INFORMATIQUE COMMUNE

Introduction aux Bases de Données Relationnelles

Serge Abiteboul
Inria, ENS Cachan, Conseil national du numérique
serge.abiteboul@inria.fr

Benjamin Nguyen
Université de Versailles St-Quentin-en-Yvelines, Inria
benjamin.nguyen@uvsq.fr

Yannick Le Bras
Mathématiques MPSI, Lycée Montesquieu, Le Mans
yannick.le-bras@prepas.org

CPGE

Logic is the beginning of wisdom, not the end.
Mr. Spock, Star Trek

Table des matières

1 Introduction

Nous allons parler de systèmes informatiques qui nous aident à gérer des données. Nous avons donc, d'un côté, un serveur de données quelque part sur la Toile, avec des disques magnétiques [1] et leurs pistes qui gardent précieusement des séquences de bits, des structures d'accès compliquées comme des index ou des arbres-B, des hiérarchies de mémoires avec leurs caches et, de l'autre, un utilisateur. Supposons que le serveur soit celui d'IMDb qui gère une base de données sur le cinéma. Supposons que l'utilisateur, disons Alice, veuille savoir quels films ont été réalisés par Alfred Hitchcock. Pour ce faire, elle spécifie des mots-clés ou remplit les champs d'un formulaire proposé par IMDb. Sa question voyage depuis son navigateur jusqu'au serveur de données. Là, cette question est transformée en un programme peut-être complexe qui s'exécute pour obtenir la réponse. Ce qui est important : ce programme, Alice n'a pas envie de l'écrire ; elle n'a pas à l'écrire.

Le système élémentaire qui permet de gérer des données est un système de fichiers. Un fichier est une séquence de bits qui peut représenter une chanson, une photo, une vidéo, un courriel, une lettre, un roman, etc. Votre ordinateur personnel et votre téléphone stockent leurs données dans des systèmes de fichiers. Et parfois quand vous ne savez plus où vous avez mis quelque chose, vous faites des *recherches* dans ces système de fichiers. C'est rudimentaire. Un moteur de recherche de la Toile ne fait pas autre chose, seulement il le fait sur un système de fichiers à l'échelle de la planète. Dans ce chapitre, nous parlerons de systèmes qui gèrent aussi des données mais qui sont bien plus sophistiqués que les systèmes de fichiers : les systèmes de gestion de bases de données. Ce sont des logiciels complexes, résultats de dizaines d'années de recherche et de développement. Ils permettent à des individus ou des programmes d'exprimer des requêtes pour interroger des bases de données ou pour les modifier. Nous nous focaliserons ici sur les plus répandus d'entre ces systèmes, les systèmes relationnels, parmi lesquels nous trouvons des logiciels commerciaux très répandus comme celui d'Oracle et des logiciels libres très utilisés comme MySQL.

Dans ce livre, nous couvrons le programme des classes préparatoires scientifiques, toutefois il nous est paru indispensable d'aller au delà d'une interprétation stricte du programme, pour permettre au lecteur de comprendre les fondements de la théorie des bases de données, sans laquelle il ne serait qu'un simple utilisateur de SGBD. Nous indiquons les éléments qui sont à la limite du programme par la notation †, et ceux qui sont au delà par la notation ‡.

1.1 Les principes et l'architecture

Au fil des ans, trois grands principes ont émergé qui ont façonné le domaine de la gestion de données :

— Abstraction : Un système de gestion de bases de données sert de médiateur entre des

1. À l'heure actuelle, un nombre croissant de serveurs utilisent désormais des disques basés sur de la mémoire Flash, appelés Solid State Drive (SSD). Les travaux historiques sur les systèmes de gestion des bases de donnés font l'hypothèse de l'utilisation de disques magnétiques. L'optimisation des SGBD aux disques SSD est du domaine de la recherche actuel (voir [**?**]).

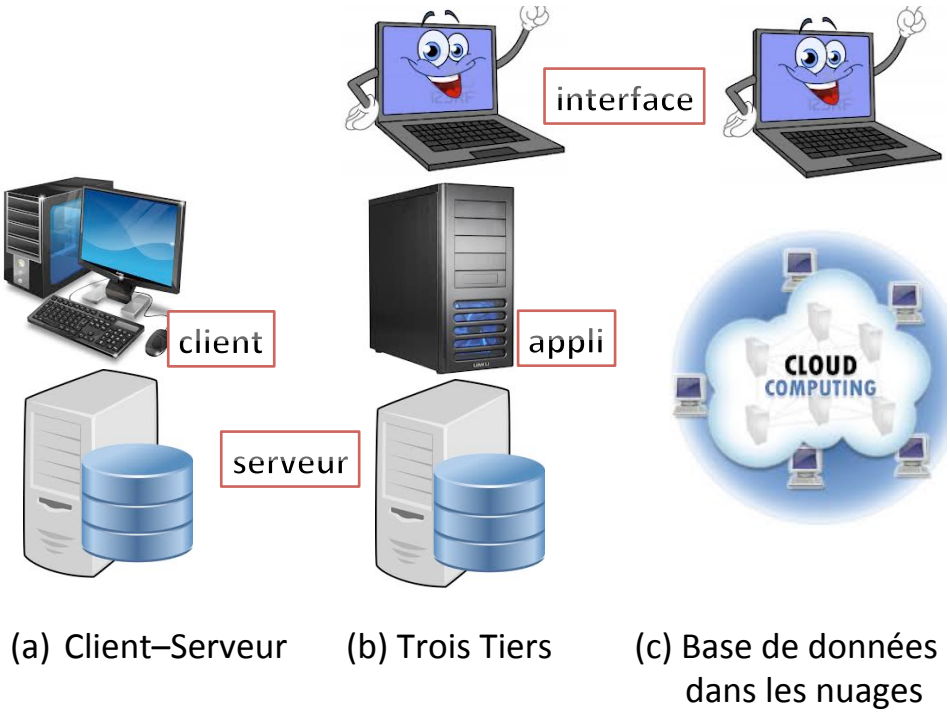

FIGURE 1.1– Architectures principales

individus et des machines. Pour mieux s'adapter aux individus, il doit organiser et présenter les données de façon intuitive et permettre de les manipuler en restant à un niveau abstrait sans avoir à considérer des détails d'implémentation.

— Indépendance : nous distinguons trois niveaux, physique, logique et externe, que l'on essaie de rendre le plus indépendants possible. Au niveau externe, nous trouvons les vues d'utilisateurs particuliers qui partagent la base de données. Chaque vue est adaptée aux besoins de l'utilisateur. Au niveau logique, nous trouvons une organisation unique des données typiquement dans le modèle relationnel que nous détaillons plus loin. Et au niveau physique, les détails de l'organisation sur disque et de structures comme des index dont le rôle est d'accélérer les calculs. Le but est de pouvoir modifier un niveau (par exemple, ajouter un nouvel utilisateur avec de nouveaux besoins) sans modifier les autres niveaux.

— Universalité : Ces systèmes visent à capturer toutes les données d'une entreprise, d'un groupe, d'une organisation quelconque, pour tout type d'applications. Il leur faut donc offrir des langages de développement d'applications puissants et une gamme de fonctionnalités très riche. Des limites de cette universalité existent aujourd'hui pour les systèmes relationnels : énormément de données moins structurées sont par exemple gérées dans des systèmes de fichiers.

Mentionnons brièvement les architectures les plus répandues de systèmes de gestion de données. Voir Figure **??**. Une première architecture est celle des systèmes client/serveur. La base de données est gérée sur un serveur. L'application tourne sur une autre machine, le client. De plus en plus, cette architecture se complique avec l'introduction d'un troisième "tiers" (ce qui signifie en réalité "niveau" en anglais), une machine qui gère l'interface, typiquement un navigateur Web sur une tablette ou un laptop.

Nous pouvons noter différentes évolutions générées par des améliorations dans les matériels disponibles :

— l'accroissement des performances notamment fondées sur les mémoires vives de plus en plus massives, et des mémoires flash encore plus massives ;

Film			Séance		
Titre	**Réalisateur**	**Acteur**	**Titre**	**Salle**	**Heure**
Casablanca	*M. Curtiz*	*Humphrey Bogart*	*Casablanca*	*Lucernaire*	*19 :00*
Casablanca	*M. Curtiz*	*Peter Lore*	*Casablanca*	*Studio*	*20 :00*
Les 400 coups	*F. Truffaut*	*J.-P. Leaud*	*Star Wars*	*Sel*	*20 :30*
Star Wars	*G. Lucas*	*Harrison Ford*	*Star Wars*	*Sel*	*22 :15*

Figure 1.2– Une base de données relationnelles

— l'utilisation de plus en plus de parallélisme massif dans des grappes de machines pour traiter d'énormes volumes de données. On parle parfois de "big data" ;

— pour simplifier la gestion de données, on tend à la déporter dans les nuages (le cloud), c'est-à-dire à mettre ses données dans des grappes de machines gérées par des spécialistes comme Amazon.

1.2 Le calcul et l'algèbre relationnels

Un système de gestion de bases de données doit aussi proposer un langage, pour exprimer des requêtes, facilement utilisable par des êtres humains. Ces exigences forment le point de départ du modèle relationnel [Cod70, AHV95] proposé par Ted Codd, un chercheur d'IBM, dans les années 1970. Des mathématiciens avaient développé à la fin du 19e siècle (bien avant l'invention de l'informatique et des bases de données) la Logique du premier ordre, pour formaliser le langage des mathématiques. Codd a eu l'idée d'adapter cette Logique pour définir un modèle de gestion de données, le modèle relationnel.

