres variables nuplets (liées) à l'aide de quantificateurs :

- $\exists u F(u)$ est évaluée à vrai s'il existe au moins un nuplet u qui vérifie F .

Par exemple $\{t.enfant \mid Parenté(t) \wedge (\exists u Parenté(u) \wedge u.enfant \neq t.enfant \wedge t.parent = u.parent)\}$ renvoie la liste des enfants ayant des (demi)-frères et/ou des (demi)-soeurs.

- $\forall u F(u)$ est évaluée à vrai si F vrai pour chaque u .

ATTENTION, ce quantificateur ne peut s'utiliser que si F débute par $R(u) \rightarrow \dots$, car sinon la formule n'est jamais vraie (une propriété, sauf si c'est une tautologie, ne peut être vraie pour "tous les nuplets du monde"). C'est le problème de ne fabriquer que des formules sûres, développé en 2.3.4.

Par exemple $\forall u(\text{Parent}(u) \rightarrow (u.\text{parent} = \text{"Pierre"}))$ est vraie ssi Pierre est le seul Parent de la base.

2.3.3 Définitions formelles

Définition 2.3.1 Une requête du calcul relationnel est de la forme $\{t \mid F(t)\}$ ou F est une formule correcte et t la seule variable libre de F . Elle renvoie les nuplets t pour lesquelles $F(t)$ est évaluée à vrai. \square

Les définitions suivantes permettent de construire les formules *correctes* du calcul à variables nuplets.

Définition 2.3.2

- Les **atomes** sont construits de la manière suivante :
 - Si R est un relation de la base, alors $R(u)$ est un atome.
 - Si u (resp. v) est une variable décrivant un nuplet de R (resp. S) et que A (resp. B) est un attribut de R (resp. S), alors $u.A \Theta v.B$ et $u.A \Theta$ constante sont des atomes (avec $\Theta \in \{=, >, <, \leq, \geq, \neq\}$) et A, B , et constante ont des domaines compatibles.
 - Rien d'autre n'est un atome
- Les **formules** sont construites de la manière suivante
 - Un atome est une formule
 - Si F est une formule, (F) et $\neg(F)$ sont des formules.
 - Si $F1$ et $F2$ sont des formules, $(F1 \wedge F2)$ et $(F1 \vee F2)$ sont des formules. Grâce à l'équivalence $(P \rightarrow Q) \Leftrightarrow \neg(P \vee \neg Q)$, on en déduit que $F1 \rightarrow F2$ est une formule.
 - Si F est une formule et v une variable libre de F , alors $(\exists v F(v))$ et $(\forall v F(v))$ sont des formules.
 - Rien d'autre n'est une formule

\square

Bien entendu, on peut aboutir à des formules différentes dans la forme mais qui sont équivalentes. Par exemple $(P1 \wedge P2) \Leftrightarrow \neg(\neg P1 \vee \neg P2)$, ou encore $(\forall x P(x)) \Leftrightarrow \neg(\exists x(\neg P(x)))$

2.3.4 Formules “sûres”

Avec ce qui précède, on peut générer des requêtes qui soit renvoient des infinités de nuplets ou, plus généralement, ne sont pas évaluables grâce au contenu des relations de la base. On dit que ces requêtes ne sont pas “sûres” et il faudra les éviter. Par exemple $\{t \mid \neg(\text{Parent}(t))\}$ n'est pas une requête sûre puisqu'elle n'indique pas dans quelle relation il faut chercher les nuplets t (elle indique seulement où ne pas les chercher). De même, la requête $\{t \mid R(t) \wedge \exists s(s.A \neq c') \wedge (s.B = t.B)\}$ n'est pas sûre car on ne sait pas où chercher les nuplets s .

La caractérisation des formules “sûres” est assez compliquée et dépasse le cadre de ce cours (on peut la trouver par exemple dans le livre de J. D. Ullman). On donne ici cependant certains “trucs” qui permettent de les éviter.

- Toujours commencer une requête par $\{t \mid R(t) \wedge \dots\}$
- Un $\exists x$ doit toujours être suivi d'un $R(x)$ (afin de savoir dans quelle relation x doit être trouvé)
- Éviter d'utiliser \forall et le remplacer par \exists grâce à l'équivalence $(\forall x P(x)) \Leftrightarrow \neg(\exists x(\neg P(x)))$

2.3.5 Présentation des résultats

Lorsqu'une requête renvoie des nuplets formés par l'association de nuplets de plusieurs relations (si on veut faire l'équivalent d'une jointure par exemple), ou bien par la projection sur certains attributs, il est nécessaire de pouvoir “construire” des nuplets résultats à partir des variables qui parcourent les relations de la base.

Pour cela, on peut utiliser les notations suivantes :

- $\{t.\text{att1}, t.\text{att2}, \dots \mid F(t)\}$ si on veut projeter le résultat sur $\text{att1}, \text{att2}, \dots$

- $\{t.att, t'.att' \mid R(t) \wedge S(t') \dots\}$ si on veut concaténer les attributs de nuplets venant de R et de S (une notation plus “formelle” consiste à déclarer un nuplet t^n ayant n attributs et de déclarer dans la formule que ce nuplet a les mêmes valeurs que t et t' pour les attributs concernés)

2.3.6 Équivalence entre algèbre et calcul

2.3.6.1 Problématique

On peut montrer que le calcul à variables nuplets (réduit aux formules sûres) est équivalent à l’algèbre relationnelle (permet d’exprimer les mêmes questions). Pour cela, il faut montrer que toute requête de l’algèbre peut être traduite en calcul et vice-versa. Si la traduction des opérateurs algébriques en formules du calcul ne pose pas de problème (voir ci-dessous), la traduction de n’importe quelle requête du calcul en une expression algébrique est évidemment plus compliquée (passer d’une forme déclarative à une forme procédurale) et ne sera pas abordée ici.

2.3.6.2 Traduction des opérateurs algébriques en formules du calcul

Les requêtes sont effectuées sur les relations $R(A_1, \dots, A_{n_1})$ et $S(B_1, \dots, B_{n_2})$

Traduction des opérateurs de base

- $\prod_{att1, att2, \dots}(R) : \{t.att1, t.att2, \dots \mid R(t)\}$
- $\sigma_F(R) : \{t \mid R(t) \wedge F'(t)\}$ où F' est obtenue en remplaçant dans F un nom d’attribut A_i par $t.A_i$
- $R \cup S : \{t \mid R(t) \vee S(t)\}$
- $R - S : \{t \mid R(t) \wedge \neg(S(t))\}$
- $R * S : \{t, t' \mid R(t) \wedge S(t')\}$

(dans la notation formelle exprimée plus haut, on aurait la requête

$$\{t^{n_1+n_2} \mid \exists u \exists v, R(u) \wedge S(v) \wedge t.A_1 = u.A_1 \wedge \dots \wedge t.A_{n_1} = u.A_{n_1} \dots \wedge t.B_1 = v.B_1 \wedge \dots \wedge t.B_{n_2} = v.B_{n_2}\})$$

Traduction des opérateurs dérivés

La traduction de la jointure et de l’intersection est évidente, elle peut être faire à titre d’exercice.

La division peut se traduire ainsi, en supposant que les attributs de R qui ne sont pas attributs de S sont les k premiers (on a donc $k = n_1 - n_2$) :

$$R \div S : \{t.A_1, t.A_2, \dots, t.A_k \mid R(t) \wedge (\forall u(S(u) \rightarrow (\exists v, R(v) \wedge t.A_1 = v.A_1 \wedge t.A_2 = v.A_2 \dots \wedge t.A_k = v.A_k \wedge v.A_{n_2} = u.A_{n_2}))\}$$

2.3.7 Exemples

1. Les parents de Pierre :

$$\{t.parent \mid Parenté(t) \wedge t.enfant = \text{“Pierre”}\}$$

2. Les parents de Pierre et Paul :

$$\{t.parent \mid Parenté(t) \wedge t.enfant = \text{“Pierre”} \wedge (\exists u, Parenté(u) \wedge u.parent = t.parent \wedge u.enfant = \text{“Paul”})\}$$

Faire attention que :

$$\{t.parent \mid Parenté(t) \wedge t.enfant = \text{“Pierre”} \wedge t.enfant = \text{“Paul”}\}$$

n’est pas la bonne solution. Cette requête renvoie le vide.

3. Les parents de Pierre et ceux de Paul $\{t.parent \mid Parenté(t) \wedge t.enfant = \text{“Pierre”} \vee t.enfant = \text{“Paul”}\}$

4. Les parents ayant des filles de plus de 13 ans :

$$\{t.parent \mid Parenté(t) \wedge (\exists u, Descriptif(u) \wedge u.personne = t.enfant \wedge u.sexe = \text{“F”} \wedge u.age > 13)\}$$

2.4 SQL

Généralités

Si les langages théoriques (algèbre, calcul) permettent de prouver l'intérêt du modèle relationnel et de le mettre en œuvre efficacement dans les SGBD, il est nécessaire de développer des langages adaptés à une *utilisation commerciale des SGBD*. Ces langages doivent intégrer la possibilité d'interroger la base de données sous forme déclarative, en se basant donc sur un langage théorique de type calcul. C'est le cas pour SQL qui est basé sur le calcul à variables n-uplets et dont nous présentons la partie "requêtes" dans ce chapitre. Mais, en plus du langage de requêtes, il faut aussi que le SGBD propose un langage de définition des données (LDD) permettant de créer et modifier le schéma de la base, de gérer des vues, des contraintes d'intégrité et les utilisateurs, et un langage de manipulation des données permettant d'ajouter, de supprimer ou de modifier des données de la base. Ces caractéristiques de SQL seront vues au chapitre 3.

Ceci dit, SQL reste plutôt une norme qu'un langage réel. Ainsi, chaque SGBD possède un interpréteur SQL différent, plus ou moins proche de la norme. Néanmoins, en dehors de variations syntaxiques, ce qui est présenté dans ce chapitre doit être communément trouvé dans tous les SGBD relationnels.

La base de données exemple est toujours celle utilisée dans les chapitres sur l'algèbre et sur le calcul.

2.4.1 Requêtes simples

2.4.1.1 Forme principale

Définition 2.4.1 Une requête SQL est de la forme :

```
select A1, A2, . . . , Am
from R1, R2, . . . , Rn
where F
```

telle que :

- $\forall i, A_i$ est un nom d'attribut
- $\forall j, R_j$ est un nom de relation
- F est une formule

□

Intuitivement, une telle requête projette sur les attributs de la clause `select`, les n-uplets parcourant les relations de la clause `from` vérifiant la formule de la clause `where`. La complexité du langage provient principalement de la complexité de la formule F . Dans le cas le plus courant, une telle requête correspond à la requête de calcul relationnel :

$$\{t_{x_1}.A_1, t_{x_2}.A_2, \dots, t_{x_m}.A_m \mid R_1(t_{a_1}) \wedge \dots \wedge R_n(t_{a_n}) \wedge F\}$$

avec $\forall i, t_{x_i} \in \{t_{a_1}, t_{a_2}, \dots, t_{a_n}\}$

Si F est une formule "du genre" de celles permises pour la condition de sélection en algèbre, alors une telle requête peut aussi se traduire en algèbre relationnelle par :

$$\prod_{A_1, A_2, \dots, A_m} (\sigma_F(R_1 * R_2 * \dots * R_n))$$

Par exemple, la requête suivante donne la liste des enfants de Marcel :

```
select Enfant
from Parenté where Parent = "Marcel"
```

2.4.1.2 Cas particuliers

- **Pas de projection** : `select * from ... where ...` renvoie en résultat tous les attributs possibles des variables déclarées dans la clause `from`. Par exemple, pour avoir la liste des personnes de plus de 20 ans avec leur âge et sexe, on pose la requête `select * from Descriptif where Age > 20`
- **Pas de sélection** : si on omet la clause `where`, cela veut dire qu'on ne veut pas de condition de sélection sur les n-uplets déclarés dans la clause `from`. Par exemple, pour avoir la liste des parents, on pose la requête `select Parent from Parenté`.
- **Élimination des doublons** : SQL n'étant pas un langage théorique, il peut renvoyer des résultats qui contiennent des doublons lorsqu'on effectue une projection. Ainsi la requête précédente renverra deux fois Toto car il est parent de Titi et de Tata. Pour éviter cela, il faut rajouter le mot clé `distinct` dans la clause `select`. La requête devient alors `select distinct Parent from Parenté`
- **Désambigüer une requête** : par défaut, les attributs déclarés dans la clause `select` sont associés à une seule variable n-uplet de la clause `from`. Dans le cas contraire, il faut préciser à quelle variable on se réfère lorsqu'on projette sur un attribut. On utilise pour cela la **notation pointée**. Elle est nécessaire par exemple pour effectuer une jointure naturelle. Ainsi, pour avoir la liste des parents et des écoles de leurs enfants, on pose la requête :

```
select Parent, Ecole from Parenté, Scolarité where Scolarité.Enfant = Parenté.Enfant
```

Deux variables peuvent parcourir la même relation. Le nom de relation n'est alors pas suffisant pour déterminer la variable. SQL donne la possibilité de **renommer une variable**. Il suffit pour cela de préciser le nouveau nom de la variable après le nom de la relation correspondante au sein de la clause `from`. Le renommage sert de raccourci syntaxique lorsque les noms de relation sont longs. Il est indispensable par exemple pour effectuer une auto-jointure. Ainsi, pour avoir la liste des grand-parents de Zoé, on pose la requête :

```
select t.Parent from Parenté t, Parenté s where s.Enfant = "Zoé" and t.enfant = s.parent
```

2.4.1.3 Présentation des résultats

- **Renommage des colonnes** : Par défaut, les résultats sont présentés sous forme de table, les têtes de colonne correspondant à la notation utilisée dans la clause `select`. Par exemple, la requête `select t.Parent from Parenté t, Parenté s where s.Enfant = "Zoé" and t.enfant = s.parent` qui donne les grand-parents de Zoé renverra le résultat sous la forme :

t.Parent
Marcel

Pour obtenir un entête de colonne plus "parlant", il suffit de renommer l'attribut de sortie dans la clause `select`. Par exemple, la requête ainsi modifiée

```
select t.Parent as GPdeZoé from Parenté t, Parenté s where s.Enfant = "Zoé" and t.enfant = s.parent
```

renverra le résultat sous la forme :

GPdeZoe
Marcel

- **Tri des n-uplets résultats** : Pour ordonner les résultats, on utilise une clause spécifique `order by`.
 - La syntaxe est `order by <spécif. de tri1>,<spécif. de tri2>, ...`
 - Une spécification de tri `<specif de tri >` précise quel attribut (colonne) doit être utilisée pour le tri et si le tri doit être ascendant (ASC) ou descendant (DESC).
 - Lorsqu'il y a plusieurs spécifications de tri, le tri s'effectue d'abord selon la première spécification puis, si deux n-uplets sont égaux pour la première spécification, ils sont ordonnés selon la deuxième spécification, etc.
 - Par exemple, si on veut trier les personnes d'abord par sexe, puis par âge, on posera la requête :
`select Personne, Age, Sexe from Descriptif order by Sexe ASC, Age DESC`

Le résultat sera :

Personne	Age	Sexe
Clara	27	F
Tata	18	F
Zoe	2	F
Johnny	65	M
Marcel	60	M
Raymond	40	M
Toto	40	M
Titi	20	M

2.4.2 Agrégats

Contrairement aux langages théoriques, SQL comportent un certain nombre de fonctions qui permettent de faire des calculs sur l'ensemble des valeurs d'une ou plusieurs colonnes : les **agrégats**

- `count(DISTINCT att1,att2,...)` renvoie le nombre de n-uplets résultats ayant des valeurs différentes pour `att1,att2,...`. Cas particulier : `count(*)` renvoie le nombre de n-uplets résultats.
- `sum(att)` renvoie la somme des valeurs de `att` des n-uplets résultats. `sum(distinct att)` renvoie la somme des valeurs différentes de `att` des n-uplets résultats.
- idem pour `avg(att)` et `avg(distinct att)` : moyenne arithmétique.
- idem pour `max(att)` et `min(att)` : valeur maximum et minimum.