Dans le modèle relationnel, les données sont organisées en tableaux à deux dimensions que nous appellerons des *relations*. À la différence des mathématiciens, nous supposons les relations de taille finie. Comme illustration, nous utiliserons une base de données consistant en une relation Film et une relation Séance (voir Figure 1.2). Une ligne de ces relations est appelée un *n-uplet* (ou *tuple* en anglais). Par exemple, $\langle$*Star Wars*, *Sel*, *22 :15*$\rangle$ est un n-uplet d'arité 3, c'est-à-dire un triplet, dans la relation Séance. Les colonnes ont des noms, appelés attributs, comme **Titre**. Un n-uplet est noté de la manière suivante : $\langle$ "*Casablanca*", "*M. Curtiz*", "*umphrey Bogart*"$\rangle$

Les données sont interrogées en utilisant comme langage le calcul relationnel. Le calcul relationnel (très fortement inspiré de la Logique du premier ordre) s'appuie sur des noms qui représentent des relations comme Film ou Séance, des entrées de ces relations comme *Star Wars*, des variables comme t, h, et des symboles logiques, $\wedge$ (et), $\vee$ (ou), $\neg$ (non), $\Rightarrow$ (implique), $\exists$ (existe), $\forall$ (pour tout). À partir de ces ingrédients, des formules logiques peuvent être construites telles que :

$$q_{HB} = \exists\, t, r;\ (\textsc{Film}(t, r, \text{"}\textit{Humphrey Bogart}\text{"}) \wedge \textsc{Séance}(t, s, h))$$

Si cela vous parait cryptique, en français, cela se lit : il existe un titre t et un réalisateur r tels que le n-uplet $\langle$ t, r, "*Humphrey Bogart*" $\rangle$ se trouve dans la relation Film, et le n-uplet $\langle$ t, s, h $\rangle$ dans Séance. Observez que s et h ne sont pas quantifiées dans la formule précédente ; nous dirons que ces deux variables sont *libres*. La formule q_{HB} peut être vue comme une requête du calcul relationnel. Elle se lit alors : donnez-moi les salles s et les horaires h, s'il existe un réalisateur r et un titre t tels que... En d'autres termes,

Où et à quelle heure puis-je voir un film avec Humphrey Bogart ?. Pour faciliter la lisibilité de ces formules logiques, on peut exprimer une requête en définissant précisément la forme du n-uplet (nommé *res* dans la suite du document) en lui affectant les variables libres que l'on cherche à obtenir, ce qui est implicite dans l'expression précédente :

$$\{res(s,h) | \exists\, t, r;\ (\textsc{Film}(t, r, \text{“}\textit{Humphrey Bogart}\text{”}) \wedge \textsc{Séance}(t, s, h)\}$$

L'intérêt de ce procédé est qu'on peut ensuite réutiliser les n-uplets résultats exactement comme s'il s'agissait d'une relation. C'est ce qu'on appelle le mécanisme des *vues*. Une vue est une relation, qui au lieu d'être stockée dans la base de données, est définie intentionnellement. Un utilisateur de la base de données peut l'utiliser comme n'importe quelle autre relation.

Ce langage, le calcul relationnel, que nous présentons en Chapitre 2 permet d'exprimer des questions dans une syntaxe qui évite les ambiguïtés de nos langues naturelles. Si elles pouvaient aimer, les machines aimeraient la simplicité et la précision du calcul relationnel. En pratique, elles utilisent le langage SQL (pour Structured Query Language) qui exprime différemment les mêmes questions. Par exemple la question précédente s'exprime en SQL comme :

```
SELECT salle, heure
FROM Film, Seance
WHERE   Film.titre = Seance.titre AND acteur= "Humphrey Bogart"
```

C'est presque compréhensible. Non ? Et qu'Alice s'exprime en français ou qu'elle utilise une interface graphique, le système transforme sa question en requête [2] SQL. Nous présentons le langage SQL au Chapitre 4.

La requête (question) du calcul relationnel précédente (ou en SQL) précise bien ce qu'Alice demande. Cette question a un sens précis : une sémantique. Elle définit [3] une réponse, un ensemble de n-uplets. Ce que la question ne dit pas c'est *comment calculer la réponse*. Pour le *comment*, on utilise l'algèbre relationnelle introduite par Codd, et que nous présentons en Chapitre 3. Une étape importante consiste à transformer une question du calcul en une expression algébrique qui permet de calculer la réponse à cette question.

L'algèbre relationnelle consiste en un petit nombre d'opérations de base qui, appliquées à des relations, produisent de nouvelles relations. Ces opérations peuvent être composées pour construire des expressions algébriques de plus en plus complexes. Pour répondre à la question qui nous sert d'exemple, il nous faudra trois opérations, la jointure, la sélection et la projection, que nous composerons dans l'expression suivante de l'algèbre relationnelle :

$$E_{HB} = \pi_{\texttt{salle},\texttt{heure}}(\pi_{\texttt{titre}}(\sigma_{\texttt{acteur}=\text{“}\textit{Humphrey Bogart}\text{”}})(\textsc{Film})) \bowtie \textsc{Séance})$$

Nous pourrons suivre l'évaluation de cette expression algébrique en Figure 1.3. L'opération de sélection, dénotée σ, filtre une relation, ne gardant que les n-uplets satisfaisant une condition, ici acteur = “*Humphrey Bogart*”. L'opération de projection, dénotée π, permet aussi de filtrer de l'information d'une relation mais cette fois en éliminant des colonnes. L'opération peut-être la plus exotique de l'algèbre, la *jointure*, dénotée $\bowtie$, combine des n-uplets de deux relations. D'autres opérations non illustrées ici permettent de faire l'union et la différence entre deux relations ou de renommer des attributs. La puissance de l'algèbre relationnelle tient de la possibilité de composer ces opérations. C'est ce que nous avons fait dans l'expression algébrique E_{HB} qui permet d'évaluer la réponse à la question q_{HB}.

Notre présentation est rapide mais il est important que le lecteur comprenne l'intérêt de l'algèbre. Nous reviendrons là dessus plus en détail au Chapitre 3. En effet, il est relativement simple d'écrire un programme qui évalue la réponse à une question du calcul relationnel. Il est plus délicat d'obtenir un programme qui calcule cette réponse efficacement. L'algèbre relationnelle découpe le travail. Un programme particulier très efficace peut être utilisé pour chacune des opérations de l'algèbre ; le résultat est obtenu en composant ces programmes. L'efficacité provient notamment de ce que les opérations considèrent des ensembles de n-uplets plutôt que les n-uplets un à un.

2. SQL va plus loin que le calcul relationnel. Par exemple, il permet d'ordonner les résultats et d'appliquer des fonctions simples comme la somme ou la moyenne.

3. Nous ne préciserons pas comment dans ce cours.

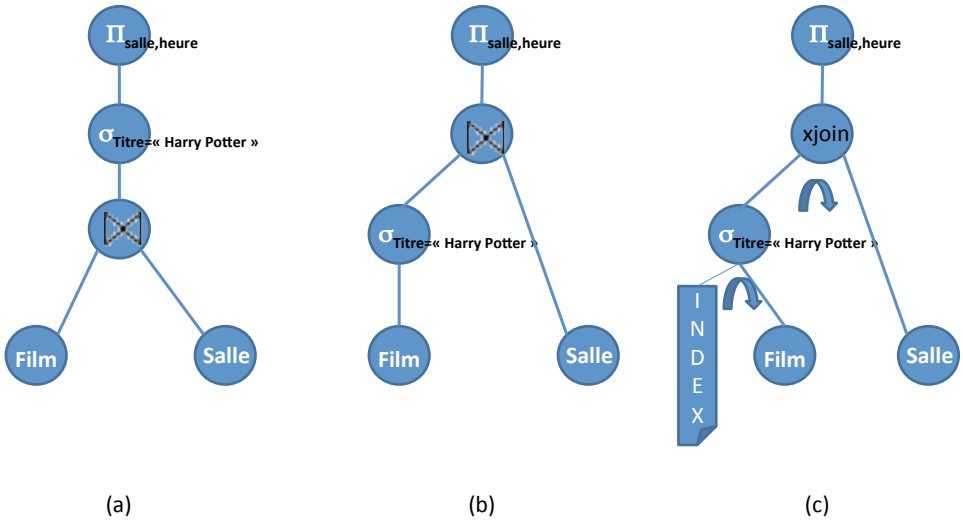

FIGURE 1.3– Des plans d'exécution algébriques

Le Théorème de Codd

Codd a démontré le théorème suivant :

Théorème 1.2.1. [Codd] : une question est exprimable en calcul relationnel si et seulement si elle peut être évaluée avec une expression de l'algèbre relationnelle, et il est facile de transformer une requête du calcul en une expression algébrique qui évalue cette requête.

Que nous apprend ce théorème ? Pas grand-chose du point de vue des mathématiques. Le calcul relationnel est emprunté aux logiciens. Une algébrisation (légèrement différente) avait même déjà été proposée par Tarski. Mais d'un point de vue informatique, Codd a posé les bases de la médiation autour des données entre individus et machines. Grâce à son résultat, nous savons que nous pouvons *exprimer* une question en calcul relationnel, qu'un système peut *traduire* cette question en expression algébrique et *calculer efficacement* sa réponse. Pourtant, quand Codd proposa cette approche, la réaction des ingénieurs qui géraient alors de gros volumes de données et de grandes applications, fut unanime : *trop lent ! ça ne passera pas à l'échelle.* Ils se trompaient. Pour traduire l'idée de Codd en une industrie de milliards de dollars, il manquait l'optimisation de requête. Après des années d'effort, les chercheurs sont parvenus à faire fonctionner les systèmes relationnels avec des temps de réponse acceptables. Avec ces systèmes, le développement d'applications gérant des données devenait beaucoup plus simple ; cela se traduisait par un accroissement considérable de la productivité des programmeurs d'applications gérant des gros volumes de données.

1.3 L'optimisation de requête†

Il existe une infinité d'expressions algébriques qui évaluent une même requête. Si elles sont syntaxiquement différentes, elles définissent la même question. D'un point de vue sémantique, elles sont équivalentes. Optimiser une requête consiste à la transformer en une autre qui donne les mêmes réponses, mais qui soit la moins coûteuse possible (typiquement en temps). D'un point

de vue pratique, il nous faut choisir un *plan d'exécution*, c'est-à-dire une expression algébrique avec des précisions sur l'algorithme à utiliser pour évaluer chacune des opérations. Un plan d'exécution, c'est quasiment un programme pour calculer la réponse. Un premier problème est que l'*espace de recherche*, c'est-à-dire l'espace dans lequel nous voulons trouver le plan d'exécution, est gigantesque. Pour éviter de le parcourir entièrement, nous allons utiliser des *heuristiques*, c'est-à-dire des méthodes qui, si elles ne garantissent pas de trouver le plan optimal, donnent assez rapidement des plans satisfaisants. Ces heuristiques utilisent souvent des règles de bon sens comme : *il faut réaliser les sélections le plus tôt possible*. L'autre difficulté est que pour choisir le plan le moins chronophage, l'optimiseur (c'est-à-dire le programme en charge de l'optimisation) doit être capable d'estimer le coût de chaque plan candidat et c'est une tâche complexe à laquelle le système ne peut se permettre d'accorder trop de ressources. Donc, l'optimiseur fait *de son mieux*. Et typiquement les optimiseurs de systèmes comme Oracle ou DB2 font des merveilles sur des requêtes simples. C'est bien moins glorieux pour les requêtes complexes, par exemple mettant en jeu des quantificateurs universels comme la question : quels sont les acteurs qui n'ont joué que dans des comédies ? Heureusement, en pratique, la plupart des questions posées par des applications utilisant des bases de données sont simples.

Sous-jacent dans la discussion sur l'optimisation de requête est la question de la difficulté d'obtenir une certaine information. Nous rencontrons la notion de *complexité*. Depuis Gödel, nous savons qu'il est des propositions qui ne peuvent être ni démontrées ni réfutées, qu'il est des problèmes qui ne peuvent être résolus. Cette notion d'indécidabilité commence péniblement à arriver jusqu'au grand public. Ce même public ne voit dans le fait qu'une requête prend plus ou moins longtemps que des raisons purement techniques. Evidemment, le temps de calcul dépend de la puissance du serveur, de la vitesse du disque ou de la qualité de l'optimiseur. Mais au-delà de tels aspects, il est des tâches qui demandent intrinsèquement plus de temps que d'autres. Par exemple, nous pouvons facilement afficher un graphe avec 100 noeuds ; ça ne prend que quelques fractions de secondes. Par contre, cela prendrait énormément de temps d'afficher un après l'autre *tous* les graphes possibles reliant ces 100 noeuds. Même parmi les problèmes dont la réponse est courte (par exemple, la réponse est *oui* ou *non*), il en est qui, bien que décidables, sont intrinsèquement bien plus complexes que d'autres ; il en est même que nous ne savons pas résoudre en temps raisonnable. Parfois, cette difficulté trouve même son utilité. Le système cryptographique RSA repose sur le fait que nous ne savons pas factoriser (en général) un très grand entier en nombres premiers, en un temps raisonnable et qu'il est donc très difficile de décrypter un message sans en connaître la clé secrète.

La complexité est un aspect particulièrement important pour le traitement de gros volumes de données. Pour une requête particulière, nous voulons savoir :

— quel temps il faut pour la réaliser ? complexité en temps,

— quel espace disque (ou quelle mémoire) est nécessaire ? complexité en espace.

Evidemment ces quantités dépendent de la taille de la base de données. Si la requête prend un temps t et que nous doublons la taille n de nos données, nous faut-il attendre le même temps (temps constant), le double de temps (temps linéaire en n), ou est-ce que le temps grandit de manière polynomiale (en n^k où n est la taille des données) voire exponentielle (en k^n) ? Ce n'est pas anodin : sur de gros volumes de données, une complexité en temps n^k exigera une grosse puissance de calcul, et une complexité en k^n sera rédhibitoire.

De nombreuses classes de complexité ont été étudiées. Intuitivement, une classe de complexité regroupe tous les problèmes qui peuvent être résolus sans dépasser certaines ressources disponibles, typiquement le temps ou l'espace. Par exemple, vous avez peut-être entendu parler de la classe P, temps polynomial. Il s'agit de l'ensemble des problèmes qu'il est possible de résoudre dans un temps n^k où n est la taille des données et k un entier arbitraire. Au-delà de P, nous atteignons les temps NP (pour non-déterministe polynomial [4]) et EXPTIME (temps

4. Un exemple de problème difficile dans NP est celui du voyageur de commerce. Etant données des villes, des

exponentiel), des temps prohibitifs ? Pourtant, il faut relativiser. Les systèmes informatiques résolvent routinièrement des problèmes parmi les plus complexes de NP. Et, a contrario, pour 1.5 téraoctets de données, n^3 est encore aujourd'hui hors d'atteinte, même en disposant de tous les ordinateurs de la planète.

Avant de poursuivre sur d'autres aspects du modèle relationnel, interrogeons-nous sur les origines de l'énorme succès des systèmes relationnels :

— les requêtes sont fondées sur le calcul relationnel, un langage logique, simple et compréhensible pour des humains surtout dans des variantes comme SQL ;

— une requête du calcul relationnel est facilement traduisible en une expression de l'algèbre relationnelle simple à évaluer pour des machines ;

— il est possible d'optimiser l'évaluation d'expressions de l'algèbre relationnelle car cette algèbre n'offre qu'un modèle de calcul limité ;

— enfin, pour ce langage relativement limité, le parallélisme permet de passer à l'échelle de très grandes bases de données.

Pour insister sur les deux derniers points qui sont essentiels, nous pourrions choisir pour les bases de données le slogan : *ici on ne fait que des choses simples mais on les fait vite.*

Continuons avec un aspect essentiel de la gestion de données, mais qui est hors programme : les transactions.

1.4 Les transactions‡

La modernisation des chaînes de fabrication a été principalement causée dans un premier temps par l'électronique et l'automatique. Avant de s'imposer aussi dans la production, l'informatique a elle profondément pénétré l'industrie en modifiant radicalement la manière dont des transactions, comme les commandes ou la paye, étaient gérées de manière automatique. Une transaction informatisée est la forme dématérialisée d'un contrat. Son coût peut se trouver incomparablement plus faible que celui d'une transaction réelle mettant en jeu des déplacements de personnes sur des échelles de temps bien plus longues. Avec des fonctionnalités considérablement élargies par le recours à l'informatique, les transactions se retrouvent au cœur de nombreuses applications qui ont largement contribué à populariser les systèmes relationnels comme, par exemple, les applications bancaires.

Les systèmes relationnels répondent aux besoins des transactions en supportant la notion de transaction relationnelle. Une transaction relationnelle garantit qu'une séquence d'opérations se réalise correctement, par exemple en empêchant qu'une somme d'argent ne s'évanouisse dans la nature (avec un compte en banque débité sans qu'un autre ne soit crédité). Même l'occurrence d'une panne [5] ne doit pas pouvoir conduire à une exécution incorrecte. Il nous faut donc formaliser la notion d'exécution correcte. Evidemment, il serait impossible de le faire précisément s'il fallait tenir compte des millions de choses que font de tels systèmes. Mais l'informatique, comme les mathématiques, dispose d'un outil fantastique : l'*abstraction*. Nous pouvons considérer ce que fait un système relationnel sous l'angle des transactions relationnelles et des modifications qu'elles apportent aux données, en faisant abstraction de toutes les autres tâches qu'il réalise. Il devient alors possible de définir formellement la notion d'exécution correcte.

Nous pouvons mentionner d'autres tâches que les systèmes relationnels accomplissent à coté de l'évaluation de requêtes et de la gestion de transactions relationnelles. Ils gèrent également :

— les contraintes d'intégrité (telles que *tout responsable de projet doit être enregistré dans la base des personnels*) ;

routes entre ces villes, et les longueurs de ces routes, trouver le plus court chemin pour relier toutes les villes.

5. Les applications qui tournent sur le système relationnel contiennent des bogues. Le système lui-même contient ses propres bogues. Enfin les matériels peuvent dysfonctionner.

— les déclencheurs ou *triggers* (tels que *si quelqu'un modifie la liste des utilisateurs, envoyer un message au responsable de la sécurité*) ;

— les droits des utilisateurs (pour contrôler qui a le droit de lire ou de modifier quoi) ;

— les vues (pour s'adapter aux besoins d'utilisateurs particuliers) ;

— l'archivage (pour pouvoir retrouver des données périmées depuis des lustres) ;

— le nettoyage des données (pour éliminer les doublons, les incohérences).

1

1.5 Conception d'une Base de Données‡

Nous ne discuterons pas non plus dans ce document d'un autre aspect très important qui est la *conception* d'une base de données, d'un point de vue abstrait. En effet, si le programme officiel utilise le terme de "conception" d'un schéma relationnel, le travail de conception de la structure de la base passe en général par un modèle beaucoup plus abstrait, nommé *Modèle Entité-Association (E-A)*, qui a été introduit par Chen dans [Che76] en 1976. Ce modèle est un modèle général de conception, qui peut être traduit ensuite dans le modèle relationel ou dans un autre modèle (par exemple objet). Lui sont associées des méthodologies de conception, c'est-à-dire des "règles" qui permettent en principe d'obtenir, à partir d'une situation de la vie réelle qu'on souhaite modéliser, un modèle correct. L'intérêt du modèle E-A est qu'il est plus intuitif que le modèle relationel, ne nécessite pas de compétences "mathématiques" et se prête bien à une présentation graphique. Il est donc utilisé pour échanger avec des personnes du métier qu'on souhaite "informatiser".

1.6 Notions du programme officiel

Le programme officiel de CPGE est défini de la manière suivante, et "pèse" pour 15% de la première année, soit environ 6 semaines, découpé en 6 séances d'1h de cours (préférablement 3 séances de 2h) et 3 séances de 2h de TP. Nous indiquons dans le tableau suivant les sections qui décrivent chaque point du programme. Le programme encourage une approche applicative au thème des bases de données, ce qui est naturel, puisque les bases de données sont partout dans la vie de tous les jours. Néanmoins, il nous parait extrêmement réducteur de faire croire que les bases de données ne sont **qu'un domaine d'application**. Outre des exemples applicatifs, nous présentons ainsi une approche plus formelle et mathématique, qui nous parait parfaitement adaptée au niveau des élèves des classes préparatoires. Il serait en effet regrettable de faire croire que l'apprentissage *fondamental* des bases de données se limite à l'utilisation de ressources présentées par les sites grand public, même si ces sites sont de bonnes références techniques pour un utilisateur débutant, et permettent une mise en oeuvre rapide de la pratique des bases de données sans passer par l'apprentissage et la compréhension de tous les concepts.

Le programme propose également les exercices suivants :

— utiliser une application de création et de manipulation de données, offrant une interface graphique, notamment pour créer une base de données simple, ne comportant pas plus de trois tables ayant chacune un nombre limité de colonnes. L'installation et l'exploitation d'un serveur SQL ne fait pas partie des attendus.

— lancer des requêtes sur une base de données de taille plus importante, comportant plusieurs tables, que les étudiants n'auront pas eu à construire, à l'aide d'une application offrant une interface graphique

— enchaîner une requête sur une base de données et un traitement des réponses enregistrées dans un fichier.

Nous avons déjà dans ce chapitre introductif, brossé un tableau des possibilités et des problèmes liés aux bases de données, en particulier de problèmes qui ne sont pas au programme.

Contenus	Précisions et commentaires
Vocabulaire des bases de données : relation(2.2), attribut(2.2), domaine(2.2), schéma de relation(2.2) ; notion de clé primaire(4.1).	Ces concepts sont présentés dans une perspective applicative, à partir d'exemples.
Opérateurs usuels sur les ensembles dans un contexte de bases de données : union, intersection, différence. Opérateurs spécifiques de l'algèbre relationnelle : projection(3.2), sélection (ou restriction)(3.2), renommage(3.2), jointure(3.2), produit(3.2) et division cartésiennes (3.3) ; fonctions d'agrégation : min, max, somme, moyenne, comptage (4.4).	Ces concepts sont présentés dans une perspective applicative. Les seules jointures présentées seront les jointures symétriques, simples (utilisant JOIN ... ON ... =...).
Concept de client-serveur(1.1). Brève extension au cas de l'architecture trois-tiers(1.1).	On se limite à présenter ce concept dans la perspective applicative d'utilisation de bases de données.