2.4.2.1 Agrégats globaux

L'utilisation d'agrégat sur l'ensemble des n-uplets résultats se fait en déclarant l'agrégat dans la clause `select`. Évidemment, il est interdit de demander en résultat à la fois des agrégats (sur l'ensemble des n-uplets) et des attributs (individuellement par n-uplet).

Par exemple, la requête `select count(*) from Descriptif` renvoie le nombre de personnes dans la base. La requête `select count(distinct Parent) from Parenté` renvoie le nombre de parents de la base (le `distinct` est nécessaire car un parent peut apparaître plusieurs fois s'il a plusieurs enfants). Enfin, pour savoir combien Marcel a de filles, on pose la requête

```
select count(distinct Enfant)
from Parenté, Descriptif
where Parent="Marcel" and Sexe='F' and Enfant=Personne.
```

2.4.2.2 Agrégats partitionnés

SQL offre la possibilité de faire des groupes dans les n-uplets résultats afin de calculer les agrégats "groupe par groupe". On utilise pour cela la clause `group by att1, att2, ...` qui regroupe les n-uplets ayant même valeur pour `att1, att2, ...` et effectue le calcul de l'agrégat sur un autre attribut. La requête aura donc la forme :

```
select att1, att2, ... AGREGAT(att)
from ...
where ...
group by att1, att2, ...
```

Le résultat sera un ensemble de n-uplets (1 par groupe) qui donne pour chaque groupe, la valeurs des attributs de partitionnement (`att1, att2, ...`) ainsi que le résultat de l'agrégat pour le groupe considéré.

On peut aussi filtrer les groupes qui vérifient une certaine condition sur la valeur de l'agrégat, en rajoutant une clause `having condition(agrégat)`.

Par exemple, si on veut la liste des moyennes d'âge par sexe, si cette moyenne est supérieure à 20, on pose la requête :

```
select Sexe, Avg(Age)
from Descriptif
group by Sexe
having Avg(Age) > 20
```

Dans notre exemple, la moyenne d'âge des filles étant de 15,7 et celle des garçons étant de 45, le résultat sera le suivant :

Sexe	Avg(Age)
M	45

2.4.3 Requêtes imbriquées

Pour effectuer des requêtes plus complexes, il est nécessaire de savoir tout ce qu'on peut mettre dans la clause where F.

La condition F de la clause where peut être très complexe. Elle se construit à l'aide des connecteurs logiques and, or, not respectivement pour le ET logique, le OU logique et la négation, et des expressions de base suivantes :

- $Exp1 \Theta Exp2$ où Θ est un comparateur et $Exp1$ et $Exp2$ sont des expressions formées à l'aide de constantes, de noms d'attributs et d'opérateurs (+, -, ...). C'est ce qui est utilisé pour effectuer les requêtes simples.
- $Exp1 \Theta (sous - requete)$ où *sous - requete* est une requête SQL ne renvoyant qu'une seule valeur. *sous - requete* est donc :
 - soit une requête renvoyant un agrégat global. Par exemple la requête suivant renvoie les gens plus âgés que la moyenne :
 select Personne from Descriptif where Age > (select Avg(Age) from Descriptif)
 - soit une sous-requête quantifiée par any ou all. Dans le cas du any (resp. all), le comparateur renverra vrai si il existe une valeur *val* dans le résultat de la sous-requête telle que (resp. si quelque soit la valeur *val* dans le résultat de la sous-requête on a) $Exp1 \Theta val$

Par exemple, la requête suivante renvoie les personnes plus âgées que toutes les filles.

```
select Personne from Descriptif where Age > all (select D.Age from Descriptif where Sexe ='F')
```

Il est important de noter que la sous-requête doit renvoyer une seule valeur *par définition (structurellement)* et non pas “par hasard”. Ainsi, la requête suivante sera **refusée par l'interpréteur de requête** car elle compare une valeur avec un ensemble :

```
select Personne from Descriptif where Age > (select D.Age from Descriptif where Personne = "Marcel")
```

- $Exp1$ [not] in (*sous - requête*) : renvoie vrai ssi la valeur renvoyée par $Exp1$ appartient (n'appartient pas dans le cas de not in) à l'ensemble de valeurs renvoyé par la sous-requête.

Par exemple, la requête suivante renvoie les personnes n'ayant pas d'enfant :

```
select Personne from Descriptif where Personne not in (select Parent from Parenté)
```

- [not] exists (*sous - requête*) : renvoie vrai ssi la sous-requête renvoie un ensemble non-vide (vide dans le cas du not exists)

Par exemple, pour obtenir les personnes ayant des (demi-)frères ou sœurs, on pose la requête :

```
select Enfant from Parenté P
where exists (select * from Parenté where P.Parent = Parent and P.Enfant < > Enfant)
```

Notons que comme SQL ne possède pas de quantificateur universel, le seul moyen de faire une division est de passer par un double not exists (ou not in dans certains cas). Ainsi pour obtenir la liste des parents ayant un enfant scolarisé dans chaque école de la base (qui correspond à une division comme nous l'avons vu au chapitre 2.2), on pose la requête suivante, qui cherche les parents tels que il n'y a pas d'école dans la base telle qu'ils n'ont pas au moins un enfant scolarisé dans cette école, soit :

```
select Parent from Parenté P1 where not exists
(select * from Scolarité S1 where not exists
(select * from * Scolarité S2, Parenté P2 where S1.Ecole = S2.Ecole and S2.Enfant = P2.Enfant
and P2.Parent = P1.Parent))
```

- $Exp1$ like *motif*, où $Exp1$ renvoie une valeur de type chaîne de caractères et *motif* est une expression régulière qui indique la condition que doit respecter cette chaîne. *motif* est formé à partir de caractères constants et des caractères spéciaux suivants :

- '_' indique n'importe quel caractère
- '%' indique n'importe quelle chaîne de caractères

Ainsi, la requête suivante renvoie la liste des enfants dont le nom des parents commence par 'T' et finit par 'to' :

```
select Enfant from Parenté where (Parent like "T%") and (Parent like "%to")
```

Chapitre 3

Manipulation et contraintes d'intégrité

3.1 Définition et manipulation de données en SQL

Dans cette section et la suivante, nous utiliserons la base de données exemple d'un club de voile suivante :

- Bateau(*b_id*, *b_nom*, *type*, *couleur*) qui donne l'identificateur, le nom et la couleur d'un bateau.
- Marin(*m_id*, *m_nom*, *niveau*, *age*) qui donne le numéro de membre, le nom, le niveau en voile et l'âge d'un marin du club.
- Résa(*m_id*, *b_id*, *jour*) qui indique quel marin a réservé quel bateau et pour quel jour.

3.1.1 Gestion de schéma en SQL

La création d'une table se fait par la commande :

```
create table nom_table(nom_att1 : domaine, nom_att2 : domaine,...)
```

La suppression d'une table se fait par la commande :

```
drop table nom_table
```

La modification d'une table se fait par la commande :

```
alter table nom_table ...
```

On peut ajouter, supprimer ou modifier un attribut de la table. Si la table contenait déjà des nuplets, cela peut poser des problèmes.

3.1.2 Manipulation de données en SQL

3.1.2.1 Insertion de nuplets

SQL permet d'insérer le nuplet soit individuellement, en fournissant les valeurs des différents attributs, soit d'insérer directement un ensemble de nuplets lorsque les valeurs proviennent de nuplets déjà existants.

Insertion individuelle

L'insertion individuelle d'un nuplet (*v1*, *v2*, ..., *vn*) dans la relation $R(A_1, A_2, \dots, A_n)$ se fait à l'aide de la commande :

```
insert into R(A1, A2, ... An) values (v1, v2, ..., vn)
```

Le fait d'explicitier la liste des attributs de R permet de spécifier dans quel ordre les attributs vont être donnés. Cet ordre peut donc être différent de celui spécifié lors de la création de la table.

Exemple : `insert into Marins(m_nom, m_id, age, niveau) values ("Toto", 12, 21, 5)`

Insertion multiple

L'insertion multiple consiste à insérer dans une table $R(A_1, A_2, \dots, A_n)$ le résultat d'une requête. Le problème est souvent que tous les attributs de la table ne proviennent pas de nuplets déjà existants, il faut alors rajouter une valeur *null* dans la clause `select`.

Par exemple, si on a dans la base une table `Etudiant(e_id, e_nom, age)` et qu'on décide que tous les étudiants de plus de 18 ans sont membre du club de voile, leur numéro d'étudiant leur servant de numéro de membre (hypothèse peu réaliste), on utilisera la commande suivante. Comme on ne connaît pas le niveau des étudiants, on est obligé de mettre une valeur nulle :

```
insert into Marins(m_nom, m_id, age, niveau)
select e_nom, e_id, age, null
from Etudiant
where age >= 18
```

3.1.2.2 Modification de nuplets

La modification de nuplets s'obtient par la commande :

```
update R set R.att = f(R) where F
```

tel que :

- R est une relation de la base et désigne donc le nuplet à modifier
- att est un attribut de R
- f(R) est la nouvelle valeur de att. C'est une expression formée à partir des attributs du nuplet, de constantes et d'opérateurs
- F est la formule permettant de filtrer les nuplets de R à modifier

Par exemple, si on veut diminuer le niveau de tous les membres qui ont moins de 15 ans, on lance la commande :

```
update Marins M set M.niveau = M.niveau - 1 where M.age < 15
```

3.1.2.3 Suppression de nuplets

La suppression de nuplets s'obtient par la commande :

```
delete from R where F
```

tel que :

- R est une relation de la base et désigne donc le(s) nuplet(s) à supprimer
- F est la formule permettant de filtrer les nuplets de R à supprimer

Par exemple, si on veut supprimer les membres pour lesquels on ne connaît pas le niveau, on lance la commande :

```
delete from Marins M where M.niveau is null
```

On s'aperçoit que cette opération risque de poser des problèmes d'intégrité dans la base. En effet, si parmi les marins supprimés, certains avaient déjà effectué des réservations, on ne peut pas les supprimer. Pour gérer ce genre de situation et empêcher des modifications, insertions ou suppressions malencontreuses, il est nécessaire de définir des *contraintes d'intégrité* sur la base de données. C'est l'objet de la section suivante.

3.2 Contraintes d'intégrité en SQL

3.2.1 Généralités sur les contraintes d'intégrité

Définition 3.2.1 Une *contrainte d'intégrité* est une règle qui définit quels sont les états (ou instances) de la base de données qui sont cohérentes, c'est-à-dire qui respectent la sémantique de l'application.

On distingue les **contraintes statiques** qui peuvent être vérifiées sur une seule instance de la base, et les **contraintes dynamique** qui nécessitent de considérer plusieurs instances de la base pour être évaluées. Dans la suite, nous ne considérerons que des contraintes statiques. □

Comme on le voit, la définition précédente inclut un grand nombre de concepts. Par exemple, le modèle de données relationnel peut être vu lui-même comme un ensemble de contraintes, puisqu'un nuplet d'une relation doit correspondre à son schéma : les mêmes attributs et le même domaine que l'attribut du schéma pour chaque attribut. La clé d'une relation est aussi une contrainte puisqu'elle impose à deux nuplets d'une relation d'avoir des valeurs différentes pour la clé.

Les opérations qui peuvent violer une contrainte d'intégrité sont, selon les cas, l'insertion, la suppression ou la modification de nuplets. On peut vérifier les contraintes après chaque opération. Dans ce cas se pose le problème qu'on risque de refuser une opération alors que l'opération suivante aurait permis de "réparer" la contrainte. C'est pourquoi on préfère parfois grouper les opérations et ne vérifier les contraintes qu'à la fin de l'exécution du groupe (ou transaction). Dans le cas où une des contraintes est violée, on peut soit annuler l'opération (ou le groupe d'opérations) et revenir à l'état précédent, qui par définition est supposé

vérifier toutes les contraintes, ou alors, dans certains cas seulement, exécuter des opérations supplémentaires prédéfinies qui vont réparer la contrainte.

Dans la suite de ce chapitre, nous voyons quelles contraintes peuvent exister dans une base de données relationnelle, comment les définir en SQL lorsqu'on crée le schéma de la base de données, et, dans le cas des contraintes référentielles, comment indiquer au système l'action à effectuer en cas de violation.

3.2.2 Typologie des contraintes dans le modèle relationnel

Les différentes contraintes en SQL peuvent se classer selon le nombre d'attributs, de nuplets et de tables qu'elles impliquent.

3.2.2.1 Contraintes de domaine

Une contrainte de domaine implique un attribut d'un nuplet d'une table.

Elle définit quelles valeurs peut prendre cet attribut : son type de données et des restrictions sur ce type (bornes inf. et sup., si l'attribut peut prendre une valeur *null*). SQL permet, en outre, de définir une valeur par défaut pour l'attribut qui sera attribuée à la création d'un nuplet de la table si aucune valeur n'est spécifiée (notamment dans le cas d'insertion multiple).

La syntaxe, utilisée lors de la commande `create table` est la suivante (les crochets indiquent des options) :

```
create table nom_table (nom_att type [check value ...] [not null] | [default valdef]...)
```

où `check value` est suivi d'une restriction sur la valeur de l'attribut (ex. `check value < 12`). Cette restriction peut être décrite en fin du `create table`, il faut alors remplacer `value` par le nom de l'attribut. `not null` indique que la valeur doit être absolument renseignée à la création d'un nuplet. La valeur *valdef* peut être soit une constante, soit autoincrément qui retourne la valeur maximum existante pour cet attribut + 1, `current date` qui retourne la date système...

Si plusieurs attributs ont les mêmes contraintes de domaine, on peut créer un domaine nommé grâce à la commande `create domaine nom_domaine type ...` qui peut ensuite être utilisé dans le `create table` en lieu et place du type d'un attribut.

3.2.2.2 Contrainte de clé

Une contrainte de clé implique un ou plusieurs attributs de plusieurs nuplets d'une seule table.

Elle définit un groupe d'attributs qui permet d'identifier les nuplets de la table : deux nuplets ne peuvent avoir la même valeur pour une clé. Il peut exister plusieurs clés sur une table. On distingue la **clé primaire** qui va permettre d'organiser physiquement la relation et qui est introduite par le mot-clé `primary key`. Les autres clés sont introduites par le mot-clé `unique`.

Syntaxiquement, lorsqu'une clé est composée d'un seul attribut, on peut la déclarer en faisant suivre la définition de l'attribut par le mot-clé correspondant. Sinon, on rajoute en fin du `create table` la définition de la clé `primary key(att1, att2, ...)` (idem pour `unique`).

3.2.2.3 Contraintes de table

Les contraintes de table généralisent les deux types de contrainte précédentes. Elles impliquent plusieurs nuplets d'une même table. Syntaxiquement, une contrainte de table est introduite en fin de `create table` par le mot-clé `[constraint nom_contrainte] check (formule)`. La formule à vérifier peut être d'une complexité variable selon les systèmes.

3.2.2.4 Contraintes d'intégrité référentielle

Ces contraintes, qui impliquent des attributs de nuplets de deux tables sont très utiles dans le modèle relationnel où les liens entre tables se font grâce à des valeurs d'attributs qui doivent être égales (jointure).

Une contrainte d'intégrité référentielle définit qu'un attribut (ou groupe d'attributs) ne peut prendre comme valeur qu'une valeur existant comme clé (primaire ou non) d'une autre relation. En d'autres termes, elle interdit à un nuplet d'une relation de référencer un nuplet inexistant dans une autre relation.

Syntaxiquement, elle s'introduit en fin de `create table` par :

```
foreign key (att1, att2, ...) references nom_table_référencée[(att'1, att'2, ...)]
```

Lorsque les attributs référencés (`att'1`, `att'2`, ...) forment la clé primaire de *nom_table_référencée*, on peut omettre de les préciser. Lorsque la contrainte ne s'applique qu'à un seul attribut, elle peut être introduite directement en fin de définition de l'attribut.

3.2.2.5 Un exemple complet

L'exemple suivant illustre les différents types de contraintes vus jusqu'ici, en créant les 3 tables de notre club de voile :

- `create domain Coul` check value in ("bleu", "blanc", "vert", ...)

Le domaine `Coul` permet de vérifier que la couleur d'un bateau sera bien une chaîne de caractères représentant une couleur. On peut maintenant l'utiliser dans la création de la table `Bateau`.