Dans la suite de ce document, nous nous consacrons aux aspects spécifiques du programme, débutant dans le Chapitre 2 toutefois par un aspect qui n'est pas explicitement au programme, mais qui n'est pas non plus explicitement hors programme, car il est fondamental : il s'agit du *calcul relationnel*, fondement théorique des bases de données, sans quoi cette partie du cours se résumérait à une *utilisation* des bases de données comme un simple outil. Puis, dans le Chapitre 3 nous discutons de l'algèbre relationnelle, et donnons quelques exemples d'optimisation, problématique fondamentale des bases de données. Dans le Chapitre 4 nous précisons la syntaxe SQL, et discutons du problème de l'aggrégation de données. Enfin, dans le Chapitre 5 nous proposons du contenu pour le déroulement de trois séances de TP, deux sur machine et une sur papier. Le découpage en trois chapitres de cours, respecte cette division que nous préconisons.

1

2 Le Calcul Relationnel

2.1 Objectif du chapitre

Dans ce chapitre, nous montrons par l'exemple comment écrire des requêtes en utilisant le formalisme logique du calcul relationnel. L'intérêt de cette écriture est qu'elle est *déclarative* et non pas *impérative*, c'est-à-dire qu'elle exprime *ce qu'on souhaite* obtenir et non pas *comment* l'obtenir. Ce paradigme est déjà utilisé dans des langages fonctionnels comme Caml qui permettent de s'abstraire de détails de calculs. Mais le calcul relationnel va bien plus loin en laissant au système le choix de l'algorithme d'"évaluation.

Avant toute chose, nous allons définir formellement des notations, et une terminogie. Cela pourra paraître un peu lourd ; on s'appuiera sur les exemples pour bien comprendre qu'il n'y a rien de vraiment complexe. Puis nous passerons au langage de requêtes proprement dit.

2.2 Concepts des Bases de Données

Nous utiliserons dans la suite de ce chapitre le schéma de base de donnée **CINEMA** de la Figure 2.1. Nous illustrerons la longue liste de définitions qui suit avec cet exemple.

2.2.1 Définitions et Notations

Définition 2.2.1. (Attribut) On dispose d'un ensemble (infini) **att** d'attributs. On associe à un attribut a, un ensemble de *constantes*, potentiellement infini noté [1] $\mathbf{dom}(a)$. On note **dom** l'union des ensembles de toutes les constantes de tous les attributs. On utilisera aussi un ensemble **var** de variables.

> **Exemple 2.2.2:** Dans la base de données **CINEMA**, `Titre` ou `Salle` sont des attributs. Les entiers, les réels ou les jours de la semaine, sont des domaines.

Définition 2.2.3. (Schéma relationnel) Un schéma relationnel R consiste en un ensemble fini de n attributs $U = \{u_1, \ldots, u_n\}$. On le note aussi $R[u_1, \ldots, u_n]$ ou tout simplement $R[U]$. On définit une fonction **sort**, qui associe à chaque schéma relationnel son ensemble d'attributs, c'est-à-dire $\mathbf{sort}(R) = U$.

> **Exemple 2.2.4:** FILM est un schéma relationnel, avec :
>
> $$\mathbf{sort}(\text{FILM}) = \{\texttt{Titre}; \texttt{Directeur}; \texttt{Acteur}\}.$$

1. Parfois, également appelé *type* de a

On omet souvent les virgules et les parenthèses pour les ensembles d'attributs ; et par exemple, on écrit :

$$\mathbf{sort}(\text{FILM}) = \texttt{Titre Directeur Acteur}.$$

Définition 2.2.5. (Arité) L'*arité* d'un schéma relationnel est son nombre d'attributs. On note donc $\mathbf{arite}(R) = \#\mathbf{sort}(R)$, où # représente le cardinal d'un ensemble.

Exemple 2.2.6: L'arité de FILM est 3.

Définition 2.2.7. (Schéma de base de données) Un *schéma de base de données* $\mathbf{R}$ est un ensemble fini de p schémas relationnels que l'on écrit $\mathbf{R} = \{R_1[U_1], \ldots, R_p[U_p]\}$ pour indiquer les schémas relationnels qui la composent ainsi que leurs sorts.

Exemple 2.2.8: CINEMA = { FILM [**Titre Directeur Acteur**] ; COORDONNÉES [**SalleAdresseTéléphone**] ; SÉANCE [**SalleTitreHoraire**] } est un schéma de base de données.

Définition 2.2.9. (n-uplet) Un *n-uplet* (ou *nuplet*, ou encore *tuple* en anglais) est une fonction sur un ensemble fini U d'attributs. Plus précisément, un n-uplet u sur U est une fonction de U dans **dom**. Dans ce cas, le *sort* de u est U, et son arité est $|U|$.

On écrit les n-uplets linéairement, par exemple, $\langle A : 5,\ B : 3\rangle$. L'ordre dans cette syntaxe est un ordre implicite $\leq_{\mathbf{att}}$ sur **att**. Soit u un n-uplet sur U et $A \in U$; comme il est standard en mathématiques, $u(A)$ dénote la valeur de u pour l'attribut A. On étend cette notation aux sous-ensembles de U. Soit $V \subseteq U$, $u[V]$ dénote le n-uplet v sur V tel que $v(A) = u(A)$ pour chaque $A \in V$ (c'est-à-dire $u[V] = u|_V$, la restriction de la fonction u à V). Le n-uplet sur un ensemble vide d'attributs est dénoté $\langle\ \rangle$.

Définition 2.2.10. (Relation) Une *relation*, appelée également *instance sur un schéma relationnel* $R[U]$ (ou sur un ensemble U d'attributs), est un ensemble fini I de n-uplets sur U. On dira que le sort de I est U et que son arité est $|U|$.

Chaque table dans la Figure 2.1 est la représentation graphique d'une relation.

Définition 2.2.11. (Base de Données) Une *base de données* $\mathcal{I}$, appelée également *instance d'un schéma de base de données* $\mathbf{R}$ est une fonction dont l'ensemble de définition est $\mathbf{R}$ et telle que pour chaque $R \in \mathbf{R}$, $\mathcal{I}(R)$ est une relation sur R.

Pour résumer, la Figure 2.1 représente une instance de la base de données **CINEMA**. Le schéma de base de données correspondant est donné par :

$$\mathbf{CINEMA} \quad = \quad \{\text{FILM}, \text{COORDONNÉES}, \text{SÉANCE}\}$$

où les les schémas relationnels FILM et COORDONNÉES, et SÉANCE sont tous d'arité 3 et ont les **sorts** suivants :

$$\begin{array}{lcl} \mathbf{sort}(\text{FILM}) & = & \{\texttt{Titre}, \texttt{Directeur}, \texttt{Acteur}\} \\ \mathbf{sort}(\text{COORDONNÉES}) & = & \{\ \texttt{Salle}, \texttt{Adresse}, \texttt{Téléphone}\} \\ \mathbf{sort}(\text{SÉANCE}) & = & \{\texttt{Salle}, \texttt{Titre}, \texttt{Horaire}\} \end{array}$$

Pour simplifier, on supposera par la suite que tous les attributs auront comme domaine l'ensemble des chaînes de caractères [2].

2. On pourrait définir par exemple l'horaire comme une expression régulière définissant un sous-ensemble des chaînes de caractères, comme défini par la norme ISO 8601.

FILM

Titre	Directeur	Acteur
Mais qui a tué Harry ?	*Hitchcock*	*Gwenn*
Mais qui a tué Harry ?	*Hitchcock*	*Forsythe*
Mais qui a tué Harry ?	*Hitchcock*	*MacLaine*
Mais qui a tué Harry ?	*Hitchcock*	*Hitchcock*
	. . .	
Cris et chuchotements	*Bergman*	*Andersson*
Cris et chuchotements	*Bergman*	*Sylwan*
Cris et chuchotements	*Bergman*	*Thulin*
Cris et chuchotements	*Bergman*	*Ullman*

COORDONNÉES

Salle	Adresse	Téléphone
Gaumont Opéra	*31 bd. des Italiens*	*47 42 60 33*
Saint André des Arts	*30 rue Saint André des Arts*	*43 26 48 18*
Le Champo	*51 rue des Ecoles*	*43 54 51 60*
	. . .	
Georges V	*144 av. des Champs-Elysées*	*45 62 41 46*
Les 7 Parnassiens	*98 bd. du Montparnasse*	*43 20 32 20*

SÉANCE

Salle	Titre	Horaire
Gaumont Opéra	*Cris et chuchotements*	*20 :30*
Saint-André des Arts	*Mais qui a tué Harry ?*	*20 :15*
Georges V	*Cris et chuchotements*	*22 :15*
	. . .	
Les 7 Parnassiens	*Cris et chuchotements*	*20 :45*

FIGURE 2.1– La base de données **CINEMA**

2.3 Calcul conjonctif

Nous allons démarrer avec des exemples de requêtes simples. Ensuite, nous les complexifions en rajoutant des concepts au fur et à mesure du chapitre. Nous débutons donc avec des opérateurs logiques en nombre restreint, mais qui permettent tout de même de répondre à certaines questions simples... et la plupart du temps les questions que se posent les utilisateurs de bases de données sont simples.

2.3.1 Exemples

Voici quelques exemples de requêtes "en français" que nous pouvons poser sur cette base. Dans la suite de ce document, nous donnerons des expressions formelles sous forme du calcul relationnel permettant de capturer la sémantique de ces requêtes, et des expressions algébriques permettant de calculer les résultats de ces requêtes.

(2.2.1) Qui est le metteur en scène de "Cris et chuchotements" ?

(2.2.2) Quelles salles affichent "Chiens de paille" ?

(2.2.3) Quels sont l'adresse et le numéro de téléphone du cinéma "Le Studio" ?

Chacune de ces requêtes ne met en jeu qu'une relation unique. Les requêtes suivantes impliquent plusieurs relations :

(2.2.4) Donner les noms et adresses des salles affichant un film de Bergman.

(2.2.5) Donner les paires de personnes telles que la première a mis en scène la seconde, et vice versa ;

(2.2.6) Peut-on voir un film de Bergman à la salle "Gaumont Opéra" ?

Considérons plus en détail la requête (2.2.4). Intuitivement, on veut dire :

si les n-uplets r_1, r_2, r_3 respectivement dans les relations
FILM, SÉANCE, COORDONNÉES sont **tels que**
le **`Directeur`** dans r_1 est "*Bergman*"
et les **`Titre`** dans r_1 et r_2 sont les mêmes
et les **`Salle`** dans r_2 et r_3 sont les mêmes
alors nous voulons les **`Salle`** et **`Adresse`** du n-uplet r_3.

2.3.2 Formules bien-formées du calcul conjonctif

Le calcul *conjonctif* est défini formellement de la manière suivante et correspond informellement à la logique du premier ordre avec *uniquement* des conjonctions et le quantificateur existentiel.

Un *terme* t est une constante ou une variable, on note $t \in \mathbf{var} \cup \mathbf{dom}$. Pour un schéma de base de donnée $\mathbf{R}$ et $R \in \mathbf{R}$, un *atome* sur R est une expression $R(t_1, \ldots, t_n)$ où $n = \mathbf{arite}(R)$ et chaque t_i est un terme.

Par exemple, la constante "*Chiens de paille*" et les variables x_d, x_a sont des termes et FILM("*Chiens de paille*", x_d, x_a) est un atome sur FILM.

Les *formules de base* incluent les atomes sur $\mathbf{R}$ et les expressions $e = e'$ pour des termes e, e'. Les formules *bien-formées* sont de la forme :

(a) φ, où φ est une *formule de base* sur $\mathbf{R}$; ou

(b) $(\varphi \wedge \psi)$ où φ et ψ sont des formules *bien-formées* sur $\mathbf{R}$; ou

(c) $\exists x\ \varphi$, où x est une variable et φ une formule *bien formée* sur $\mathbf{R}$.

Exemple 2.3.1: $\varphi_0 = \text{FILM}(x_t, \text{“}Bergman\text{”}, x_a) \wedge \text{SÉANCE}(x_s, x_t, x_h)$ est une formule bien-formée, puisque $\text{FILM}(x_t, \text{“}Bergman\text{”}, x_a)$ et $\text{SÉANCE}(x_s, x_t, x_h)$ sont deux *atomes*.

Les notions d'*occurrence* de variables *libres* et *liées* dans une formule sont définies de la manière usuelle. Une occurrence d'une variable x est *libre* dans une formule φ si :

(i) φ est un atome ; ou

(ii) $\varphi = (\psi \wedge \xi)$ et l'occurrence de x dans ψ ou ξ est libre ; ou

(iii) $\varphi = \exists y\ \psi$, x et y sont des variables distinctes, et l'occurrence de x est libre dans ψ.

L'ensemble $libre(\varphi)$ est l'ensemble des variables qui ont au moins une occurrence libre dans φ. Une variable est liée si elle n'est pas libre.

Exemple 2.3.2: $libre(\varphi_0) = \{x_t, x_a, x_s, x_h\}$.

Définition 2.3.3. Requête Une *requête* est définie comme une expression de la forme

$$\{t_1, \ldots, t_m \mid \varphi\}$$

où $t_1, \ldots, t_m$ sont des termes et où l'ensemble des variables apparaissant dans $t_1, \ldots, t_m$ est exactement $libre(\varphi)$.

Exemple 2.3.4: $\{\text{“}Bergman\text{”}, x_t, x_a, x_s, x_h \mid \text{FILM}(x_t, \text{“}Bergman\text{”}, x_a) \wedge \text{SÉANCE}(x_s, x_t, x_h)\ \}$ est une requête.

Une telle requête va définir une fonction. Appliquée à une instance de schéma de base de données, elle va retourner une relation d'arité m, c'est-à-dire un ensemble de n-uplets d'arité m, les *résultats*. On pourra donner à la relation résultat :

$$res = \{t_1, \ldots, t_m \mid \varphi\}$$

On dira qu'on a défini une *vue* de la base de données. La différence entre une relation de la base de données et une vue de cette même base, est que la relation est stockée sur disque et qu'elle peut être modifiée, alors que la vue est calculée à partir des relations stockées et qu'elle n'est pas directement modifiable.

2.3.3 Exercices corrigé

Nous donnons ici les requêtes permettant d'exprimer les exemples de la Section 2.3.1.

Exemple 2.3.5: La requête (2.2.1) *« Qui est le Directeur de “Chiens de paille” ? »* s'écrit :

$$res = \{x_d \mid \exists x_a, \text{FILM}(\text{“}Chiens\ de\ paille\text{”}, x_d, x_a)\}\ .$$

On notera que la réponse est un *ensemble* de nuplets. Donc on pourra avoir zéro, une ou plusieurs réponses.

Exemple 2.3.6: La requête (2.2.2) *« Quelles salles projettent “Chiens de paille” ? »* s'écrit :

$$res = \{x_s \mid \exists x_h, \text{SÉANCE}(x_s, \text{“}Chiens\ de\ paille\text{”}, x_h)\}\ .$$

Exemple 2.3.7: La requête (2.2.3) *« Adresse et numéro de téléphone du “Studio” ? »* s'écrit :

$$res = \{x_a, x_t \mid \text{COORDONNÉES}(\text{“}Studio\text{”}, x_a, x_t)\}\ .$$

Exemple 2.3.8: La requête (2.2.4) *« Nom et Adresse des salles projetant un film de Bergman ? »* s'écrit :

$$res = \{x_s, x_{ad} \mid \begin{array}{l} \exists x_{tel}, x_h, x_{act}, x_{titre}, \text{FILM}(x_{titre}, \text{“}Bergman\text{”}, x_{act}) \\ \wedge \text{SÉANCE}(x_s, x_{titre}, x_h) \\ \wedge \text{COORDONNÉES}(x_s, x_{ad}, x_{tel})\} \end{array} .$$

Observons que la variable x_{titre} apparait dans l'atome sur la relation FILM et celui sur SÉANCE. On parle bien d'un film de Bergman, et on cherche une séance de ce film et pas d'un autre.

Notons que cette même requête peut également s'écrire en regroupant les variables quantifiées ensemble.

$$res = \{x_s, x_{ad} \mid \begin{array}{l} \exists x_h, x_{act}, x_{titre}, \text{FILM}(x_{titre}, \text{“}Bergman\text{”}, x_{act}) \\ \wedge \text{SÉANCE}(x_s, x_{titre}, x_h) \\ \wedge \exists x_{tel}, \text{COORDONNÉES}(x_s, x_{ad}, x_{tel})\} \end{array} .$$

Exemple 2.3.9: La requête (2.2.5) *« Paires directeur/acteur ? »* s'écrit :

$$res = \{x_d, x_a \mid \exists x_t, \text{FILM}(x_t, x_d, x_a)\}$$

Exemple 2.3.10: La requête (2.2.5) *« Peut-on voir un film de Bergman à la salle “Gaumont Opéra” ? »* s'écrit :

$$res = \{\langle\,\rangle \mid \begin{array}{l} \exists x_t, x_s, x_h \ \text{FILM}(x_t, \text{“}Bergman\text{”}, x_a) \\ \wedge \text{FILM}(\text{“}Gaumont\ Opéra\text{”}, x_t, x_h)\} \end{array}$$

On notera que l'ensemble résultat contiendra le n-uplet d'arité 0 $\langle\,\rangle$ si la formule logique est vraie, et sera vide sinon.

2.3.4 Conclusion

Le calcul conjonctif est simple et possède une équivalence sémantique très intéressante avec l'algèbre “SPJR”, présentée dans la Section 3.2. Toutefois, il ne permet pas de poser certaines questions :

(2.3.1) « Où puis-je voir le film “Annie Hall” ou “Manhattan” ? »– il faudrait la *disjonction*, c'est-à-dire le OU ($\vee$), ce qui donne le *calcul positif* !

(2.3.2) « Quels sont les acteurs qui ont joué dans tous les films de Hitchcock ? »– il faudrait le quantificateur universel.

(2.3.3) « Quels sont les films que Hitchcock a dirigés, mais dans lesquels il n'a pas joué en tant qu'acteur ? »– il faudrait la *négation* ! Dans la section suivante, nous présentons le calcul relationnel, qui permet d'exprimer ces types de questions.

Ce qui est intéressant, c'est qu'en rajoutant simplement la négation, on obtient à la fois la disjonction et le quantificateur universel, puisque $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$, et $\forall x\varphi \equiv \neg(\exists x(\neg\varphi))$.

2.4 Calcul relationnel

En ajoutant la négation au calcul conjonctif, on obtient le calcul relationnel, qui est essentiellement le calcul des prédicats du premier ordre (sans symbole de fonction). On a un problème : celui de garantir des résultats finis.

On donnera ici une interprétation particulière simplificatrice : l'interprétation dans le *domaine actif* ; et nous envisagerons ensuite des interprétations plus riches.

2.4.1 Formules bien-formées du calcul relationnel

Un *terme* est une constante ou une variable, c'est-à-dire un élément de $\mathbf{var} \cup \mathbf{dom}$. Pour un schéma de base de donnée $\mathbf{R}$ et $R \in \mathbf{R}$, un *atome* sur R est une expression $R(t_1, \ldots, t_n)$ où $n = \textit{arité}(R)$ et chaque t_i est un terme.

Les *formules de base* incluent les atomes sur $\mathbf{R}$ et les expressions $e = e'$ pour des termes e, e'. Les formules (*bien-formées*) sont de la forme :

(a) $\neg\varphi$, où φ est une *formule de base* sur $\mathbf{R}$;

(b) $\neg\varphi$, où φ est une formule *bien formée* sur $\mathbf{R}$;

(c) $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ où φ et ψ sont des formules *bien formées* sur $\mathbf{R}$; et

(d) $\exists x\ \varphi$, $\forall x\ \varphi$, où x est une variable et φ une formule*bien formée* sur $\mathbf{R}$.

Comme il est usuel,

$$\forall x_1, x_2, \ldots, x_m\ \varphi \text{ est une abréviation de } \forall x_1 \forall x_2 \ldots \forall x_m\ \varphi,$$

et on inclue souvent deux connecteurs logiques supplémentaires *implique* ($\rightarrow$) et *est équivalent à* ($\leftrightarrow$) que l'on voit de la manière suivante

$$\begin{array}{rcl} \varphi \rightarrow \psi & \equiv & \neg\varphi \vee \psi \\ \varphi \leftrightarrow \psi & \equiv & (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi) \end{array}$$

Nous étendons les notions d'occurrence de variables libres et liées déjà introduites. Une occurrence d'une variable x est *libre* dans une formule φ si :

(i) φ est un atome ; ou

(ii) $\varphi = (\psi \wedge \xi)$ et l'occurrence de x dans ψ ou ξ est libre (et idem pour $\vee$) ; ou

(iii) $\varphi = \exists y\ \psi$, x et y sont des variables distinctes variables, et l'occurrence de x est libre dans ψ (et idem pour $\forall$).

L'ensemble $\textit{libre}(\varphi)$ est l'ensemble des variables qui ont au moins une occurrence libre dans φ. Une variable est liée si elle n'est pas libre.