- `create table Bateau` (

- `b_id` Integer default autoincrement primary key,

- `b_nom` Char(10) unique,

- `type` Char(10),

- `couleur` Coul

- `check` (not exists (select * from Bateau B where type = B.type and couleur <> B.couleur)))

On note que la table comporte une clé primaire (`b_id`) et une autre clé (`b_nom`). On note aussi qu'il ne peut pas exister deux bateaux ayant le même type et des couleurs différentes : on dit que le type d'un bateau détermine sa couleur.

- `create table Marin` (

- `m_id` Integer default autoincrement primary key,

- `m_nom` Char(20) not null unique,

- `niveau` Integer,

- `age` Integer check value < 120

- `check` niveau < (age / 10))

On note que le niveau d'un marin ne peut dépasser le dixième de son âge.

- `create table Résa` (

- `m_nom` Char(20) ,

- `b_id` Integer,

- `jour` Date,

- `primary key` (b_id, jour)

- `unique`(m_nom)

- `foreign key` (b_id) references Bateau

- `foreign key` (m_nom) references Marin(m_nom))

On note qu'un bateau ne peut être réservé que s'il existe (sic) et par un membre existant du club. Par ailleurs, la définition de la clé primaire et de la deuxième clé nous indiquent qu'un même bateau ne peut être réservé qu'une seule fois pour la même journée et qu'un marin ne peut avoir en cours qu'une seule réservation. Cette situation est différente de celle où un marin ne peut effectuer qu'une seule réservation pour une journée donnée : dans ce cas on aurait une seule clé (`b_id`, `jour`, `m_nom`)

3.2.2.6 Contraintes générales sur plusieurs tables

Les contraintes générales sur plusieurs tables ne peuvent pas être incluses dans la définition d'une table. Elles impliquent un nombre quelconque de nuplets, attributs, relations. Elles sont donc à utiliser avec précaution car elles peuvent être très longues à évaluer. Syntaxiquement, elles sont introduites par la commande suivante :

```
create assertion nom_contrainte check formule
```

où *formule* est exprimée comme une requête booléenne SQL.

Par exemple, la contrainte suivante exprime que le club de voile doit rester de dimension raisonnable :

```
create assertion PetitClub
check( (select count(*) from Marin + select count(*) from Bateau) < 100)
```

3.2.3 Maintenance des contraintes

Comme on l'a vu en 3.2.1, le système doit garantir que les contraintes sont toujours vérifiées, soit directement après chaque opération, soit après un groupe d'opérations. La stratégie dépend du type de la contrainte :

- pour une contrainte de domaine, il est évident que la vérification doit se faire au moment de l'opération (modification ou insertion de nuplet) et que celle-ci doit être refusée si elle ne respecte pas le domaine.
- pour une contrainte de clé ou une contrainte de table, on peut soit tester la contrainte après chaque opération sur la table, soit tester en fin de transaction (groupe d'opérations). Dans le premier cas, on peut refuser exactement l'opération qui viole la contrainte au plus tôt. Dans le deuxième cas, il faudra annuler tout le groupe mais on peut espérer que celui-ci contienne une opération qui va réparer la contrainte. Par exemple, si dans le même groupe on insère un nuplet avec une valeur de clé déjà existante mais qu'ensuite on supprime le nuplet qui avait préalablement cette valeur de clé, le groupe d'opération ne viole pas la contrainte.
- pour une contrainte générale, la meilleure stratégie est de ne tester qu'à la fin de l'exécution du groupe.
- le cas des contraintes d'intégrité référentielle est plus intéressant car on peut prédéfinir des opérations qui peuvent réparer la contrainte, selon que l'action fautive est une suppression ou bien une modification. SQL propose d'expliciter la stratégie à suivre, après la définition du FOREIGN KEY :
 - on delete (resp. on update) indique que la stratégie concerne une violation dûe à une suppression (resp. une modification)
 - ensuite on a la choix entre :
 - restrict (stratégie par défaut). L'action fautive est refusée. On peut néanmoins attendre la fin du groupe d'opérations en ajoutant la clause check on commit.
 - cascade. Dans le cas d'une suppression, les nuplets qui référençaient le nuplet supprimé sont automatiquement supprimés. Dans le cas d'une modification, celle-ci est propagée vers les nuplets qui référençaient le nuplet modifié.
 - set default. Dans les deux cas, le ou les attributs référençant des nuplets qui référençaient le nuplet supprimé ou modifié sont mis à leur valeur par défaut.
 - set null. Dans les deux cas, le ou les attributs référençant des nuplets qui référençaient le nuplet supprimé ou modifié sont mis à la valeur *null*.

Chapitre 4

Théorie de la conception de bases de données relationnelles

4.1 Conception d'un schéma de base de données relationnelle

4.1.1 Généralités

Dans les chapitres 1.2 et 2.1 nous avons vu comment établir un schéma de base de données relationnelle *de manière intuitive* : par l'observation du monde réel et dialogue avec les autres intervenants, on établit un schéma Entité-Association décrivant les objets du monde réel et les liaisons existant entre ces objets. Ce schéma entité-association est ensuite traduit automatiquement en un schéma relationnel, dans lequel le concepteur doit ensuite intégrer les contraintes d'intégrité qui ne pouvaient être modélisées auparavant. Le problème est que l'ajout de certaines contraintes d'intégrité, comme il intervient après la définition des différents schémas de relation de la base, peut révéler des problèmes de conception qui nécessitent une refonte du schéma de la base.

Dans cette partie, nous étudierons ce problème de manière théorique pour une catégorie très importante de contraintes d'intégrité : les dépendances fonctionnelles. Nous verrons quelles anomalies elles peuvent mettre en exergue dans un schéma relationnel fait de façon intuitive, comment on les définit, comment on peut les mettre en forme pour qu'elles soient acceptées par des processus d'amélioration du schéma. Enfin, nous caractériserons la qualité d'un schéma par rapport aux dépendances fonctionnelles (les formes normales) et verrons comment obtenir le "meilleur" schéma possible à partir d'un schéma quelconque.

4.1.2 Anomalies d'un schéma relationnel

Pour montrer qu'un schéma relationnel fait de manière intuitive peut comporter des anomalies, nous nous basons sur le schéma S1 suivant, qui modélise les informations d'une entreprise sur ses fournisseurs :

S1 : FournisseursProduits(NomFournisseur, Adresse, Produit, Prix)

Les attributs décrivent respectivement le nom du fournisseur, son adresse, un produit qu'il fournit et le prix auquel il fournit ce produit. La clé est donc formée par les attributs `NomFournisseur` et `Produit`.

4.1.2.1 Redondance d'information

Le principal problème avec le schéma S1 est qu'il permet des *redondances* (d'information). En effet, supposons que le fournisseur "Toto" fournisse 1000 produits différents. Il s'en suit que la relation `FournisseursProduits` va contenir 1000 nuplets correspondants. Or ces 1000 nuplets contiennent chacun le nom et l'adresse de "Toto". Cette redondance, qui vient du fait qu'un fournisseur n'a qu'une seule adresse, est à ne pas confondre avec la duplication d'une valeur d'attribut, qui est nécessaire lorsque cette valeur doit être associée à plusieurs valeurs différentes d'un autre attribut. Ainsi, si le fournisseur "Toto" pouvait avoir plusieurs adresses différentes, le schéma S1 ne serait pas redondant.

La redondance d'information rend la gestion de la base de données difficile car elle n'optimise pas l'espace de stockage de la base de données mais surtout car elle est source d'anomalies.

4.1.2.2 Différentes anomalies liées à la redondance

La redondance d'information dans un schéma relationnel engendre des possibilités d'anomalie pour les différentes opérations possibles sur une relation : insertion, modification, suppression.

- Anomalie de modification : imaginons que le fournisseur "Toto" déménage. Il est alors nécessaire de modifier tous les nuplets de S1 où il figure. Si certains sont oubliés, il s'en suivra que la base ne sera pas cohérente puisqu'elle contiendra certains nuplets avec la nouvelle adresse de "Toto" et certains avec son ancienne adresse.
- Anomalie d'insertion : supposons qu'on veuille entrer dans la base un nouveau fournisseur pour lequel on ne connaît pas encore les produits qu'il va fournir ni les tarifs qu'il va pratiquer. On pourrait envisager de créer un nuplet avec uniquement le nom et l'adresse de ce fournisseur, en laissant les autres attributs avec une valeur *null* mais cela est impossible car l'attribut `Produit` fait partie de la clé de la relation.

- Anomalie de suppression : si, pour une raison quelconque, on supprime tous les produits d'un fournisseur, on perdra aussi son adresse, ce qui n'est peut-être pas le résultat escompté si on désire refaire appel à ce fournisseur ultérieurement.

4.1.3 Changer de schéma

S1 n'est pas un bon schéma car il permet des redondances d'information. Un meilleur schéma, qui résout les problèmes énoncés plus haut serait le schéma S2 suivant :

S2 : Fournisseurs(NomFournisseur, Adresse)
Produits(NomFournisseur, Produit, Prix)

Le but des chapitres suivants est d'étudier de manière théorique comment passer de S1 à S2, en d'autres termes :

- Comment détecter les redondances possibles. Celles-ci sont principalement dues à un classe de contraintes d'intégrité : les *dépendances fonctionnelles*, comme par exemple, le fait qu'un fournisseur ne peut avoir qu'une seule adresse. Nous verrons comment regrouper et mettre en forme l'ensemble des dépendances fonctionnelles qui s'appliquent à un schéma donné.
- Comment passer d'un schéma relationnel à un autre qui modélise autant que possible le même monde réel, grâce aux notions de *décomposition de schéma sans perte d'information et sans perte de dépendances*.
- Comment caractériser les schémas relationnels comportant le moins de redondances possible, grâce à la notion de *forme normale* et comment décomposer un schéma quelconque vers le meilleur schéma possible.

4.2 Dépendances fonctionnelles dans un schéma relationnel

4.2.1 Rappel

Un schéma relationnel R peut avoir plusieurs instances $r(R), r'(R), \dots$. Les contraintes d'intégrité sont des propriétés que toute instance doit satisfaire, elles font donc partie du schéma. Parmi les contraintes, certaines portent sur plusieurs valeurs à la fois et peuvent introduire des redondances : puisqu'on sait que la contrainte est satisfaite, on peut déduire de la valeur d'un ou plusieurs attributs la valeur d'un autre attribut. Le cas le plus flagrant est celui des *dépendances fonctionnelles*, comme dans le cas vu précédemment : comme un fournisseur ne peut avoir qu'une seule adresse, deux nuplets pour le même fournisseur vont fournir la même information, donc il y aura de la redondance. Dans la suite de cette section, on formalise la notion de dépendance fonctionnelle et on voit comment mettre en forme un ensemble de dépendance fonctionnelles.

4.2.2 Notion de dépendance fonctionnelle (DF)

4.2.2.1 Notations

Dans la suite, nous utilisons les notations suivantes :

- X, Y, Z, \dots désignent des ensembles d'attributs.
 U désigne l'univers des attributs, c'est-à-dire l'ensemble des attributs du schéma
- A, B, C, \dots désignent des attributs
- XY désigne l'ensemble d'attributs $X \cup Y$, AB désigne l'ensemble d'attributs $\{A, B\}$
- f, g, h, \dots désignent des dépendances fonctionnelles.
- F, G, H, \dots désignent des ensembles de dépendances fonctionnelles.
- R, S, T, \dots désignent des schémas de relation.
- r, s, t, \dots désignent des relations. $r(R)$ exprime que la relation r a pour schéma R .
- t, t', t'', \dots désignent des nuplets.
- $t.A$ désigne la valeur de l'attribut A pour le nuplet t . Si $X = ABC\dots$, $t.X$ désigne le nuplet $(t.A, t.B, t.C, \dots)$

4.2.2.2 Définition

Définition 4.2.1 Soient R un schéma de relation et X et Y des sous-ensembles de R .

On dit qu'il existe une **dépendance fonctionnelle** de X vers Y (ou bien que X détermine fonctionnellement Y), qu'on note $X \rightarrow Y$ si :

$$\forall r, r(R), \quad r \text{ "satisfait" } X \rightarrow Y, \text{ c'est à dire : } \forall t, t' \in r, (t.X = t'.X) \Rightarrow (t.Y = t'.Y)$$

□

Cette définition peut s'interpréter ainsi : lorsque je connais la valeur d'un nuplet pour X , alors je connais sa valeur pour Y puisqu'il n'y en a qu'une seule possible.

Notons que cette définition contraint tout couple de nuplet de toute instance de R . Une dépendance fonctionnelle fait donc partie du schéma et ne peut donc pas être découverte à partir d'une instance particulière. Il est donc nécessaire, comme pour la modélisation du schéma sous forme de table (ou sous forme entité-association), de se référer au monde réel et à ceux qui en connaissent les règles. Par exemple, on peut s'interroger si, sur le schéma S2 on a la dépendance fonctionnelle $\text{Produit} \rightarrow \text{NomFournisseur}$, ou alors si un même produit peut être fourni par plusieurs fournisseurs différents.

4.2.2.3 Cas particuliers

Notons tout de même un cas particulier déjà bien connu, celui des *surclés*. En effet, une surclé X d'un schéma de relation R est telle que deux nuplets différents ne peuvent avoir la même valeur, on a donc la dépendance fonctionnelle $X \rightarrow R$ (lorsque je connais la valeur de la surclé, alors je connais la valeur du nuplet, donc de tous les attributs).

Un autre cas particulier est celui des *dépendances triviales*, qui sont de la forme $XY \rightarrow Y$ et sont vérifiées quels que soient X et Y .

4.2.2.4 Évaluation d'une dépendance fonctionnelle

Par définition, une dépendance $X \rightarrow Y$ peut être imposée au schéma selon trois cas possibles :

- X et Y sont tous deux inclus dans un schéma de relation R . Dans ce cas, la dépendance ne peut être violée que lorsqu'on insère un nouveau nuplet dans une relation de schéma R ou lorsqu'on modifie la valeur d'un nuplet d'une telle relation. Il suffit dans les deux cas de rechercher dans la relation les nuplets ayant la même valeur pour X que le nuplet inséré ou modifié. Cette évaluation peut, surtout si la relation est indexée sur X , se faire en temps raisonnable.
- X et Y sont chacun inclus dans des schémas de relation différents. Pour évaluer la dépendance, il faut surveiller les insertions et modifications dans deux relations et joindre les nuplets correspondants, ce qui, dans le cas général, ne peut pas s'effectuer en temps raisonnable.
- X ou Y n'est pas inclus dans un seul schéma de relation. L'évaluation de la dépendance est encore plus compliquée que dans le cas 2.

En conclusion, il est nécessaire, pour qu'on puisse évaluer une dépendance fonctionnelle $X \rightarrow Y$, que X et Y soient inclus dans un même schéma de relation. Ceci justifie que la définition 4.2.1 soit basée sur un schéma de relation R .

4.2.3 Calcul des dépendances fonctionnelles

Comme toute contrainte d'intégrité, une dépendance fonctionnelle est une formule logique. On peut donc, à partir d'un ensemble de dépendances fonctionnelles exprimées par les concepteurs de la base de données, inférer d'autres dépendances fonctionnelles qui sont des conséquences logiques des premières. Par exemple, à partir d'un ensemble vide de dépendance fonctionnelle, on pourra déduire uniquement des dépendances triviales.

4.2.3.1 Implication logique de DF

Définition 4.2.2 Soit F un ensemble de dépendances fonctionnelles sur un ensemble d'attributs U .