Une requête est définie comme une expression de la forme

$$\{t_1, \ldots, t_m \mid \varphi\}$$

où $t_1, \ldots, t_m$ sont des termes et où l'ensemble des variables apparaissant dans $t_1, \ldots, t_m$ est exactement $\textit{libre}(\varphi)$.

2.4.2 Exercices corrigés

Exemple 2.4.1: La requête (2.3.1) *« Salles où on peut voir "Annie Hall" ou "Manhattan" »* est exprimée par :

$$\textit{res} = \{x_t \mid \exists x_{s1}, x_{s2}(\textsc{Séance}(x_t, \text{“}\textit{Annie Hall}\text{”}, x_{s1}) \vee \textsc{Séance}(x_t, \text{“}\textit{Manhattan}\text{”}, x_{s2}))\}.$$

Notons que si la question implique qu'on voulait les voir *exactement au même horaire de début*, alors il faudrait écrire :

$$\textit{res} = \{x_t \mid \exists x_s(\textsc{Séance}(x_t, \text{“}\textit{Annie Hall}\text{”}, x_s) \vee \textsc{Séance}(x_t, \text{“}\textit{Manhattan}\text{”}, x_s))\}$$

Exemple 2.4.2: La requête (2.3.2) *« Acteurs qui ont joué dans tous les films de Hitchcock »* est exprimée par :

$$\textit{res} = \{x_a \mid \forall x_f, \textsc{Film}(x_f, \text{“}\textit{Hitchcock}\text{”}, x_a)\}.$$

Exemple 2.4.3: La requête (2.3.3) *« Films dirigés par Hitchcock dans lesquels il n'a pas joué »* est exprimée par :

$$res = \{x_t \mid \exists x_a \textsc{Film}(x_t, \text{“}\textit{Hitchcock}\text{”}, x_a) \ \wedge \ \neg \textsc{Film}(x_t, \text{“}\textit{Hitchcock}\text{”}, \text{“}\textit{Hitchcock}\text{”})\} \ .$$

2.4.3 Pour aller plus loin‡

Cette section est hors-programme. Elle est utile pour le lecteur qui souhaite comprendre plus profondément le Théorème d'équivalence, énoncé à la Section 3.4.1

Requêtes non sûres

Avant de présenter une sémantique précise, on présente intuitivement les problèmes dans les expressions suivantes :

$$\begin{array}{lll} \textit{(non-sûre-1)} & \{x & \mid \ \neg \textsc{Film}(\text{“}\textit{Cris et chuchotements}\text{”}, \text{“}\textit{Bergman}\text{”}, x)\} \\ \textit{(non-sûre-2)} & \{x, y & \mid \ \textsc{Film}(\text{“}\textit{Cris et chuchotements}\text{”}, \text{“}\textit{Bergman}\text{”}, x) \\ & & \quad \vee \textsc{Film}(y, \text{“}\textit{Bergman}\text{”}, \text{“}\textit{Ullman}\text{”})\} \end{array}$$

Dans le calcul des prédicats, la requête *non-sûre-1* produit tous les nuplets $\langle a \rangle$ tels que $a \in \mathbf{dom}$ et $\langle$“*Cris et chuchotements*”, “*Bergman*”, $a\rangle$ n'est pas dans la relation Film. Comme la relation Film est par définition finie et que **dom** est infini, la requête donne un résultat infini. Comme l'input est par définition fini, le requête donne un résultat infini. La même remarque est vraie pour la requête *non-sûre-2*, même si elle n'utilise pas explicitement la négation.

Une solution naturelle (encore que pas totalement satisfaisante) est de considérer que les variables prennent leurs valeurs dans un contexte restreint, par exemple, une variable de type *acteur* ne prend comme valeurs que les acteurs connus de la base. Même si les domaines sous-jacents sont finis, les utilisateurs ne sont pas censés les connaître et il est gênant que le résultat des requêtes dépende de cette information.

Un autre problème assez subtil vient de la quantification. Considérons la requête

$$\textit{(non-sûre-3)} \qquad \{x \mid \forall y \ R(x, y)\}$$

Si R est fini, alors la réponse à cette requête est finie, mais elle dépend du domaine des y auxquels on applique le quantificateur. En effet, si y prend sa valeur dans la totalité de **dom(y)** et que **dom(y)** est infini, alors aucun x ne pourra vérifier cette relation, puisqu'il ne peut être au mieux en relation qu'avec un nombre fini de y. Ainsi, le résultat serait toujours vide. Si on considère une interprétation sur un domaine fini, alors le résultat pourrait être non vide, puisqu'il n'y a alors plus d'incompatibilité entre la possibilité qu'il existe un x en relation avec un ensemble fini de valeurs de y.

C'est la notion de domaine actif (le domaine effectivement utilisé) qui va nous permettre de contourner ces difficultés.

Interprétation dans le domaine actif

Soit une requête $q = \{libre(\varphi), constantes \mid \varphi\}$, et une base de données $\mathcal{I}$ (c'est à dire une instance d'un schéma $\mathbf{R}$). On note $adom(q, \mathcal{I})$ l'ensemble des constantes apparaissant soit dans q soit dans $\mathcal{I}$, et on le considère comme domaine pour les variables de la requête (en d'autres termes, on n'autorise comme valeur possible des variables uniquement des valeurs présentes dans la base de données ou dans la requête). On considère donc des valuations ν de $libre(\varphi)$ dans $adom(q, \mathcal{I})$. Alors, $\mathcal{I}$ *satisfait* φ pour ν (dans cette interprétation dans le domaine actif), dénoté $\mathcal{I} \models_{adom} \varphi[\nu]$, si

(a) $\varphi = R(u)$ est un atome et $\nu(u) \in \mathcal{I}(R)$;

(b) $\varphi = (s = s')$ est un atome d'égalité et $\nu(s) = \nu(s')$;

(c) $\varphi = (\psi \wedge \xi)$ et $\mathcal{I} \models_{adom} \psi[\nu]$ et $\mathcal{I} \models_{adom} \xi[\nu]$;

(d) $\varphi = (\psi \vee \xi)$ et $\mathcal{I} \models_{adom} \psi[\nu]$ ou $\mathcal{I} \models_{adom} \xi[\nu]$;

(e) $\varphi = \neg\psi$ et $\mathcal{I} \not\models_{adom} \psi[\nu]$, c'est-à-dire, $\mathcal{I} \models_{adom} \psi[\nu]$ n'est pas vrai ;

(f) $\varphi = \exists x\ \psi$ et pour quelque $c \in adom$, $\mathcal{I} \models_{adom} \psi[\nu \cup \{x/c\}]$; ou

(g) $\varphi = \forall x\ \psi$ et pour chaque $c \in adom$, $\mathcal{I} \models_{adom} \psi[\nu \cup \{x/c\}]$.

où $\nu \cup \{x/c\}$ représente l'extension de u obtenue en ajoutant l'élément x à son domaine et en lui assignant la valeur c.

Soit $\mathbf{R}$ un schéma, et $q = \{t_1, \ldots, t_n \mid \varphi\}$ une requête du calcul sur $\mathcal{I}$. Alors, l'*image* de $\mathcal{I}$ par q (pour cette sémantique) est donnée par :

$$q_{adom}(\mathcal{I}) = \{\nu(\langle t_1, \ldots, t_n \rangle) \mid \mathcal{I} \models_{adom} \varphi[\nu], \\ \nu \text{ est une valuation sur } libre(\varphi) \\ \text{dont l'image est incluse dans } adom\}.$$

Autres sémantiques

On peut définir la sémantique d'une requête en prenant **dom** à la place du domaine actif. Dans ce cas, la réponse à une requête peut être considérée comme indéfinie si le résultat est infini. Les requêtes intéressantes d'un point de vue pratique sont celles dont la réponse ne dépend pas du choix du domaine (en supposant qu'il contienne au moins le domaine actif). Malheureusement, cette propriété est indécidable, c'est-à-dire qu'il n'existe pas d'algorithme qui décide si une requête dépend ou non du domaine. On a donc développé une gamme de critères suffisants (qui garantissent que les requêtes sont bien indépendantes du domaine). Les langages utilisés en pratique répondent à ces critères, notamment en forçant l'utilisateur à préciser le domaine des variables qu'il utilise. Nous verrons cela avec SQL.

2

3 L'Algèbre Relationnelle

3.1 Objectif du chapitre

Dans ce chapitre, nous montrons comment écrire des requêtes en utilisant l'algèbre relationnelle. L'intérêt de cette écriture est qu'elle conduit directement à des algorithmes pour évaluer les requêtes, calculer leurs réponses. Nous donnons (sans preuve formelle) le résultat sans doute le plus fondamental de ce domaine : le théorème d'équivalence de Codd. Ce théorème établit l'équivalence entre le calcul et l'algèbre relationnels. En fait, l'intérêt réside dans la traduction des requêtes (provenant de l'utilisateur) en requêtes directement évaluables (et optimisables) par le système. Si on voit la base de donnée comme un outil tampon entre l'utilisateur (humain) et une machine, ce résultat est la pierre angulaire de cette interface.

3.2 Algèbre conjonctive

Par algèbre, nous entendons ici simplement un ensemble avec des opérations fermées sur cet ensemble. L'algèbre et ses opérations vont apporter une perspective très différente sur les requêtes des bases de données relationnelles. Nous considérons des opérations unaires et binaires sur des relations. L'application de ces opérations exige certaines contraintes de typage. (C'est une algèbre à plusieurs **sorts**). Nous verrons pour chaque opération les contraintes qu'elle impose à ses arguments.

Dans un premier temps, nous considérons quatre opérations : sélection, projection, jointure et renommage. Nous appellerons *algèbre conjonctive* l'algèbre obtenue avec ces quatre opérations. Un résultat essentiel est l'équivalence entre le calcul conjonctif et l'algèbre conjonctive.

Commençons par un exemple qui nous permettra d'illustrer deux opérations unaires et une opération binaire :

— la sélection notée σ qui permet d'éliminer des nuplets (des lignes de relation)

— la projection notée π qui permet d'éliminer des attributs (des colonnes de relation).

— la jointure notée $\bowtie$ qui permet de combiner les informations de plusieurs relations.

Ces opérations seront définies formellement plus loin.

Exemple 3.2.1: La requête (2.2.4), *« Quels sont les nom et adresse des salles affichant un film de Bergman »* peut être construite en utilisant des opérations algébriques. D'abord, on utilise la sélection pour extraire les n-uplets de FILM avec *Bergman* comme Directeur :

$$I_1 := \sigma_{\texttt{Directeur}=\text{“}Bergman\text{”}}(\text{FILM}).$$

Ensuite, on obtient la liste des titres de films par projection :

$$I_2 := \pi_{\texttt{Titre}}(I_1).$$

Observez que ces deux opérations permettent de "filtrer" l'information en réduisant le nombre de lignes et de colonnes d'une relation.

La jointure nous sert à relier des données contenues dans plusieurs tables. On a par I_2 les titres de films de Bergman. On y rajoute les salles et adresses par jointures : I_2 et SÉANCE on en commun le **Titre** et SÉANCE et COORDONNÉES on en commun l'attribut **Salle** :

$$I_3 := I_2 \bowtie \text{SÉANCE} \bowtie \text{COORDONNÉES}.$$

Le résultat est obtenu en se débarrassant des colonnes inutiles :

$$I_4 := \pi_{\texttt{Salle},\texttt{Adresse}}(I_3).$$

En d'autres termes, la requête (2.2.4) peut être obtenue par :

$$\pi_{\texttt{Salle},\texttt{Adresse}}(\pi_{\texttt{Titre}}(\sigma_{\texttt{Directeur}=\text{“}Bergman\text{”}}(\text{FILM})) \bowtie \text{SÉANCE} \bowtie \text{COORDONNÉES}),$$

ou simplement par :

$$\pi_{\texttt{Salle},\texttt{Adresse}}(\sigma_{\texttt{Directeur}=\text{“}Bergman\text{”}}(\text{FILM} \bowtie \text{SÉANCE} \bowtie \text{COORDONNÉES}))$$

Comme le montre l'exemple précédent, la jointure naturelle peut être utilisée pour construire des "ponts" entre données venant de plusieurs relations en utilisant les noms d'attributs communs. Cependant, cela n'est pas possible si on veut "relier" deux relations suivant des noms d'attributs distincts. Un autre exemple va nous permettre de montrer comment réaliser cela en utilisant une quatrième opération, le renommage. Nous poursuivons donc par un exemple, et définissons ensuite ces quatre opérations formellement.

3

Exemple 3.2.2: Supposons que la base de données **CINEMA** est étendue par une relation AIME de schéma {**Personne**, **Directeur**}, où AIME(x, y) indique que **Personne** x aime **Directeur** y. Pour répondre à la requête "Quels directeurs sont aimés d'au moins un de leurs pairs ?" on aimerait faire la jointure entre les relations FILM et AIME en associant la colonne **Directeur** de la première à la colonne **Personne** de la seconde. Pour ce faire, on commence par "renommer" (opérateur ρ) l'attribut **Directeur** en **Personne** dans FILM. On peut ensuite faire la jointure du résultat avec AIME et répondre à la requête par :

$$\pi_{\texttt{Directeur}}(\rho_{\texttt{Directeur}\to\texttt{Personne}}(\text{FILM}) \bowtie \text{AIME})$$

Notons que l'attribut **Directeur** sur lequel on projette au final est l'attribut de la relation AIME, l'attribut de la relation FILM ayant été renommé !

Les opérations sont définies formellement de la manière suivante.

Sélection : La sélection a la forme $\sigma_{A=a}$ et $\sigma_{A=B}$, $A, B \in \textbf{att}$ et $a \in \textbf{dom}$. Pour I une relation (ayant A (respectivement A, B) parmi ses attributs),

$$\sigma_{A=c}(I) = \{t \in I \mid t(A) = c\} \quad \sigma_{A=B}(I) = \{t \in I \mid t(A) = t(B)\}.$$

Au contraire, $\sigma_{C=c}(I)$ est incorrect car I n'a pas d'attribut C.

Projection : La projection a la forme $\pi_{A_1,\ldots,A_n}$, $n \geq 0$ où $A_1, \ldots, A_n \in \textbf{att}$. Pour I une relation (ayant $A_1, \ldots, A_n$ parmi ses attributs),

$$\pi_{A_1,\ldots,A_n}(I) = \{\langle A_1 : t(A_1), \ldots, A_n : t(A_n)\rangle \mid t \in I\}$$

Au contraire, $\pi_{C,C'}(I)$ est incorrect si I n'a pas d'attribut C ou C'.

Si l'on considère un n-uplet comme une fonction, on peut définir la projection à l'aide de la notion de restriction d'une fonction à un sous-ensemble de son ensemble de définition de la façon suivante :

$$\pi_{A_1,\ldots,A_n}(I) = \{t|_{A_1,\ldots,A_n} \mid t \in I\}.$$

Jointure (naturelle) : Cette opération, dénoté $\bowtie$, s'applique à deux relations quelconques I et J de sorts V et W, respectivement, et produit une relation de sort $V \cup W$:

$$I \bowtie J \;=\; \{t \text{ sur } V \cup W \mid \text{il existe } \; v \in I \text{ et } w \in J, \\ t|_V = v \text{ et } t|_W = w\}$$

Quand $\mathbf{sort}(I) = \mathbf{sort}(J)$, $I \bowtie J = I \cap J$ (c'est l'intersection ensembliste classique), et quand $\mathbf{sort}(I) \cap \mathbf{sort}(J) = \emptyset$, alors $I \bowtie J$ est le produit cartésien de I et J, dénoté $I \times J$.

Par exemple, considérons I_1, I_2, deux relations sur AB, I_3 sur BC et I_4 sur CD. Alors on a :

— $I_1 \bowtie I_2 = I_1 \cap I_2$ est une relation sur AB,

— $I_1 \bowtie I_3$ est une relation sur ABC, et

— $I_1 \bowtie I_4 = I_1 \times I_4$ est une relation sur $ABCD$.

L'opération de jointure est associative et admet comme élément neutre la relation d'arité 0 non vide $\{\langle\rangle\}$. Comme la jointure est associative, on s'autorise à la voir comme polyadique, par exemple, on écrira $I_1 \bowtie \cdots \bowtie I_n$, ou même $\bowtie \{I_i\}$ pour un ensemble $\{I_i\}$ d'instances en utilisant aussi la commutativité de la jointure.

Renommage : Un *renommage* pour un ensemble fini d'attributs U est une fonction injective de U dans **att** que l'on notera $A_1 \to B_1, \ldots, A_n \to B_n$. L'opération de renommage $\rho_{A_1 \to B_1, \ldots, A_n \to B_n}$ transforme une relation sur $A_1, \ldots, A_n$ en une relation sur $B_1, ..., B_n$ de la manière suivante :

$$\rho_{A_1 \to B_1, \ldots, A_n \to B_n}(I) \;=\; \{v \text{ sur } B_1, \ldots, B_n \mid \text{il existe } \; u \in I, v(B_i) = u(A_i) \text{ pour tout } i \in [1..n]\}$$

Nous allons définir l'algèbre conjonctive. Les éléments de cette algèbre sont appelés des *requêtes algébriques* et ses opérations sont les opérations de sélection (σ, appelé également *restriction*), de projection (π), de renommage (ρ) et de jointure naturelle ($\bowtie$).

Une requête algébrique est une application qui associe à une instance $\mathcal{I}$ d'un schéma $\mathbf{R}$ une relation d'un schéma particulier. Pour construire l'ensemble des requêtes algébriques, on définit en premier lieu un ensemble de requêtes algébriques de base :

— si R est un schéma de relation de $\mathbf{R}$, alors $[R]$ est une requête algébrique ;

— Si A est un attribut de **att** et a un élément du domaine **dom**, alors $\{\langle A : a\rangle\}$ est une requête algébrique.

Ce second type de requêtes est appelé *requête constante*. On donne à ces deux types de requêtes algébriques la sémantique suivante :

$$[R](\mathcal{I}) = \mathcal{I}(R)$$
$$\{\langle A : a\rangle\}(\mathcal{I}) = \{\langle A : a\rangle\}$$

c'est à dire que lorsque l'on applique $[R]$ à une base de données $\mathcal{I}$ de schéma $\mathbf{R}$, le résultat est la relation $\mathcal{I}(R)$, et lorsque l'on applique $\{\langle A : a\rangle\}$ à une base de données de schéma $\mathbf{R}$, le résultat est la relation à une ligne et une colonne $\{\langle A : a\rangle\}$. Ainsi, le sort de la requête $[R]$ est tout simplement $\mathbf{sort}(R)$ et celui de $\{\langle A : a\rangle\}$ est A.

Exemple 3.2.3: Sur le schéma **CINEMA**, le sort de [FILM] est **`Titre`**, **`Directeur`**, **`Acteur`** et celui de $\{\langle$**`Directeur`** : *Bergman*$\rangle\}$ est **`Directeur`**.

L'ensemble des requêtes algébriques est engendré à partir de ces requêtes de bases et des opérateurs de sélection, projection, renommage et jointure de la façon suivante : si q_1 et q_2 sont deux requêtes algébriques, $A_1, \ldots, A_n, B_1, \ldots, B_n$ des attributs et c un élément de **dom**, alors :

— $\sigma_{A_1=c}(q_1)$ et $\sigma_{A_1=A_2}(q_1)$ sont des requêtes algébriques ;

— $\pi_{A_1,\ldots,A_n}(q_1)$ est une requête algébrique ;

— $\rho_{A_1 \to B_1,\ldots,A_n \to B_n}(q_1)$ est une requête algébrique ;

— $q_1 \bowtie q_2$ est une requête algébrique.

On donne aux opérateurs et à la jointure de requêtes algébriques la sémantique suivante, où $\mathcal{I}$ est une instance d'un schéma de base de données $\mathbf{R}$ et Ψ un opérateur de sélection, projection ou renommage et q_1 et q_2 deux requêtes algébriques :

— $\Psi(q_1)(\mathcal{I}) = \Psi(q_1(\mathcal{I}))$;

— $(q_1 \bowtie q_2)(\mathcal{I}) = q_1(\mathcal{I}) \bowtie q_2(\mathcal{I})$.

Exemple 3.2.4: [FILM] $\bowtie$ {⟨**Directeur** : *Bergman*⟩} est une requête algébrique, dont la sémantique donne :

$$\begin{aligned}&[\text{FILM}] \bowtie \{\langle \texttt{Directeur} : \textit{Bergman}\rangle\}(\textbf{CINEMA})\\ &\quad = [\text{FILM}](\textbf{CINEMA}) \bowtie \{\langle \texttt{Directeur} : \textit{Bergman}\rangle\}(\textbf{CINEMA})\\ &\quad = \text{FILM} \bowtie \{\langle \texttt{Directeur} : \textit{Bergman}\rangle\}\end{aligned}$$

C'est donc la jointure naturelle entre la relation FILM et la relation {⟨**Directeur** : *Bergman*⟩}, c'est-à-dire, l'ensemble des films dont le directeur est *Bergman*. On remarque que la requête algébrique suivante donne le même résultat :

$$\sigma_{\texttt{Directeur}=\textit{Bergman}}([\text{FILM}])(\textbf{CINEMA}).$$

On peut donc écrire :

$$[\text{FILM}] \bowtie \{\langle \texttt{Directeur} : \textit{Bergman}\rangle\} = \sigma_{\texttt{Directeur}=\textit{Bergman}}([\text{FILM}])$$

Le résultat important de la section est le suivant :

Théorème 3.2.5. (Théorème d'équivalence) Une requête est exprimable en calcul conjonctif si et seulement si elle est exprimable en algèbre conjonctive.