On dit que F **implique logiquement** $f : X \rightarrow Y$ (avec $XY \subset U$) qu'on note $F \models f$ si :

$$\forall r, r(R) \text{ et } R \subset U, \quad (\forall g \in F, r \text{ satisfait } g) \Rightarrow (r \text{ satisfait } f)$$

□

Par exemple, si $A \rightarrow B$ et $B \rightarrow C$, il est aisé de montrer que $A \rightarrow C$. Ce procédé permet d'exhiber ainsi toutes les dépendances fonctionnelles qui seront vérifiées lorsqu'on maintient un ensemble donné de DF. Bien entendu, on peut ensuite répéter le processus jusqu'à trouver toutes les dépendances fonctionnelles impliquées (on dit aussi déduites) par un ensemble de dépendances fonctionnelles, c'est la notion de *fermeture transitive d'un ensemble de dépendances fonctionnelles*

Définition 4.2.3 On appelle *fermeture transitive d'un ensemble F de dépendances fonctionnelles* l'ensemble F^+ de toutes les dépendances fonctionnelles qu'on peut déduire de F , soit :

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

□

Par exemple, soit $U = \{A, B, C\}$ et $F = \{A \rightarrow B, B \rightarrow C\}$, la fermeture transitive de F vaut $F^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C, A \rightarrow A, A \rightarrow AB, A \rightarrow BC, A \rightarrow ABC, A \rightarrow AC, AB \rightarrow A, AB \rightarrow AB, AB \rightarrow BC, AB \rightarrow ABC, ABC \rightarrow A, ABC \rightarrow AB, ABC \rightarrow BC, ABC \rightarrow AC, ABC \rightarrow ABC, B \rightarrow B, B \rightarrow BC, BC \rightarrow B, BC \rightarrow BC, C \rightarrow C, BC \rightarrow C\}$

On voit que, même sur un exemple de très petite taille, le calcul de la fermeture transitive d'un ensemble de DF peut être très long et coûteux (exponentiel). Fort heureusement, il sera toujours par la suite possible d'éviter de calculer entièrement une telle fermeture. Néanmoins, il est nécessaire de pouvoir déduire des dépendances fonctionnelles à partir d'un ensemble de DF donné. Trois possibilités sont offertes :

1. Considérer une dépendance à déduire et chercher à démontrer directement qu'elle peut se déduire à partir de l'ensemble donné. Par exemple, on peut montrer que $\{A \rightarrow B, B \rightarrow C\} \models A \rightarrow C$ en revenant à la définition des DF.
En effet, de $A \rightarrow B$ on sait que [1] $\forall r, r(R) \text{ et } R \subset U, \quad \forall t, t' \in r, (t.A = t'.A) \Rightarrow (t.B = t'.B)$.
De même, de $B \rightarrow C$ on sait que [2] $\forall r, r(R) \text{ et } R \subset U, \quad \forall t, t' \in r, (t.B = t'.B) \Rightarrow (t.C = t'.C)$.
De [1] et [2] on déduit aisément que,
 $\forall r, r(R) \text{ et } R \subset U, \quad \forall t, t' \in r, \left((t.A = t'.A) \Rightarrow (t.B = t'.B) \right) \wedge \left((t.B = t'.B) \Rightarrow (t.C = t'.C) \right)$. Par une simple transformation logique on trouve $\forall r, r(R) \text{ et } R \subset U, \quad \forall t, t' \in r, (t.A = t'.A) \Rightarrow (t.C = t'.C)$ ou encore, $A \rightarrow C$, cqfd.
2. En utilisant le calcul, moins coûteux, de la *fermeture transitive d'un ensemble d'attributs* (cf. 4.2.3.2).
3. En utilisant les axiomes d'Armstrong (cf. 4.2.3.3)

4.2.3.2 Fermeture transitive d'un ensemble d'attributs

La fermeture transitive d'un ensemble d'attributs donné *par rapport à un ensemble de DF* est formée par tous les attributs déterminés fonctionnellement par les attributs de l'ensemble donné, directement ou indirectement, grâce aux dépendances fonctionnelles données.

Définition 4.2.4 Soit X un ensemble d'attributs de U et F un ensemble de dépendances fonctionnelles sur U , on appelle *fermeture de X par rapport à F* et on note $[X]_F^+$ l'ensemble suivant :

$$[X]_F^+ = \{A \in U \mid F \models X \rightarrow A\}$$

□

L'algorithme pour calculer $[X]_F^+$ est très simple. Il consiste à partir de X , de trouver toutes les DF $Y \rightarrow Z$ qui s'appliquent à X (c'est-à-dire telles que $Y \subset X$) et d'inclure à chaque fois Z dans la fermeture, puisque lorsqu'on connaît X , on connaît Z grâce à la DF appliquée. Comme le processus est expansif, il faut

à chaque fois qu'on a ajouté des attributs dans la fermeture, vérifier si des DF qui ne s'appliquaient pas jusqu'à présent ne s'appliquent pas dorénavant.

Algorithme de calcul de la fermeture d'un ensemble d'attributs X par rapport à un ensemble de DF F

Entrée : X et F Sortie : $[X]_F^+$

1. $[X]_F^0 = X$ $i = 0$
2. $[X]_F^{i+1} := [X]_F^i$
3. Pour tout $f : Y \rightarrow Z \in F$ tel que $Y \subset [X]_F^i$ faire :
 - (a) $[X]_F^{i+1} := [X]_F^i \cup Z$
 - (b) $F := F \setminus \{f\}$
4. Si $[X]_F^{i+1} \neq [X]_F^i$ alors $i := i + 1$ et aller en 2.
5. Sinon, $[X]_F^+ := [X]_F^{i+1}$, retourner $[X]_F^+$

Si on applique notre algorithme à $U = \{\text{NomFournisseur, Adresse, Produit, Prix, FraisTransport}\}$ et $F = \{\text{NomFournisseur} \rightarrow \text{Adresse}, \{\text{NomFournisseur, Produit}\} \rightarrow \text{Prix}, \{\text{Adresse, Produit}\} \rightarrow \text{FraisTransport}\}$ on obtient pour le calcul de $[\text{NomFournisseur, Produit}]_F^+$:

1. $[\text{NomFournisseur, Produit}]_F^0 = \{\text{NomFournisseur, Produit}\}$
2. $[\text{NomFournisseur, Produit}]_F^1 = \{\text{NomFournisseur, Produit, Adresse, Prix}\}$ car on a pu appliquer les DF $\text{NomFournisseur} \rightarrow \text{Adresse}$ et $\{\text{NomFournisseur, Produit}\} \rightarrow \text{Prix}$
3. $[\text{NomFournisseur, Produit}]_F^2 = \{\text{NomFournisseur, Produit, Adresse, Prix, FraisTransport}\}$ car on a pu appliquer la DF $\{\text{Adresse, Produit}\} \rightarrow \text{FraisTransport}$.
4. $[\text{NomFournisseur, Produit}]_F^3 = [\text{NomFournisseur, Produit}]_F^2$ car il n'y a plus de DF à appliquer.
Donc $[\text{NomFournisseur, Produit}]_F^+ = \{\text{NomFournisseur, Produit, Adresse, Prix, FraisTransport}\}$

Utilisation de la fermeture d'un ensemble d'attributs

Le calcul de la fermeture d'un ensemble d'attributs par rapport à un ensemble de DF est intéressant car il permet de déterminer si $F \models Y \rightarrow Z$. En effet, il est évident que :

Lemme 4.2.1

$$(F \models Y \rightarrow Z) \Leftrightarrow (Z \in [Y]_F^+)$$

Notons aussi que la fermeture transitive d'un ensemble d'attributs est un moyen, entre autres, de déterminer la ou les clés d'un schéma de relation R . En effet, une clé est un ensemble minimal d'attributs du schéma de relation qui permet de déterminer tous les autres. Or, l'ensemble de tous les attributs du schéma de relation est une surclé (il permet de se déterminer lui-même, donc tous les attributs du schéma de relation). Pour trouver une clé, il suffit donc de supprimer des attributs de cet ensemble jusqu'à ce qu'on arrive à un ensemble K tel que :

- $K \subseteq R$
- $[K]_F^+ = R$
- $\forall A \in K, [K \setminus \{A\}]_F^+ \neq R$

4.2.3.3 Axiomes d'Armstrong [1974]

Les axiomes d'Armstrong permettent d'automatiser la déduction de DF à partir de DF connues. Il existe trois axiomes principaux, et d'autres règles qui peuvent s'en déduire.

Soient X, Y et Z trois sous-ensembles d'attributs de U . Les trois axiomes principaux sont les suivants :

- (réflexivité ou DF triviales) : $XY \rightarrow Y$
- (augmentation) : $\{X \rightarrow Y\} \Rightarrow XZ \rightarrow YZ$

- (transitivité) : $\{X \rightarrow Y, Y \rightarrow Z\} \Rightarrow X \rightarrow Z$

Théorème 4.2.1

Les axiomes d'Armstrong sont :

- *Sains, c'est-à-dire que si $F \Rightarrow X \rightarrow Y$ alors $F \models X \rightarrow Y$*
- *Complets, c'est-à-dire que si $F \models X \rightarrow Y$ alors $F \Rightarrow X \rightarrow Y$*

En d'autres termes, ce théorème montre non seulement que si on utilise ces axiomes pour inférer une nouvelle DF, alors le résultat est correct, mais aussi que, pour toute DF qui se déduit d'un ensemble de DF existantes, il existe une preuve de cette déduction grâce aux axiomes. La preuve de la première partie du théorème est très facile (on l'a fait déjà pour la transitivité), la seconde partie est plus compliquée (voir le chapitre 7.3 dans [Ullman88]).

A partir des trois axiomes de base, on peut trouver d'autres règles d'inférence. Notons que d'après le théorème précédent, on peut maintenant utiliser \models au lieu de \Rightarrow

- (union) : $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$
- (pseudo-transitivité) : $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow YZ$
- (décomposition) : $\{X \rightarrow YZ\} \models X \rightarrow Y$

Comme exemple d'utilisation des axiomes d'Armstrong, montrons que si $F = \{A \rightarrow C, B \rightarrow D\}$ alors AB est une surclé de $R = ABCD$.

Par augmentation de $A \rightarrow C$ on obtient $AB \rightarrow ABC$. De même, par augmentation de $B \rightarrow D$ on obtient $B \rightarrow BD$, puis $ABC \rightarrow ABCD$. Par transitivité entre $AB \rightarrow ABC$ et $ABC \rightarrow ABCD$ on obtient $AB \rightarrow ABCD$. AB est donc bien une surclé de R .

4.2.4 Equivalence et couverture minimale d'ensembles de DF

On a vu qu'à partir de dépendances fonctionnelles on peut en déduire d'autres. Il est naturel alors de se poser plusieurs questions :

- deux ensembles de DF peuvent-ils être équivalents, c'est-à-dire exprimer les mêmes contraintes sur les données de la base ?
- un ensemble de DF peut-il être redondant, c'est-à-dire contient-il des DF qui peuvent se déduire les unes des autres ?
- si deux ensembles de DF sont équivalents, existe-t-il l'un d'entre eux qui soit "meilleur" que l'autre ?
- peut-on trouver une forme la plus compacte possible pour un ensemble de dépendances fonctionnelles ?

Dans cette section, nous allons voir comment répondre de manière affirmative à toutes ces questions.

Tout d'abord, on va définir l'équivalence entre deux ensembles de DF. Deux ensembles de DF sont équivalents si ils expriment les mêmes contraintes sur les données de la base. Or, pour connaître l'ensemble des contraintes qu'impose un ensemble de DF, il est naturel de calculer sa fermeture transitive. Deux ensembles de DF sont donc équivalents si ils ont la même fermeture transitive.

Définition 4.2.5 *Soient F et F' deux ensembles de dépendances fonctionnelles sur U , on dit que F et F' sont équivalents, qu'on note $F \equiv F'$ si ils ont la même fermeture transitive :*

$$F \equiv F' \quad \text{si} \quad F^+ = F'^+$$

□

Il s'agit maintenant, à partir d'un ensemble de DF donné, de trouver un autre ensemble de DF qui lui soit équivalent et qui soit meilleur, c'est-à-dire plus compact et mieux manipulable par la suite. Pour cela, il faut vérifier plusieurs choses, qui définissent un *ensemble minimal de DF*.

Définition 4.2.6 *Soit F un ensemble de DF sur U . On dit que F est un ensemble minimal de DF si les trois conditions suivantes sont respectées :*

1. F est sous forme canonique, c'est-à-dire que toute dépendance de F est sous la forme $X \rightarrow A$ (un seul attribut à droite)
2. F ne contient pas de DF redondante, c'est-à-dire qu'aucune DF de F ne peut être déduite à partir d'autres DF de F

$$\forall f \in F, F \setminus \{f\} \not\equiv F$$

3. F ne contient aucune DF redondante à gauche, c'est-à-dire qu'on ne peut pas supprimer des attributs de la partie gauche d'une DF de F tout en obtenant un ensemble équivalent à F .

$$\forall f : XY \rightarrow A \in F, ((F \setminus \{XY \rightarrow A\}) \cup \{X \rightarrow A\}) \not\equiv F$$

□

La première condition est purement technique, elle permet d'avoir un ensemble plus facilement manipulable en théorie. Les deux suivantes expriment bien la compacité recherchée. On peut maintenant définir la notion de *couverture minimale* d'un ensemble de DF.

Définition 4.2.7 Soit F un ensemble de DF sur U . F' est une **couverture minimale** de F si :

1. $F \equiv F'$ et
2. F' est un ensemble minimal de DF.

□

L'algorithme qui permet de calculer la couverture minimale d'un ensemble de DF suit la définition. Après avoir mis F sous forme canonique grâce à la règle de décomposition d'Armstrong, on essaie de supprimer toutes les DF possibles de F tout en gardant un ensemble équivalent. Enfin, on essaie de supprimer, dans les DF restantes, tout attribut dans la partie gauche en gardant un ensemble de DF équivalent.

Algorithme de calcul de la couverture minimale d'un ensemble de DF F

Entrée : F Sortie : une couverture minimale de F

1. Mettre F sous forme canonique, c'est-à-dire remplacer toute DF $X \rightarrow ABC \dots$ par l'ensemble de DF $\{X \rightarrow A, X \rightarrow B, X \rightarrow C, \dots\}$. On obtient $F1$
2. $F2 := F1$. Pour toute DF f de $F1$, si $(F2 \setminus f) \equiv F2$, alors remplacer $F2$ par $(F2 \setminus f)$
3. $F3 := F2$. Pour toute DF $f : X \rightarrow A$ de $F2$, et soit $f' : Y \rightarrow A$ tel que :
 - $Y \subset X$ (inclusion stricte)
 - $((F2 \setminus \{f\}) \cup \{f'\}) \equiv F2$
 - $\forall Z \subset Y, f'' : Z \rightarrow A$, et $((F2 \setminus \{f\}) \cup \{f''\}) \not\equiv F2$
 faire $F3 := (F2 \setminus \{f\}) \cup \{f'\}$
4. retourner $F3$, une couverture minimale de F

Pour calculer, à l'étape 2. et 3., si un ensemble $F \setminus f$ est équivalent à F , il suffit d'utiliser le lemme suivant.

Lemme 4.2.2

$$(F \setminus f) \equiv F \text{ si et seulement si } (F \setminus f) \models f$$

D'après le lemme 4.2.1, on voit que pour vérifier que $F \models \{X \rightarrow A\}$, il suffit de calculer $[X]_{F \setminus A}^+$ et de vérifier si A y figure ou non.

Notons que les trois premières étapes correspondent bien aux trois conditions d'un ensemble minimal. Notons aussi que l'algorithme n'est pas déterministe. L'ordre dans lequel on étudie les DF à l'étape 2., ainsi que la façon de choisir Z parmi X à l'étape 3. ne sont pas fixés. Ceci implique qu'il est possible de trouver plusieurs couvertures minimales d'un ensemble de DF, selon les choix faits aux étapes 2. et 3.

Notons enfin que lorsqu'on supprime une DF de $F2$ à l'étape 2., elle ne pourra plus être considérée pour tester si une autre DF doit être supprimée.

Par exemple, cherchons une couverture minimale de $F = \{AB \rightarrow CD, ACE \rightarrow B, D \rightarrow C, C \rightarrow D, CD \rightarrow BE\}$:

1. Mise sous forme canonique : $F1 = \{AB \rightarrow C, AB \rightarrow D, ACE \rightarrow B, D \rightarrow C, C \rightarrow D, CD \rightarrow B, CD \rightarrow E\}$
2. Test de redondance des DF :
 - Testons $AB \rightarrow C$. Elle est redondante car $AB \rightarrow D$ et $D \rightarrow C$. (on aurait pu aussi calculer la fermeture de AB par rapport à $F1 \setminus AB \rightarrow C$).
Donc maintenant $F1 = \{AB \rightarrow D, ACE \rightarrow B, D \rightarrow C, C \rightarrow D, CD \rightarrow B, CD \rightarrow E\}$
 - Testons $AB \rightarrow D$. Calculons $[AB]_{F1}^+ = \{A, B\}$, qui ne contient pas D , donc $AB \rightarrow D$ n'est pas redondante, on la garde.
 - Testons $ACE \rightarrow B$. $[ACE]_{F1}^+ = \{A, C, E, D, B\}$, qui contient B . On supprime la DF $ACE \rightarrow B$.
Donc maintenant $F1 = \{AB \rightarrow D, D \rightarrow C, C \rightarrow D, CD \rightarrow B, CD \rightarrow E\}$
 - de la même manière, on montre que les autres DF ne sont pas redondantes.
Donc $F2 = \{AB \rightarrow D, D \rightarrow C, C \rightarrow D, CD \rightarrow B, CD \rightarrow E\}$
3. Test de redondance des attributs à gauche :
 - Testons si on peut supprimer A dans $AB \rightarrow D$. Pour cela, on calcule $[B]_{F2}^+ = \{B\}$ qui ne contient pas D , on ne peut pas supprimer A à gauche. Testons si on peut supprimer B : $[A]_{F2}^+ = \{A\}$ qui ne contient pas D , on ne peut pas supprimer B à gauche. Conclusion, la DF $AB \rightarrow D$ ne contient aucun attribut redondant à gauche.
 - $D \rightarrow C$ et $C \rightarrow D$ n'ont qu'un seul attribut à gauche donc forcément ne contiennent pas d'attribut redondant à gauche.
 - Comme elles ont les mêmes parties gauche, on peut tester $CD \rightarrow B$ et $CD \rightarrow E$ simultanément :
 $[C]_{F2}^+ = \{C, D, B, E\}$ qui contient B et E donc D est redondant à gauche dans les deux DF.
 $[D]_{F2}^+ = \{C, D, B, E\}$ qui contient B et E donc D est redondant à gauche dans les deux DF.
Donc on peut remplacer $CD \rightarrow B$ soit par $C \rightarrow B$ soit par $D \rightarrow B$, et on peut remplacer $CD \rightarrow E$ soit par $C \rightarrow E$ soit par $D \rightarrow E$

On obtient donc au moins 4 couvertures minimales différentes pour F :

$$G1 = \{AB \rightarrow D, D \rightarrow C, C \rightarrow D, C \rightarrow B, C \rightarrow E\}, G2 = \{AB \rightarrow D, D \rightarrow C, C \rightarrow D, D \rightarrow B, D \rightarrow E\},$$

$$G3 = \{AB \rightarrow D, D \rightarrow C, C \rightarrow D, C \rightarrow B, D \rightarrow E\}, G4 = \{AB \rightarrow D, D \rightarrow C, C \rightarrow D, D \rightarrow B, C \rightarrow E\}$$

4.3 Décomposition de schéma et formes normales

Dans tout ce chapitre, on va supposer que les ensembles de DF considérés sont tout minimaux (cf. 4.2.4)

4.3.1 Décomposition de schéma relationnel

On a vu au chapitre 4.2 que, du fait des DF, un schéma relationnel fait “au hasard” d’après un schéma E/A pouvait contenir des redondances, et qu’un schéma avec les mêmes attributs mais décomposé en plusieurs relations pouvait éliminer ces problèmes. Dans cette section on va voir de manière plus détaillée ce qu’est une décomposition de schéma et quelles bonnes propriétés on va en exiger.

4.3.1.1 Définition

Définition 4.3.1 Soit $R(A_1, \dots, A_n)$ un schéma de relation. Une **décomposition de R** est un ensemble de schémas de relation R_1, \dots, R_k tel que :

- $\forall i, R_i \subset R$
- $R = \bigcup_i R_i$
- $\forall i, j \exists l_1, l_2, \dots, l_m, R_i \cap R_{l_1} \neq \emptyset \wedge R_{l_1} \cap R_{l_2} \neq \emptyset \wedge R_{l_2} \cap R_{l_3} \neq \emptyset \wedge \dots \wedge R_{l_m} \cap R_j \neq \emptyset$

□

Ces trois conditions expriment que la décomposition possède exactement les mêmes attributs que R et qu’il est possible de recomposer R par une jointure sur les R_i .

Un exemple de décomposition est celui vu au chapitre 4.2. On passe de S1 à S2 par décomposition du schéma de relation FournisseursProduits en Fournisseurs, Produit. On avait vu que cela permet entre autres de pouvoir stocker un fournisseur même si il ne fournit aucun produit.

Les questions qui se posent à présent sont : est-ce qu’en général c’est toujours bien de décomposer ? Notamment, est-ce que le schéma décomposé contient les mêmes informations que le schéma de départ, qui lui reflète la réalité observée ? Comme c’étaient les DF qui faisaient le “mauvais schéma”, ce sont elles aussi qui vont nous dire si la décomposition est bonne ou non.

4.3.1.2 Décomposition sans perte d’information (SPI)

Pour répondre à la question précédente, prenons r , instance de R . Si on décompose R en R_1, \dots, R_k alors r va être décomposé naturellement en r_1, \dots, r_k par projection : $r_1 = \prod_{R_1}(r), \dots, r_k = \prod_{R_k}(r)$. Ensuite, pour retrouver $r(R)$, on va faire une jointure naturelle des r_i , $s = r_1 \bowtie r_2 \bowtie \dots \bowtie r_k$.

Si $s = r$, c’est OK, la décomposition contient bien la même information que le schéma initial. Sinon notre décomposition “perd” de l’information. En fait, si on ne tient pas compte des DF, on risque d’avoir n’importe quoi. Par exemple :

$R(A, B, C)$	$R_1(A, B)$	$R_2(B, C)$	$(R_1 \bowtie R_2)(A, B, C)$
(a_1, b_1, c_1)	(a_1, b_1)	(b_1, c_1)	(a_1, b_1, c_1)
(a_2, b_1, c_2)	(a_2, b_1)	(b_1, c_2)	(a_1, b_1, c_2)
			(a_2, b_1, c_1)
			(a_2, b_1, c_2)

On voit sur cet exemple que dans la jointure naturelle finale, on a “inventé” deux n-uplets qui n’étaient pas dans la relation de départ. Cette décomposition n’est pas sans perte d’information.

Définition 4.3.2 Une décomposition de R en $\{R_1, \dots, R_k\}$ est **sans perte d’information par rapport à un ensemble de DF F** si :

$$\forall r(R), r = \prod_{R_1}(r) \bowtie \dots \bowtie \prod_{R_k}(r)$$

□

NB : en fait, le terme exact est sans perte d'information aux jointures, mais comme on ne voit que cette forme de perte d'information, on confondra les deux termes.

Le problème est de tester si une décomposition est SPI ou non. Pour cela, on peut établir le lemme suivant qui permet de n'avoir à tester qu'une inclusion au lieu d'une égalité.

Lemme 4.3.1

$$\forall R, \forall \{R_1, \dots, R_k\} \text{ décomposition de } R, \forall r(R), \quad r \subset \prod_{R_1}(r) \bowtie \dots \bowtie \prod_{R_k}(r)$$

Preuve : La preuve de ce lemme est évidente. Il suffit de prendre un n-uplet quelconque dans r , de faire les projections puis la jointure et on voit qu'on retrouve le n-uplet.

D'après ce lemme, on voit donc que, pour tester si une décomposition est sans perte d'information, il suffit de vérifier qu'on n'invente pas de n-uplet (ce qui est bien une perte d'information, dans l'hypothèse de monde fermé) car on ne risque pas d'en perdre.

Donc, pour tout $r(R)$, on veut savoir si, étant donné un n-uplet dans la jointure, est que ce n-uplet est dans r . Pour cela, on utilise l'algorithme de poursuite ("chase").

Algorithme de poursuite pour déterminer si une décomposition est SPI par rapport à un ensemble de DF F

Entrée : $R(A_1, \dots, A_n)$, $\{R_1, \dots, R_k\}$ décomposition de R et F

Sortie : Vrai ou Faux, selon que la décomposition est SPI ou non.

1. Construire une table à n colonnes (une par attribut de R) et k lignes (une par relation dans la décomposition)

On suppose qu'on a un n-uplet (a_1, \dots, a_n) dans la jointure et on veut savoir si il provient de la relation de départ. Dans ce cas, il va être formé par jointure des différentes projections de ce n-uplet. Le tableau représente les n-uplets de la relation de départ qui ont donné ces différentes projections. On a au maximum k n-uplets possibles, d'où les k lignes. Chaque ligne j correspond donc au n-uplet contenant la projection sur R_j du n-uplet trouvé dans la jointure. Pour tous les attributs A_i de R_j , on va donc renseigner la ligne j à la colonne i par la valeur a_i . Pour les autres colonnes on ne sait pas, on va donc mettre b_{ij} (b = bidon, inconnu), qu'on suppose tous différents deux à deux.

Ensuite, il faut voir si les DF nous permettent de conclure.
2. Pour chaque DF $X \rightarrow Y$ de F , on "applique" la DF : si on a deux lignes qui sont égales sur X , alors elle sont égales sur Y . Si un des deux n-uplet a une valeur non-bidon sur une ligne, alors l'autre aussi. On remplace donc la valeur bidon par la valeur connue.
3. Une fois qu'on a fini toutes les DF, on recommence en 2. car de nouvelles valeurs connues ont pu être ajoutées et donc d'autres DF peuvent éventuellement s'appliquer
4. On s'arrête lorsqu'une des deux conditions suivantes est satisfaite :
 - Si on a un n-uplet entièrement renseigné, on s'arrête, et on retourne Vrai
 - Si un passage de toutes les DF ne permet aucun ajout de valeur non-bidon, on s'arrête et on retourne Faux.

Appliquons cet algorithme sur la décomposition de FournisseursProduits en Fournisseurs et Produit. F vaut ici $\{NomFournisseur \rightarrow Adresse, \{NomFournisseur, Produit\} \rightarrow Prix\}$. Soit $t = (nom, adr, prod, px)$ un n-uplet dans la jointure Produits \bowtie Fournisseurs, le tableau de poursuite est :

NomFournisseur	Adresse	Produit	Prix
<i>nom</i>	<i>adr</i>	<i>b</i> ₃₁	<i>b</i> ₄₁
<i>nom</i>	<i>b</i> ₂₂	<i>prod</i>	<i>px</i>

La première DF nous dit que les deux n-uplets doivent avoir même adresse. Donc l'attribut Adresse du deuxième n-uplet vaut *adr* et c'est gagné puisque t est dans la relation de départ. La décomposition est SPI

Appliquons l'algorithme de poursuite à un autre exemple, où on décompose FournisseursProduits en $R_1(\text{NomFournisseur}, \text{Adresse}, \text{Produit})$ et $R_2(\text{NomFournisseur}, \text{Prix})$. Soit $t(\text{nom}, \text{adr}, \text{prod}, \text{px})$ notre n-uplet dans la jointure $R_1 \bowtie R_2$, le tableau de poursuite est :

NomFournisseur	Adresse	Produit	Prix
<i>nom</i>	<i>adr</i>	<i>prod</i>	<i>b₄₁</i>
<i>nom</i>	<i>b₂₂</i>	<i>b₃₂</i>	<i>px</i>

La première DF nous dit que les deux n-uplets doivent avoir même adresse. Donc l'attribut Adresse du deuxième n-uplet vaut *adr*. La seconde DF ne peut pas être appliquée. Mais ensuite on ne peut plus rien appliquer, donc ÉCHEC. La décomposition n'est pas SPI.

Dans le cas particulier où la décomposition ne comporte que deux relations, on peut éviter de construire le tableau de poursuite en utilisant le théorème suivant :

Théorème 4.3.1

Soit $\{R_1, R_2\}$ une décomposition de R et F l'ensemble des DF qui s'appliquent à R , alors :
 La décomposition de R en $\{R_1, R_2\}$ est SPI ssi $((R_1 \cap R_2) \rightarrow R_1) \vee ((R_1 \cap R_2) \rightarrow R_2)$

En d'autres termes, la décomposition est SPI ssi l'intersection de R_1 et R_2 est une surclé d'au moins une des deux relations. Par exemple $R(A, B, C)$ se décompose SPI en $R_1(AB)$ et $R_2(AC)$ mais pas $R_2(BC)$ si $F = \{A \rightarrow C\}$

4.3.1.3 Décomposition sans perte de dépendance (SPD)

La propriété SPI est nécessaire pour une (bonne) décomposition. Malheureusement, elle n'est pas suffisante comme le montre l'exemple suivant.

Soit $R(\text{CodePostal}, \text{Ville}, \text{Rue})$ et $F = \{\{ \text{Ville}, \text{Rue} \} \rightarrow \text{CodePostal}, \text{CodePostal} \rightarrow \text{Ville}\}$. On décompose R en $R_1(\text{CodePostal}, \text{Rue})$ et $R_2(\text{CodePostal}, \text{Ville})$. Cette décomposition est SPI d'après le théorème 4.3.1 car CodePostal est une clé de R_2 .

Le problème est sur la DF $\{ \text{Ville}, \text{Rue} \} \rightarrow \text{CodePostal}$. On a vu au chapitre 4.2 qu'on ne peut vérifier efficacement que des DF dont tous les attributs sont inclus dans un schéma de relation. Ici, les attributs sont partagés entre R_1 et R_2 . Il faut donc faire attention lorsqu'on décompose, car on ne peut vérifier que des dépendances projetées.

Définition 4.3.3 Une DF $X \rightarrow Y$ est **projetée** si il existe une relation de la décomposition qui contient XY □

Il faut donc que la décomposition permette de vérifier toutes les DF. **Attention**, on n'est pas obligé de retrouver une DF directement, le tout est qu'elle soit maintenue grâce aux DF projetées, i.e. vérifiables sur une seule table. Par exemple, si on a : $R(A, B, C, D)$ et $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A\}$ et qu'on décompose R en $\{R_1(AB), R_2(BC), R_3(CD)\}$ on peut projeter les trois premières DF, mais pas la 4e. L'a-t-on perdue? Non car on peut déduire de F que $B \rightarrow A$ qu'on va projeter sur R_1 . De même $D \rightarrow C$ sur R_3 et $C \rightarrow B$ sur R_2 . Or, $\{B \rightarrow A, D \rightarrow C, C \rightarrow B\} \models D \rightarrow A$. On peut donc, en vérifiant des DF projetées, garantir que $D \rightarrow A$. En fait, il ne faut pas considérer F mais F^+ .

Définition 4.3.4 On note F_R les DF de F qui se projettent sur le schéma de relation R .

Une décomposition de R en $\{R_1, \dots, R_k\}$ est **sans perte de dépendances par rapport à F** si :

$$F^+ = [F_{R_1}^+ \cup F_{R_2}^+ \dots F_{R_k}^+]^+$$

□

D'après la définition précédente, on se dit alors qu'il va falloir calculer deux fermetures transitives d'ensembles de DF, alors qu'on sait que c'est très coûteux. Heureusement, ce n'est pas le cas grâce aux deux lemmes suivants.

Lemme 4.3.2 *On ne peut pas créer de nouvelles DF lorsqu'on fait une décomposition, en d'autres termes :*

$$F^+ \supseteq [F_{R_1}^+ \cup F_{R_2}^+ \dots F_{R_k}^+]^+$$

Ce résultat est évident car on fait la fermeture d'un ensemble de DF inclus dans F^+ .

Lemme 4.3.3 *Pour que la décomposition soit SPD, il faut et il suffit que toute DF de F puisse se déduire des DF de F^+ projetées sur la décomposition, en d'autres termes :*

$$(\forall f \in F^+, f \in [F_{R_1}^+ \cup F_{R_2}^+ \dots F_{R_k}^+]^+) \Leftrightarrow (\forall f \in F, f \in [F_{R_1}^+ \cup F_{R_2}^+ \dots F_{R_k}^+]^+)$$

Ce résultat est tout aussi évident.

Donc le but est, pour tout f dans F , d'essayer de la déduire à partir des DF de $[F_{R_1}^+ \cup F_{R_2}^+ \dots F_{R_k}^+]^+$. On peut bien entendu essayer de le montrer directement, comme dans l'exemple avec $R(A, B, C, D)$, et tel que $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A\}$ et $\{R_1(AB), R_2(BC), R_3(CD)\}$ mais dans le cas général, on utilise l'algorithme suivant.

Algorithme de test si une DF est perdue par décomposition

Entrée : $R(A_1, \dots, A_n)$, $\{R_1, \dots, R_k\}$ décomposition de R , F et $f : X \rightarrow Y$ dans F
Sortie : Vrai ou Faux, selon que la DF f est perdue ou non.

1. $Z := X$
2. pour $i = 1$ à k faire $Z := Z \cup ([Z \cap R_i]_F^+ \cap R_i)$
3. Si Z contient Y , alors stop et retourner Faux.
Sinon, si Z n'a pas augmenté à l'étape 2. alors stop et retourner Vrai.
4. continuer en 2.

Pourquoi cet algorithme marche-t-il? Parce qu'on fait systématiquement la fermeture des attributs par rapport à F , ce qui est la même chose que de faire la fermeture par rapport à F^+ .

Si on applique notre algorithme à l'exemple précédent (on cherche à savoir si $D \rightarrow A$ est perdue).

1. $Z = \{D\}$
 - (a) $i = 1$. on ne trouve rien (forcément)
 - (b) $i = 2$ on ne trouve rien non plus
 - (c) $i = 3$. $Z = \{D\} \cup ((\{D\} \cap \{C, D\})_F^+ \cap \{C, D\}) = \{D\} \cup (\{D\}_F^+ \cap \{C, D\}) = \{D\} \cup (\{A, B, C, D\} \cap \{C, D\}) = \{C, D\}$
2. Z a augmenté, on recommence donc à faire une passe sur les DF. Après la troisième passe, on obtiendra $Z = \{A, B, C, D\}$ qui contient A , on retourne Faux (la DF n'est pas perdue)

4.3.2 Formes normales

On a vu qu'il y a des schémas redondants et qu'on peut décomposer en espérant trouver un meilleur schéma sans perdre la sémantique trouvée dans la modélisation Entité-Association enrichie des contraintes d'intégrité (y compris les DF). Les questions qui se posent maintenant sont :

- est-ce qu'on peut caractériser la qualité d'un schéma par rapport à la redondance due aux DF?
- peut-on trouver des décompositions SPI, SPD qui permettent d'arriver à une "meilleure" qualité?

La réponse à la première question est dans la notion de forme normale. La réponse à la deuxième est dans les algorithmes de décomposition associés. Il existe plusieurs formes normales pour le modèle relationnel. Ici nous ne verrons que celle relative aux DF. La première forme normale exprime le respect du modèle relationnel. La forme normale de Boyce-Codd (FNBC) est la plus restrictive, elle caractérise les schémas de relation qui ne permettent aucune redondance dues aux DF, malheureusement il n'est pas toujours possible de décomposer SPI et SPD un schéma de relation en un ensemble de schémas de relation qui soient tous

en FNBC. La troisième forme normale (3FN) est moins restrictive que la forme de Boyce-Codd, elle permet certaines redondances mais il existe un algorithme de décomposition SPI et SPD qui permet de passer d'un schéma de relation quelconque à un ensemble de schémas de relation tous en 3FN. Il existe aussi une deuxième forme normale (2FN), encore moins restrictive que la 3FN, mais dont l'intérêt est limité. Un schéma de relation en FNBC est forcément en 3FN, un schéma de relation en 3FN est forcément en 2FN, un schéma de relation en 2FN est forcément en 1FN¹. En résumé :

$$FNBC \Rightarrow 3FN \Rightarrow 2FN \Rightarrow 1FN$$

4.3.2.1 Forme normale de Boyce-Codd (FNBC)

Comme dit précédemment, la forme normale de Boyce-Codd est la plus restrictive lorsqu'on ne considère que les DF : un schéma en FNBC ne permet aucune redondance due aux DF.

Définition 4.3.5 Soient R un schéma de relation et F un ensemble de DF. On dit que R est en **forme normale de Boyce-Codd par rapport à F** (FNBC ou BCNF en anglais) si pour toute DF non-triviale $X \rightarrow A$ de F^+ applicable à R , alors X est une surclé de R . \square

Cette définition dit que, dans R , tout attribut "dépend" uniquement de la ou des clés. Considérons à nouveau l'exemple $R(\text{CodePostal}, \text{Ville}, \text{Rue})$ et $F = \{\{\text{Ville}, \text{Rue}\} \rightarrow \text{CodePostal}, \text{CodePostal} \rightarrow \text{Ville}\}$. On peut montrer facilement que les clés possibles sont $\{\text{CodePostal}, \text{Rue}\}$ et $\{\text{Ville}, \text{Rue}\}$ Donc la DF $\text{CodePostal} \rightarrow \text{Ville}$ viole la forme FNBC (CodePostal n'est pas une surclé de R). On a donc des redondances par rapport aux DF.

Notons le cas particulier intéressant suivant.

Lemme 4.3.4 Toute relation à deux attributs est FNBC

4.3.2.2 Troisième forme normale (3FN)

La troisième forme normale est un peu plus faible que FNBC mais elle est intéressante car on peut toujours l'"obtenir" (cf 4.3.2.4).

Définition 4.3.6 Soient R un schéma de relation et F un ensemble de DF. On dit que R est en **troisième forme normale par rapport à F** (3FN ou 3NF en anglais) si pour toute DF non-triviale $X \rightarrow A$ applicable à R , alors X est une surclé de R ou bien A est premier (A appartient à une clé minimale de R). \square

Dans l'exemple $R(\text{CodePostal}, \text{Ville}, \text{Rue})$ et $F = \{\{\text{Ville}, \text{Rue}\} \rightarrow \text{CodePostal}, \text{CodePostal} \rightarrow \text{Ville}\}$, on peut montrer facilement que R est en 3FN.

L'exemple FournisseursProduits(NomFournisseur, Adresse, Produit, Prix) avec $F = \{\text{NomFournisseur} \rightarrow \text{Adresse}, \{\text{NomFournisseur}, \text{Produit}\} \rightarrow \text{Prix}\}$, par contre, n'est pas en troisième forme normale. En effet, la seule clé de FournisseursProduits est $\{\text{NomFournisseur}, \text{Produit}\}$, donc la DF $\text{NomFournisseur} \rightarrow \text{Adresse}$ viole la 3FN.

4.3.2.3 2e forme normale (2FN)

Si un schéma de relation R n'est pas en 3FN, alors il existe une DF $X \rightarrow A$ applicable à R telle que :

- Soit A n'est pas premier.
- Soit X n'est pas surclé.

Deux cas sont alors possibles :

- X est inclus strictement dans une clé de R : on dit alors que $X \rightarrow A$ est une DF *partielle*.
- X n'est inclus dans aucune clé de R : on dit alors que $X \rightarrow A$ est une DF *transitive*.

Définition 4.3.7 Soient R un schéma de relation et F un ensemble de DF. On dit que R est en **deuxième forme normale (2FN)** si il n'y a pas de DF partielle applicable à R . \square

¹la première forme normale (1FN) exprime le respect du modèle relationnel

L'exemple FournisseursProduits(NomFournisseur, Adresse, Produit, Prix) avec $F = \{NomFournisseur \rightarrow Adresse, \{NomFournisseur, Produit\} \rightarrow Prix\}$, n'est pas en deuxième forme normale. En effet, la DF $NomFournisseur \rightarrow Adresse$ est une dépendance partielle.

Cas particulier :

Toute relation dont la seule clé a un seul attribut est forcément 2FN.

Récapitulatif :

- 2FN = 1FN et pas de DF partielle
- 3FN = 2FN et pas de DF transitive
- FNBC = 3FN et pas de DF $X \rightarrow A$ applicable à R où X n'est pas surclé de R .

4.3.2.4 Algorithmes de passage aux formes normales FNBC et 3FN

• Algorithme de passage en 3FN, SPI et SPD

Entrée : $R(A_1, \dots, A_n)$ et F un ensemble minimal de DF

Sortie : une décomposition SPI et SPD $\{R_1, \dots, R_n\}$ de R telle que $\forall i, R_i$ est en 3FN.

1. Regrouper les DF qui ont même partie gauche.
2. Créer un schéma de relation R_i avec les attributs de chaque groupe de DF
3. Si aucune clé minimale de R n'apparaît dans un R_i existant, rajouter un schéma de relation formé par les attributs d'une clé minimale de R .
4. Eliminer les schémas de relation inclus dans d'autres.
5. Retourner les schémas de relations produits restants.

La preuve que la décomposition de R fournie par cette algorithme est SPD est évidente. La preuve qu'elle est SPI est moins aisée, et peut être faite à titre d'exercice. De même pour la preuve que chaque schéma de relation produit est en 3FN.

• Algorithme de passage en FNBC, SPI

Cet algorithme se base sur le lemme 4.3.4, le théorème 4.3.1 et sur le lemme suivant.

Lemme 4.3.5 Soit R un schéma de relation et F un ensemble de DF. Si $\{R_1, R_2, \dots, R_n\}$ est une décomposition SPI de R et $\{S_1, \dots, S_m\}$ est une décomposition SPI de R_1 , alors $\{S_1, \dots, S_m, R_2, \dots, R_n\}$ est une décomposition SPI de R

L'algorithme est le suivant : Entrée : $R(A_1, \dots, A_n)$ et F un ensemble minimal de DF

Sortie : une décomposition SPI $\{R_1, \dots, R_n\}$ de R telle que $\forall i, R_i$ est en FNBC.

1. $S := R$
2. Si S n'est pas FNBC, alors il existe $X \rightarrow A$ tel que A n'est pas surclé de S .
Sinon : terminé
3. Recommencer l'étape 2. avec $S := R_1 = \{XA\}$ puis avec $S := R_2 = S \setminus \{A\}$

A chaque passage à l'étape 3., la décomposition est clairement SPI, puisque l'intersection entre R_1 et R_2 est $\{X\}$ qui est clairement clé de R_1 d'après le théorème 4.3.1. D'après le lemme 4.3.5, on sait que la décomposition de R reste SPI à chaque application des étapes 2. et 3. Enfin, à chaque étape, le nombre d'attribut diminue : R_1 et R_2 ont strictement moins d'attributs que S . Donc au pire des cas, on doit décomposer jusqu'à ce que R_1 (resp. R_2) n'ait plus que deux attributs, ce qui est une forme FNBC d'après le lemme 4.3.4.

Appliquons l'algorithme sur l'exemple $R(\text{CodePostal}, \text{Ville}, \text{Rue})$. On a vu que R n'est pas en FNBC par rapport à $F = \{\{\text{Ville}, \text{Rue}\} \rightarrow \text{CodePostal}, \text{CodePostal} \rightarrow \text{Ville}\}$.

Avec l'algorithme, on calcule les clés de R qui sont $\{\text{CodePostal}, \text{Ville}\}$ et $\{\text{Ville}, \text{Rue}\}$. On voit que la dépendance fonctionnelle $\text{CodePostal} \rightarrow \text{Ville}$ viole la forme FNBC et on a $R_1 = \{\text{CodePostal}, \text{Ville}\}$ qui est FNBC et $R_2 = R \setminus \{\text{Ville}\} = \{\text{CodePostal}, \text{Rue}\}$ lui-même FNBC.

Le résultat est $\{R_1(\text{CodePostal}, \text{Ville}), R_2(\text{CodePostal}, \text{Rue})\}$ une décomposition SPI de R telle que R_1 et R_2 sont BCNF.

Il n'est pas évident que cet algorithme soit très efficace, car il nécessite de calculer systématiquement toutes les clés possibles (à partir de F , et non pas uniquement des DF qui se projettent) pour faire le test. On trouve dans le livre de Ullman [Ullman88] un algorithme qui évite ce calcul mais introduit d'autres calculs.

Le problème est que cette décomposition n'est pas SPD. En effet, la dépendance $\{\text{Ville}, \text{Rue}\} \rightarrow \text{CodePostal}$ est perdue. On peut le vérifier en utilisant l'algorithme de la section 4.3.1.3, mais on peut constater que dans F^+ , il ne peut pas exister une DF telle que $X \rightarrow \text{CodePostal}$ et $\neg(\{\text{Ville}, \text{Rue}\} \subseteq X)$. Donc on ne peut pas projeter ni sur R_1 ni sur R_2 une DF non-triviale qui détermine CodePostal . Donc en faisant la fermeture transitive des dépendances de F^+ projetées sur R_1 et R_2 , on ne pourra jamais trouver une DF non-triviale qui détermine CodePostal .

Cet exemple prouve que l'algorithme de décomposition ne garantit pas une décomposition SPD. Malheureusement, il n'existe pas d'algorithme qui fournisse une décomposition SPI et SPD en FNBC. C'est pourquoi, on pourra préférer l'algorithme de décomposition en 3FN.

Chapitre 5

Transactions et concurrence

5.1 Transactions

5.1.1 Problématique

L'utilisation d'un SGBD doit permettre d'assurer aux utilisateurs *sécurité, intégrité, concurrence et fiabilité*.

- **Sécurité** : l'accès aux données doit être protégé (ex. on doit pouvoir définir des droits d'accès en lecture, en écriture, selon les utilisateurs).
- **Intégrité** : les données doivent respecter les contraintes d'intégrité.
- **Fiabilité** : les données doivent être disponibles et "propres" (ex. la précision numérique annoncée doit être garantie)
- **Concurrence** : plusieurs accès simultanés à la même BD, en respectant la sémantique de chacun et sans défavoriser certains utilisateurs par rapport à d'autres. Notamment, dans le cas d'utilisation interactive, un utilisateur peut très bien "monopoliser la base" alors qu'il ne fait pas d'accès (il réfléchit, dort, téléphone pour se mettre d'accord...). Mais pas seulement en interactif : si le programme X commence et en a pour 2 heures, et que le programme Y arrive et en a pour 10 secondes, on va essayer si c'est possible de ne pas le faire attendre 2h.

Pour cela, on ne permet pas aux utilisateurs de "faire n'importe quoi, n'importe comment, n'importe quand", d'autant plus qu'ils sont plusieurs. Afin de permettre au SGBD de **contrôler** ce qui se passe, on donne un **cadre** pour l'accès à la BD : les **transactions**.

5.1.2 Transactions

5.1.2.1 Définitions

Définition 5.1.1 Une transaction peut être définie selon trois points de vue :

- Une transaction est générée soit par l'exécution d'un programme, soit par une session interactive, soit un mélange des deux. Ici on s'intéresse principalement au premier cas.
- Une transaction peut donc être vue comme une suite finie d'opérations dont certaines sont des accès (lecture ou écriture) à la base.
- Du point de vue de la base de données elle-même, une transaction est un changement d'état qu'on peut noter : $BD_i \xrightarrow{T} BD_{i+1}$. BD_i est l'état initial de la transaction et BD_{i+1} l'état final de la transaction (il se peut que ces deux états soient égaux, par exemple lorsqu'il n'y a que des lectures)

□

Notation : on note les transactions T_i

Notion de granule : Les données de la base que manipulent les transactions sont appelés **granules** (ou parfois items). Les granules peuvent être des relations, des pages, des n-uplets, des attributs... Pour simplifier on considérera que les granules sont *deux à deux disjoints*. Habituellement on les note x, y, z, t, u, v . On notera g pour "n'importe quel granule"

5.1.2.2 Syntaxe des transactions

Une transaction est délimitée par `BEGIN TRANSACTION... END TRANSACTION`. À l'intérieur, elle est composée de trois sortes d'instructions :

- **les lectures/écritures** sur la base :
 - $L_i(x_j)$: la transaction i lit le granule x_j .
 - $E_i(x_j, val)$: la transaction i écrit la valeur val dans le granule x_j . Lorsqu'on ne s'intéresse pas à la valeur écrite on notera $E_i(x_j)$
- **les opérations sans rapport avec la base**, i.e. les instructions classiques ("calculs") d'un langage de programmation. Ces opérations ne nous intéresseront pas car elles n'affectent pas directement l'état de la base de données. Ainsi, $var_k := L_i(x_j); E_i(x_l, var_k)$ et $var_k = L_i(x_j); var_k := 0; E_i(x_l, var_k)$ ne sont pas distingués. Tout ce qu'on retient c'est que la transaction i a lu x_j avant d'écrire x_l .
- **les opérations propres au système transactionnel** :

- $commit_i$: validation de la transaction i (END TRANSACTION, par défaut, génère un commit).
- $abort_i$: abandon de la transaction T_i . Les **effets de T_i** doivent être **défais** ce qui est dans certains cas difficile et coûteux.
- points de reprise : *point de rep 1...si erreur reprise en 1*. On ne les utilisera pas dans ce cours.
- verrouillage explicite : on verra cela plus tard.

La notion de transaction permet de bien délimiter le rôle de l'utilisateur et celui du système dans la "maintenance" de la BD.

- L'utilisateur doit écrire des transactions *bien formées* (syntaxiquement), *correctes*, c-à-d faisant bien ce que veut l'application, et *complètes*.
- le système garantit que l'exécution des transactions se fera en garantissant les bonnes propriétés énoncées plus haut (fiabilité, concurrence...etc). Pour cela, l'exécution doit respecter un certain nombre de propriétés, réunies sous l'acronyme *ACID*.

5.1.2.3 Propriétés ACID des transactions

Une transaction est une *unité de travail cohérente* sur le SGBD. Pour cela, celui-ci doit assurer les propriétés ACID.

Définition 5.1.2 *Les propriétés ACID que doivent respecter les transactions sont les suivantes :*

- **A-tomicité** : *une transaction doit être effectuée en entier (i.e. validée par un "commit"), ou abandonnée ("abort", à la demande de l'utilisateur ou parce que le système ne peut continuer la transaction). Donc si une transaction ne peut pas être validée et si elle a déjà commencé à produire des effets sur la base, il faut défaire ceux-ci, et ce n'est pas forcément évident.*
Une transaction est délimitée par BEGIN TRANSACTION... END TRANSACTION.
La transaction prend la BD dans un état initial et en produit un état final. Tous les états de la BD générés en cours de transaction sont appelés états intermédiaires. Seul l'état final peut être écrit.
- **C-ohérente** : *l'état initial et l'état final doivent être cohérents, i.e. respecter les contraintes d'intégrité. Pour l'état initial c'est une hypothèse (puisque toutes les transactions sont ACID), donc il suffit de vérifier l'état final (on suppose les contraintes statiques).*
- **I-solation** : *une transaction doit s'exécuter indépendamment (niveau logique) des transactions qui s'exécutent en même temps \Rightarrow les états intermédiaires sont cachés aux autres transactions. On veut éviter que les effets des différentes transactions qui s'exécutent en concurrence ne se mélangent n'importe comment.*
- **D-urabilité** : *lorsqu'une transaction est validée, ses effets ne peuvent être perdus (ie. remis en cause).*

□

Les SGBD actuels assurent les propriétés ACID des transactions. Bien entendu, celles-ci sont relativement faciles à assurer lorsque il n'y a pas de concurrence. Sinon, il est plus difficile d'assurer l'isolation et de mettre en œuvre l'atomicité de manière efficace. La durabilité n'est pas un problème théorique de mise en œuvre, mais une propriété dont il faut tenir compte. Enfin, la cohérence dépend des CI, qui ne sont pas toujours toutes disponibles sur les systèmes commerciaux. Donc on va commencer à s'intéresser à l'isolation et ensuite, plus succinctement, à l'atomicité.

Le critère le plus souvent utilisé pour l'isolation est la *sérialisabilité*, c'est ce qu'on va étudier (longuement) dans la section suivante.

5.2 Concurrence - Sérialisabilité

En général, plusieurs transactions sont adressées en même temps à la même base de données. Une façon de faire est de refuser la concurrence, c'est à dire de ne démarrer une transaction que lorsque la précédente est soit confirmée, soit abandonnée et ses effets défais. Mais cela nécessite des temps d'attente qui peuvent être très longs, d'autant plus que cela est très souvent inutile puisque les transactions n'opèrent pas, en grande partie, sur les mêmes données. Donc on va accepter que les transactions s'exécutent simultanément, mais il faut faire attention pour préserver l'ACID (surtout I. On verra ensuite A.)

Pour l'instant on ne se préoccupe pas du commit, donc on peut dire qu'une transaction est une suite d'opérations de lecture et d'écriture (les commandes de début et de fin de transaction sont implicites) :

$$T_i = (op_i^1; op_i^2; \dots; op_i^n) \text{ telle que } op_i^j \in \{L_i(g), E_i(g, val)\}$$

Tout d'abord on définit ce qu'est une *exécution*.

5.2.1 Exécution ou ordonnancement (d'un ensemble de transactions)

Définition 5.2.1 Une *exécution concurrente de transactions* peut se définir formellement de la manière suivante :

- Soit S une suite d'opérations de lecture/écriture effectués par des transactions, on appelle *projection* de S sur T_i la sous-suite de S formée par les opérations de T_i qui sont dans S :

$$Proj_i(S) = (op_k^j \in S, k = i)$$

- Une *exécution des transactions* T_1, T_2, \dots, T_n est une suite d'opérations adressées dans l'ordre à la base, et telles que la projection sur une transaction est exactement la suite (ordonnée) des opérations de la transaction.

$$Ex = (op_{1i}; \dots; op_{jk}; \dots) \text{ telle que } \forall i = 1, \dots, n \quad Proj_i(Ex) = T_i$$

□

Par exemple, soient $T_1 = (L_1(x), E_1(x))$ et $T_2 = (L_2(x), E_2(y))$, la suite $(L_1(x), L_2(x), E_1(x), E_2(y))$ est une exécution de T_1, T_2 , alors que la suite $(E_1(x), L_1(x), L_2(x), E_2(y))$ ne l'est pas (les instructions de T_1 ne sont pas dans le bon ordre).

Exemple d'exécution posant le problème de la sérialisabilité

Soit le programme bancaire :

$$\text{Transfert}(C, C', M) : X = L(C); E(C, X - M); Y = L(C'); E(C', Y + M)$$

Ce programme n'affecte pas la somme des comptes. Pourtant, si on exécute $T_1 = \text{Transfert}(A, B, 100)$ et $T_2 = \text{Transfert}(B, C, 100)$ de la manière suivante, la situation initiale étant $A = B = C = 300$, donc $A + B + C = 900$:

1. $X_1 = L_1(A)$;
2. $E_1(A, X_1 - 100)$;
3. $Y_1 = L_1(B)$;
4. $X_2 = L_2(B)$;
5. $E_2(B, X_2 - 100)$;
6. $Y_2 = L_2(C)$;
7. $E_2(C, Y_2 + 100)$;
8. $E_1(B, Y_1 + 100)$;

On a à la fin $A = 200, B = 400, C = 400$, et donc $A + B + C = 1000$. Notre exécution n'est donc pas correcte. Le problème est que T_2 tient compte d'un état intermédiaire de T_1 , celui où A est débité mais B pas encore crédité. Si on veut s'assurer que cette situation n'arrivera pas, il suffit de s'assurer que l'effet d'une exécution simultanée de plusieurs transactions est le même que celui qui serait produit si on n'acceptait pas la concurrence. Pour cela on introduit le critère de *sérialisabilité* d'une exécution.

5.2.2 Sérialisation des transactions

La définition de la sérialisabilité d'une exécution nécessite un certain nombre de définitions préliminaires.

5.2.2.1 Exécution en série

Définition 5.2.2 Une exécution de transactions est en série si, dans l'exécution, les opérations de chaque transaction sont toutes adjacentes (seules des actions de T_i sont effectuées entre $BEGIN_i$ et END_i). En d'autres termes, les transactions sont exécutées les unes après les autres, sans se mélanger. \square

5.2.2.2 Actions permutable et exécutions équivalentes

Définition 5.2.3 Deux actions de deux transactions différentes, consécutives dans une exécution, sont **permutables** si le résultat de l'ensemble des deux actions ne dépend pas de l'ordre dans lequel elle sont effectuées

Il en résulte que :

- Deux actions sur deux granules différents sont toujours permutable.
- Deux lectures sur un même granule sont permutable
- Une lecture et une écriture sur le même granule ne sont pas permutable,
- En général deux écritures sur le même granule ne sont pas permutable.

ou en résumé : deux opérations sont permutable sauf si elle portent sur le même granule ET au moins une des deux est une écriture. \square

On peut maintenant définir l'équivalence entre deux exécutions.

Définition 5.2.4 Deux exécutions sont **équivalentes** si on peut passer de l'une à l'autre par une suite de permutations (interchangement) de 2 opérations permutable (donc consécutives !). \square

Par exemple, les deux exécutions $L_1(X), L_2(X), E_1(X), E_2(Y)$ et $L_2(X), L_1(X), E_2(Y), E_1(X)$ sont équivalentes (permutation $L_2(X) \leftrightarrow L_1(X)$ puis permutation $E_2(Y) \leftrightarrow E_1(X)$) mais ne sont pas équivalentes à $L_1(X), E_1(X), L_2(X), E_2(Y)$

On en déduit facilement que deux exécutions sont équivalentes si elles le sont pour chaque granule.

5.2.2.3 Sérialisabilité

Définition 5.2.5 Une exécution de T_1, T_2, \dots, T_k est **sérialisable** si elle est équivalente à une exécution en série de T_1, T_2, \dots, T_k . \square

Le problème est qu'on ne va pas s'amuser à faire toutes les permutations possibles pour essayer de trouver une exécution en série... On va utiliser un théorème. Pour cela il nous faut définir la précedence entre deux transaction dans une exécution.

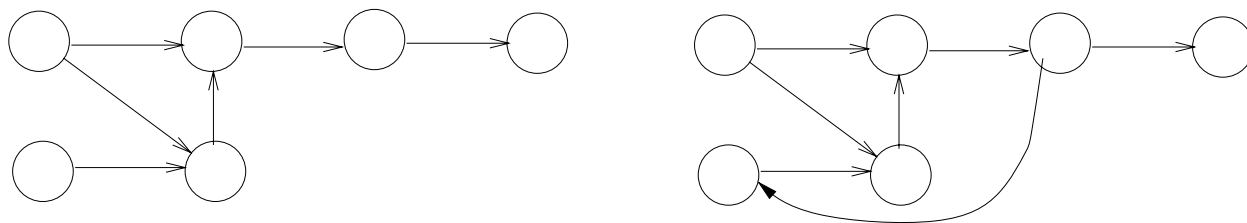
5.2.2.4 Relation de précedence et théorème

Définition 5.2.6 Dans une exécution, T_i **précède** T_j , noté $T_i \rightarrow T_j$, s'il existe au moins un granule x pour lequel une action de T_i sur x précède une action non permutable sur x de T_j .

On en déduit une **relation de précedence entre transactions d'une exécution**, qui peut se représenter par le **graphe de précedence de l'exécution** \square

La meilleure façon de construire le graphe de précedence est d'examiner granule par granule.

Exemple : le graphe de précedence de l'exécution $E_1(X), L_4(Y), L_4(Z), E_2(X), E_5(X), E_3(Y), L_2(Y), L_1(Z), L_3(Z), E_3(Z), L_6(X)$ est celui de la partie gauche de la figure suivante (on note sur chaque arc de précedence les granules qui ont provoqué cette précedence). Si on ajoute une opération à T_4 et qu'on obtient l'exécution $E_1(X), L_4(Y), L_4(Z), E_2(X), E_5(X), E_3(Y), L_2(Y), L_1(Z), L_3(Z), E_3(Z), L_6(X), L_4(X)$, alors le graphe de précedence sera celui de la partie droite de la figure.



Théorème 5.2.1

Une exécution est sérialisable si et seulement si sa relation de précédence est un ordre (en général partiel), i.e. son graphe de précédence est sans circuit (cf. théorie des graphes).

NB : en général on ne connaît pas l'ordre, on sait qu'il existe ou pas. Le problème est d'assurer qu'il existe, donc d'éviter les circuits dans le graphe de précédence. Comment faire... Un moyen, qui permet en plus d'en déduire l'ordre partiel, est la *tri topologique*.

1. Chercher un sommet dont aucun arc ne part. Si la recherche échoue, CIRCUIT.
2. Sinon, sortir le sommet du graphe et tous les arcs qui en sortent, poser le sommet sur le haut de la pile. S'il reste des sommets dans le graphe, recommencer en 1. Sinon, la pile donne un ordre séquentiel équivalent.

Par exemple, si on fait un tri topologique sur le graphe de gauche de la figure précédente, on trouve l'ordre $T_4, T_1, T_3, T_2, T_5, T_6$. Si on le fait sur le graphe de la partie droite, on détectera un circuit.

5.2.3 Contrôle de la concurrence

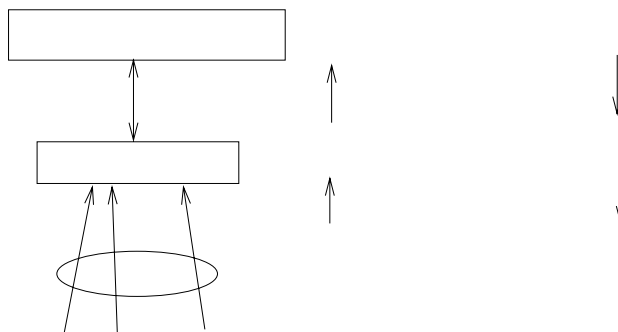
5.2.3.1 Définition

Le contrôleur de concurrence est un module logiciel à qui sont transmises les transactions à exécuter et qui s'occupe de les ordonnancer de manière sérialisable, et d'assurer leur exécution, éventuellement en défaisant les transactions abandonnées.

5.2.3.2 Les différentes stratégies

Pour assurer la sérialisabilité, trois grandes catégories de stratégies sont envisageables : *Prévention*, *Evitement*, *Détection*

- **Prévention** : Le principe de la prévention est de **déterminer a priori l'ordonnancement sérialisable**. Pour cela on détermine les conflits possibles entre transactions et on n'exécute en parallèle que les transactions sans conflit. Le gros problème est que pour cela il faut connaître l'ensemble de tous les accès nécessaires pour une transaction. Pour faire mieux, on utilise le plus souvent l'**estampillage**. Les transactions sont numérotées, par ex. selon l'ordre d'arrivée. Cette numérotation est donc un ordre séquentiel équivalent *prédéfini*. Donc, ensuite, sur chaque granule, lorsque deux actions non-permutables sont effectuées dans l'ordre inverse de l'estampillage, on abandonne une des deux transactions. On verra ce mécanisme en détail plus loin.
- **Détection** : On accepte toutes les opérations (méthode optimiste) et on teste à chaque étape si il y a un circuit. Si c'est le cas, on abandonne une ou des transactions. Le fait d'avoir à gérer ce graphe et à défaire des transactions dès qu'il y a un conflit fait que cela n'est jamais mis en œuvre.
- **Evitement** : C'est cette technique qu'on va détailler car en pratique c'est celle là qui est implantée dans les systèmes. Le principe est de **faire attendre des transactions** en cours d'exécution **avant qu'un circuit ne se produise** dans le graphe de précédence. Pour cela on utilise la technique du **verrouillage**. Un verrou permet d'"éviter" un accès incompatible à un granule si celui-ci est "utilisé" par une autre transaction. Ce mécanisme est utilisé depuis longtemps dans les systèmes d'exploitation.



Dans un mécanisme de verrouillage, deux choses sont importantes :

1. La façon dont le système met en attente ou accorde les verrous à une transaction.
2. La façon dont les transactions demandent les verrous = protocole. Le plus connu est le protocole V2P, car il assure la sérialisabilité.

ATTENTION : dans la suite, on va supposer que les transactions suivent un protocole minimum, c'est à dire qu'elles demandent un verrou sur un item avant d'y accéder (On ne permet pas qu'elles accèdent à une donnée non verrouillée par elle). De même, tous les verrous demandés par une transactions seront relâchés par la suite.

5.3 Le verrouillage deux phases

5.3.1 Deux types de verrou

On a vu que, du point de vue de la sérialisabilité, ce qu'il faut éviter ce sont les actions incompatibles sur le même granule (c'est pour celles-ci qu'on doit avoir toujours le même ordre entre transactions). Donc si on a un seul type de verrou, le gestionnaire de verrous va bloquer une transaction qui lit X dès qu'une autre a un verrou dessus pour la lire. Or ce n'est pas nécessaire puisque les deux lectures peuvent être permutées. Donc il faut deux types de verrous :

- l'un avant de faire une lecture, pour prévenir que les autres transactions ne fassent pas une écriture entre temps = VERROU-P (verrou partagé, S-lock en anglais). Le verrou est partagé car on peut donner autant de verrous L qu'on veut sur le même granule.
- l'un avant de faire une écriture, pour prévenir toute opération sur le granule = VERROU-X (verrou eXclusif, X-lock en anglais). Le verrou est exclusif car il ne peut être posé que si le granule n'est pas verrouillé par une autre transaction, et interdit tout autre verrou sur le même granule.

Donc, finalement, dans les transactions, "on" introduit les opérations suivantes :

- $VP_i(x)$: demande de verrou partagé sur x par T_i . Si il existe un verrou-X sur x , la transaction T_i est mise en attente, sinon la transaction est mise dans la liste des verrous-P de x .
- $VX_i(x)$: demande de verrou exclusif sur x par T_i . Si il existe un verrou (X ou P) sur x , la transaction T_i est mise en attente, sinon la transaction obtient LE verrou-X de x . Cependant, si T_i est la seule transaction à posséder un verrou-P lorsqu'elle demande un verrou-X, on peut lui accorder, c'est ce qu'on appelle une promotion.
- $D_i(x)$: déverrouillage de x par T_i .

Mais le verrouillage à lui tout seul (sans protocole) ne garantit pas la sérialisabilité. Il suffit pour s'en convaincre de reprendre l'exemple du transfert bancaire en verrouillant les granules juste avant l'accès et en déverrouillant juste après l'accès, on permet ainsi le même ordonnancement qui n'est pas sérialisable malgré l'utilisation de verrous.

5.3.2 Le protocole V2P

Définition 5.3.1 Une transaction suit le **protocole de verrouillage en deux phases (V2P)** si le premier déverrouillage qu'elle demande vient après qu'elle ait demandé et obtenu tous les verrous dont elle a besoin.

□

On a donc deux phases bien distinctes, la première, *phase d'acquisition* où la transaction demande tous les verrous dont elle aura besoin, la deuxième, *phase de déverrouillage*, où elle les relâche tous, et ne pourra plus en demander d'autre. L'avantage du protocole V2P est qu'il garantit la sérialisabilité, comme l'exprime le théorème suivant.

Théorème 5.3.1

Toute exécution de transactions respectant toutes le protocole V2P est sérialisable.

ATTENTION : ce résultat marche aussi avec un seul type de verrou.

Preuve : à faire en exercice, par l'absurde.

L'intuition de ce théorème est la suivante. Une fois qu'une transaction a pris tous ses verrous, elle possède un état cohérent de la BD (puisque les autres transactions respectent elles aussi V2P) qu'elle peut lire et écrire à sa guise jusqu'au déverrouillage, sans interférences, même si elle a mis le temps pour obtenir cet état. Finalement, on peut dire qu'elle verrouille toutes les données nécessaires avant de travailler, d'un point de vue logique, et qu'elle n'en libère aucune avant d'avoir fini.

Si on reprend l'exemple du transfert bancaire, on voit que, si les transactions respectent le protocole V2P, l'exécution proposée (non-sérialisable) n'est pas possible.

Si le protocole V2P est avantageux, il a aussi ses défauts. Le premier est qu'il est restrictif : il existe des exécutions sérialisables qu'on ne pourrait pas obtenir à l'aide du verrouillage V2P (en d'autres termes, la réciproque du théorème n'est pas vraie)

5.3.3 Interblocage

5.3.3.1 Problème

Le gros problème du verrouillage est qu'on impose de faire attendre les transactions. On ne sait pas jusqu'à quand, cela dépend des autres transactions et c'est cela le problème. Dans le cas général, on n'est même pas sûr que la transaction ne va pas attendre indéfiniment. C'est particulièrement vrai dans le cas où les transactions respectent le protocole V2P, puisque dans ce cas on essaie de prendre le maximum de verrous avant d'en relâcher un seul.

Exemple : soit l'ordonnancement suivant, qui respecte V2P.

$$VP_1(x), VX_2(z), VX_3(y), VP_2(y), VP_1(z), VX_3(x), \dots D_1(x) \dots$$

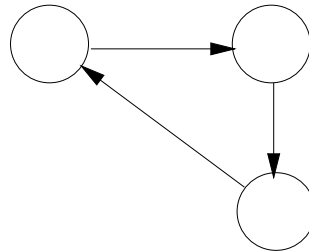
Habituellement, les verrous accordés aux transactions ainsi que les demandes en attente sont stockées dans une table, la *table des verrous*. Pour l'ordonnancement précédent, la table des verrous à la suite de la demande $VX_3(X)$ serait la suivante :

Granule	Verrou-X	Verrous-P	Attente Verrou-X	Attente Verrou-P
x		T_1	T_3	
y	T_3			T_2
z	T_2			T_1

On voit que la situation est bloquée!! En effet, la demande de T_3 ne pourra être satisfaite que lorsque T_1 libèrera le verrou-P qu'il a sur x . Or elle ne peut le faire que lorsqu'elle aura obtenu tous les verrous qu'elle demande (protocole V2P). Pour cela, il faut qu'elle obtienne le verrou sur z . Mais pour la même raison, T_2 ne peut libérer le verrou-X qu'elle a sur z , car il faut d'abord qu'elle obtienne préalablement le verrou sur

y . Or elle ne l'obtiendra que si T_3 libère son verrou-X, ce qu'il ne peut faire ... que lorsqu'il aura obtenu le verrou-X qu'il demande sur x ! La boucle est bouclée et la situation bloquée.

Si on représente la relation d'attente entre les transaction (une relation en attend une autre si elle demande un verrou sur un granule et qu'elle est mise en attente parce que l'autre possède un verrou sur ce même granule - donc au moins un des deux verrous est un verrou-X -) par un graphe, qu'on appelle le **graphe d'attente**, on obtient, pour cet exemple, le graphe suivant. **Attention !** Ce graphe ne doit pas être confondu avec le graphe de précedence vu à la section 5.2.2.3



Définition 5.3.2 Il y a **interblocage** lorsque, par transitivité de la relation d'attente, au moins une transaction attend une transaction qui l'attend elle même. Il y a donc un interblocage ssi il existe un circuit dans le graphe d'attente. \square

5.3.3.2 Solutions au problème d'interblocage

Pessimiste : prévention. Optimiste : détection.

- **Prévention** : les transactions sont numérotées par ordre d'arrivée. Deux possibilités, préemptif ou non. Soit T_i désirant un verrou détenu par T_j .
 - préemptif : si T_j est plus jeune que T_i , T_i prend le verrou et T_j est abandonnée. Sinon, T_i attend.
 - non préemptif : si T_j est plus jeune que T_i , T_i attend. Sinon T_i est abandonnée

Dans les deux cas, à chaque fois qu'on abandonne une transaction, on la relance avec le même numéro. Donc une transaction finira toujours par être la plus vieille, il y a toujours au moins une transaction qui s'exécute en entier. C'est pourquoi les interblocages sont impossibles mais cela conduit à abandonner et donc défaire des transactions inutilement.

- **Détection** : comme pour la sérialisation, on laisse faire et on détecte les circuits. Dès qu'il y a un interblocage, l'ordonnanceur choisit une victime à abandonner. Pour cela on utilise des critères comme le nombre d'opérations déjà effectuées par les transactions en conflit, le nombre d'opérations qu'il leur reste à accomplir, le nombre d'abandons nécessaires (cascade)...

5.4 Sérialisation par estampillage (time-stamping)

5.4.1 Principe

C'est le principe de sérialisation par prévention, c'est à dire qu'on décide avant exécution de l'ordre dans lequel seront sérialisées les transactions. Pour cela on affecte à chaque transaction T une estampille $TS(T)$, habituellement correspondant à la date d'arrivée dans le système. Donc on va laisser une opération s'effectuer si et seulement si elle respecte cet ordre.

5.4.2 Mécanisme

Etiquetage des granules

A chaque granule g on associe deux étiquettes, une étiquette lecture $EL(G)$ et une étiquette écriture $EE(g)$.

- $EL(g)$ contient l'estampille de la transaction la plus "jeune" à avoir lu g .
- $EE(g)$ contient l'estampille de la transaction la plus "jeune" ayant déjà écrit sur g .

Règles de lecture et d'écriture

- Soit T une transaction qui tente de lire le granule g .
 - si $TS(T) \geq EE(g)$, la lecture est acceptée et $EL(g) \leftarrow \text{Max}(EL(G), TS(T))$.
 - sinon la lecture est refusée (T arrive trop tard...) et T est abandonnée.
 - Soit T une transaction qui veut écrire g .
 - si $TS(T) \geq EE(G) \wedge TS(T) \geq EL(G)$, alors l'écriture est acceptée et on modifie l'étiquette $EE(G) \leftarrow TS(T)$.
 - sinon l'écriture est refusée (T arrive trop tard...) et T est abandonnée.
 - Dans les deux cas, si T est abandonnée, elle repart avec une nouvelle estampille (donc différent de l'estampille utilisée pour le verrouillage) plus jeune, sinon elle ne passera jamais...
- Par exemple, l'ordonnancement suivant serait accepté si $TS(T_1) < TS(T_2)$.

$$L_1(b), L_2(b), E_2(b), L_1(a), L_2(a), E_2(a)$$

On aurait alors $EL(a) = EE(a) = EL(b) = EE(b) = TS(T_2)$

Si arrive ensuite une demande d'écriture de b par T_1 , celle-ci sera refusée par la règle d'écriture et T_1 sera abandonnée puis relancée avec une nouvelle estampille $TS(T_1) > TS(T_2)$ et le problème ne se posera plus.

Théorème 5.4.1

Toute exécution utilisant l'estampillage des transactions et respectant les deux règles de lecture et d'écriture ci-dessus est sérialisable

NB : Il existe des ordonnancement qui ne sont pas V2P et qui sont admis par estampillage et vice-versa. L'exemple $L_1(b), L_2(b), E_2(b), L_1(a), L_2(a), E_2(a)$ n'est pas V2P...

5.4.3 Règle de Thomas

Considérons l'ordonnancement suivant

$$L_1(b), E_2(b), L_1(a), L_2(a), E_2(a), E_1(b)$$

La règle d'écriture précédente va refuser la dernière écriture et T_1 va être abandonnée. Or on peut montrer que cet abandon n'est pas nécessaire.

En effet, si T veut écrire une valeur v sur g , et que $EE(g) > TS(T)$ MAIS QUE $TS(T) > EL(g)$, v est obsolète, mais de toute façon, v n'aurait jamais été lue puisqu' aucune transaction qui aurait dû lire v , ie. plus jeune que T , n'est passée avec succès sur g car $TS(T) > EL(G)$. Donc v n'aurait pas été lue et ne sera jamais lue. Donc, il n'y a pas de problème de sérialisabilité. En revanche, la valeur actuelle de g (celle écrite par la transaction dont l'estampille figure dans $EE(g)$) doit rester puisqu'elle est postérieure à v .

D'où la nouvelle règle d'écriture :

- si $TS(T) \geq EE(G) \wedge TS(T) \geq EL(G)$, l'écriture est acceptée et $EE(G) \leftarrow TS(T)$.
- si $TS(T) < EE(G) \wedge TS(T) \geq EL(G)$, rien.
- sinon l'écriture est refusée (T arrive trop tard...) et T est abandonnée.

La règle de lecture reste la même.

Si on applique la règle de Thomas à l'ordonnancement $L_1(b), E_2(b), L_1(a), L_2(a), E_2(a), E_1(b)$, T_1 ne sera pas abandonnée.

5.5 Recouvrabilité

On a vu que, dans de nombreux cas, une transaction est abandonnée et doit donc être défait, ce qui veut dire **annuler ses effets**, faire comme si la transaction n'avait jamais été exécutée. Or ceci est loin d'être évident car d'autres choses se sont passées entre temps, du fait de la concurrence.

Rappel : Lorsqu'une transaction est validée, c'est définitif (atomicité des transactions)

Exemple : Etat initial : $X = Y = 1$ Exécution : $E_1(X, 2), A = L_2(X), E_2(Y, A)(*), Abort_1$

Ici, $E_2(Y, 2)$ est faux, puisque comme T_1 aborte, il faut mettre la valeur précédente, ie. $Y := 1$. Donc on doit abandonner T_2 aussi. Mais si on avait un $Commit_2$ en (*), ce ne serait pas possible. Il faut *absolument* éviter cela. Pour cela, on exige que toute exécution soit *recouvrable*.

Définition 5.5.1 Soit un ordonnancement de transactions T_1, \dots, T_n

- On dit que T_j **lit depuis** T_i si $\exists x, T_j$ lit x après que T_i a écrit x ET T_i n'est pas abandonnée au moment où T_j lit x ET $\forall T$, si T a écrit x après T_i et avant la lecture, elle a abandonné. En d'autres termes " T_i est la transaction ayant effectivement écrit la valeur lue par T_j ".
- Une exécution est **recouvrable** si, pour toute transaction T_j lisant depuis une transaction T_i ,

$$T_j \text{ committe} \rightarrow T_i \text{ a committé avant } T_j$$

□

Ainsi on évite à une transaction de committer en ayant lu des données "sales", c'est à dire des données de transaction non committées, donc dont on n'est pas sûr qu'elles vont persister. Le problème est que cela ne suffit pas à empêcher les abandons en cascade qui sont coûteux, même si ils n'introduisent pas d'incohérence. En effet, si une transaction a lu une donnée sale, elle peut être amenée à abandonner, et ainsi de suite... Donc on va essayer de les empêcher.

Définition 5.5.2 Une exécution est **sans abandon en cascade** (sans cascade) si pour toute transaction T_j lisant depuis une transaction T_i en t ,

$$T_j \text{ committe} \Rightarrow T_i \text{ a committé avant } t$$

□

C'est très restrictif car chaque lecture de X doit attendre que toutes les transactions qui ont écrit X soient committées ou abandonnées. On empêche ainsi toute lecture de données sale. Cependant, ce n'est pas encore parfait, car même si on ne fait qu'un seul abort, encore faut il pouvoir le faire simplement. Or le moyen le plus simple, ie. ce qu'on aimerait bien, est de stocker l'image avant d'une écriture, ie. la valeur qu'avait x juste avant la dernière écriture sur x en date, de manière à pouvoir restaurer cette image en cas d'abandon.

Exemple : initialement $x = y = 0$ $E_1(X, 1), E_1(Y, 3), E_2(Y, 1), Commit_1, L_2(X), Abort_2$

Images avant : $x : (T_1, 0), \quad y : (T_2, 3)$

Cette exécution est clairement sans cascade. Mais comment faire l'abort ? La valeur à restaurer pour y est effectivement l'image avant de $E_2(y, 1)$, ie. 3. Mais dans le cas suivant cela ne marche plus :

Exemple : initialement $x = 1$ $E_1(X, 2), E_2(X, 3), Abort_1, Abort_2$.

Ici, lorsqu'on aborte T_2 , il faut restaurer $x := 1$, puisque c'est la valeur initiale. Or, dans l'image avant de $E_2(X, 3)$ on a $x = 2$, et en abortant T_1 on n'a rien fait.

\Rightarrow Si on veut vraiment pouvoir toujours utiliser l'image avant, il faut imposer des exécutions strictes

Définition 5.5.3 Soit un ordonnancement de transactions T_1, \dots, T_n

- On dit que T_j **écrit après** T_i si $\exists x, T_j$ écrit x après que T_i a écrit x ET T_i n'est pas abandonnée au moment où T_j écrit x ET $\forall T$, si T a écrit x après T_i et avant l'écriture par T_j , elle a abandonné. En d'autres termes " T_i est la transaction ayant effectivement écrit la valeur que remplace T_j ".
- Une exécution est **stricte** si elle est sans cascade ET SI pour toute transaction T_j "écrivant après" une transaction T_i en t ,

$$T_j \text{ committe} \Rightarrow T_i \text{ a committé avant } t$$

□

Cette définition est strictissime (elle réduit énormément la concurrence), c'est pourquoi, si on veut accepter des exécutions non-strictes, on utilise une structure plus riche que les images avant : le LOG ou Journal, qui garde trace de tout ce qui s'est fait pendant une session.

LOG : { (id_tr, item, valeur_avant, valeur_apres) }

Dans l'exemple cela donne :

initialement X vaut 1...

$E_1(X, 2), E_2(X, 3), Abort_1, Abort_2$.

LE log :

$(T_1, X, 1, 2)$

$(T_2, X, 2, 3)$

et là, il suffit de dérouler le log à l'envers