Point Crucial : Toute requête du calcul conjonctif peut facilement être exprimée dans l'algèbre conjonctive : $\exists$ est simulé par une restriction, et $\wedge$ par la jointure. Pour la simulation de l'algèbre par les requêtes conjonctives, on remarque dans un premier temps que cela est évident pour des requêtes de la forme :

$$(*)\ \pi_U(\sigma_\gamma(\sigma_{\gamma_1}(\rho|_{f_1}(R_1)) \bowtie \cdots \bowtie \sigma_{\gamma_n}(\rho|_{f_n}(R_n))))$$

où chaque γ_i est une condition de sélection, chaque f_i un renommage, et chaque R_i une relation de la base de données. (Voir l'exemple plus loin.) Pour des requêtes algébriques plus complexes, on se ramène à la forme $(*)$ en utilisant des équivalences algébriques telles que :

$\sigma_F(\sigma_{F'}(q))$	$\leftrightarrow$	$\sigma_{F'}(\sigma_F(q))$	
$\pi_X(\pi_Y(q))$	$\leftrightarrow$	$\pi_{X\cap Y}(q)$	
$\sigma_F(\pi_X(q))$	$\leftrightarrow$	$\pi_X(\sigma_F(q))$	si F porte sur des attributs de X
$q_1 \bowtie q_2$	$\leftrightarrow$	$q_2 \bowtie q_1$	
$\sigma_F(q_1 \bowtie q_2)$	$\rightarrow$	$\sigma_F(q_1) \bowtie q_2$	si F porte sur des attributs de $\mathbf{sort}(q_1)$
$\pi_X(q_1 \bowtie q_2)$	$\rightarrow$	$\pi_X(q_1) \bowtie \pi_X(q_2)$	si $X \subseteq \mathbf{sort}(q_1) \cap \mathbf{sort}(q_2)$

□

L'exemple suivant illustre la traduction de requête algébrique en requête calcul.

Exemple 3.2.6: Considérons la requête algébrique suivante :

$$\pi_{AC}(\sigma_{A=C'}(\sigma_{C=A'}(\sigma_{B=1}(R) \bowtie \rho|_{ABC/A'B'C'}(R))))$$

où $sort(R) = ABC$. Une requête conjonctive équivalente est :

$$\{x, z \mid \exists y'(R(x, 1, z) \wedge R(z, y', x)\}$$

L'exemple suivant illustre une réécriture algébrique.

Exemple 3.2.7: Soit $sort(R) = ABC$ et $sort(S) = BCD$. Alors nous avons :

$$\begin{array}{ll} \pi_{AD}(\sigma_{C=1}(R \bowtie S)) & \leftrightarrow \\ \pi_{AD}(\sigma_{C=1}(R) \bowtie S) & \leftrightarrow \\ \pi_{AD}(\pi_{ABD}(\sigma_{C=1}(R) \bowtie S)) & \leftrightarrow \\ \pi_{AD}(\pi_{AB}(\sigma_{C=1}(R)) \bowtie \pi_{BD}(S)) & \end{array}$$

3.3 Algèbre relationnelle

Pour obtenir une algèbre avec un pouvoir d'expression équivalent à celui du calcul relationnel, il suffit de rajouter l'union ($\cup$) et la différence ($\setminus$), deux opérations qui ne peuvent être appliquées que sur deux relations de même *sort*.

Exemple 3.3.1: La requête (2.3.1) *« Salles où on peut voir "Annie Hall" ou "Manhattan" »* est exprimée par :

$$\pi_{\texttt{Salle}}(\sigma_{\texttt{Titre}=\text{“}Annie\ \text{Hall”}}\text{SÉANCE} \cup \sigma_{\texttt{Titre}=\text{“}Manhattan\text{”}}\text{SÉANCE}).$$

Exemple 3.3.2: Division Cartésienne. Pour écrire la requête (2.3.2) *« Nom des acteurs ayant joué dans tous les films de Hitchcock »*, il faut réussir à écrire le quantificateur universel en utilisant la différence. Pour faciliter la lisibilité, nous décomposons ce calcul en plusieurs étapes. Tout d'abord il faut calculer tous les films de Hitchcock (F_{Hitch}). On note également A la projection sur l'attribut **Acteur**. On construit ensuite grâce au produit cartésien un ensemble de toutes les combinaisons possibles d'acteurs et de films de Hitchcock (X). Si on calcule $Y = X \setminus$ FILM, on aura retiré tous les acteurs ayant joué dans tous les films de Hitchcock. Il nous restera à comparer la projection sur **Acteur** de Y avec l'ensemble des acteurs Y pour obtenir le résultat attendu Z !

1. $F_{Hitch} = \pi_{\texttt{Titre},\texttt{Directeur}}(\sigma_{\texttt{Directeur}=\text{“}Hitchcock\text{”}}(\text{FILM}))$
2. $A = \pi_{\texttt{Acteur}}(\text{FILM})$
3. $X = F_{Hitch} \times A$
4. $Y = X \setminus \text{FILM}$
5. $Z = A \setminus Y$

Cette opération s'écrit également de la manière suivante, en utilisant l'opérateur de division cartésienne, noté $\div$: $\pi_{\texttt{Acteur}}(\text{FILM} \div \pi_{\texttt{Titre}}\sigma_{\texttt{Directeur}=\text{“}Hitchcock\text{”}}(\text{FILM}))$.

Formellement, la division d'une relation R_1 par une relation R_2 est une relation $Q = R_1 \div R_2$ de $\textbf{sort}(Q) = \textbf{sort}(R_1) \setminus \textbf{sort}(R_2)$ telle que $Q \times R_2 \subseteq R_1$.

Exemple 3.3.3: La requête (2.3.3) *« Films dirigés par Hitchcock dans lesquels il n'a pas joué »* est exprimée par :

$$\pi_{\texttt{Titre}}\sigma_{\texttt{Directeur}=\text{“}Hitchcock\text{”}}(\text{FILM}) \setminus \pi_{\texttt{Titre}}\sigma_{\texttt{Acteur}=\text{“}Hitchcock\text{”}}(\text{FILM}).$$

3.4 Théorème d'Equivalence de Codd

Nous concluons ce Chapitre avec un résultat fondamental, l'équivalence entre le calcul relationnel et l'algèbre relationnelle. La preuve de ce Théorème est hors programme, nous la donnons

néanmoins pour le lecteur intéressé. L'impact de ce Thèorème, comme nous l'avons expliqué dans le Chapitre 1 est qu'il permette à l'utilisateur de formuler ses requêtes dans un langage *descriptif* et que le logiciel pourra les exécuter, puisqu'il est capable de les transformer en une requête algébrique et donc *impérative*.

Théorème 3.4.1. (Théorème d'Equivalence) Le calcul avec la sémantique de domaine actif et l'algèbre relationnelle ont des puissances d'expression équivalentes.

Nous démontrons l'équivalence de l'algèbre et du calcul avec domaine actif.

Lemme 3.4.2. Chaque requête algébrique est exprimable dans le calcul avec domaine actif.

Démonstration. Pour simplifier la présentation, nous utilisons ici une algèbre basée sur les numéros de colonnes au lieu des noms d'attributs. Soit q une requête algébrique d'arité n. Nous construisons $q' = \{x_1, \ldots, x_n \mid \varphi_q\}$ équivalente à q par induction sur les "sous-requêtes" de q. (Une requête est construite en appliquant une opération à une ou deux sous-requêtes.) En particulier, pour une sous-requête q' de q, on définit $\varphi_{q'}$ par :

(a) si q' est R pour $R \in \mathbf{R}$: $\varphi_{q'}$ est $R(x_1, \ldots, x_{arite(R)})$.

(b) si q' est $\{u_1, \ldots, u_m\}$ où chaque u_j est un nuplet d'arité α : $\varphi_{q'}$ est

$$(x_1 = u_1(1) \wedge \cdots \wedge x_\alpha = u_1(\alpha)) \ \vee \ \ldots \ \vee \ (x_1 = u_m(1) \wedge \cdots \wedge x_\alpha = u_m(\alpha)).$$

(c) si q' est $\sigma_F(q_1)$: $\varphi_{q'}$ est $\varphi_{q_1} \wedge \psi_F$, où ψ_F est la formule obtenue à partir de F en remplacant chaque numéro de colonne i par une nouvelle variable x_i.

(d) si q' est $\pi_{i_1,\ldots,i_n}(q_1)$: $\varphi_{q'}$ est

$$\exists y_{i_1} \ldots y_{i_n}((x_1 = y_{i_1} \wedge \cdots \wedge x_n = y_{i_n}) \ \wedge \ \exists y_{j_1} \ldots \exists y_{j_l} \varphi_{q_1}(y_1, \ldots, y_{arite(q')})),$$

où $j_1, \ldots, j_l$ est une liste de $[1, arite(q')] - \{i_1, \ldots, i_n\}$.

(e) si q' est $q_1 \times q_2$: $\varphi_{q'}$ est $\varphi_{q_1} \wedge \varphi_{q_2}(x_{arite(q_1)+1}, \ldots, x_{arite(q_1)+arite(q_2)})$.

(f) si q' est $q_1 \cup q_2$: $\varphi_{q'}$ est $\varphi_{q_1} \vee \varphi_{q_2}$.

(g) si q' est $q_1 - q_2$: $\varphi_{q'}$ est $\varphi_{q_1} \wedge \neg\varphi_{q_2}$.

Le lecteur pourra vérifier les détails de cette construction. □

Lemme 3.4.3. Pour chaque requête du calcul, il existe une requête algébrique équivalente.

Point Crucial : Soit $q = \{x_1, \ldots, x_n \mid \varphi\}$ une requête du calcul. Il est facile de construire une requête algébrique construisant le domaine actif :

$$q_{adom}(\mathbf{I}) \ = \ \{\langle a \rangle \mid a \in adom(q, \mathbf{I})\}$$

Ensuite, par induction, on associe à chaque sous-formule ψ une requête algébrique q_ψ telle que :

$$\{y_1, \ldots, y_m \mid \psi\}(\mathbf{I}) \ = \ q_\psi(\mathbf{I})$$

On illustre cette construction avec quelques cas. Supposons que ψ est une sous formule de φ. Alors q_ψ est construit de la manière suivante :

(a) $\psi(y_1, \ldots, y_m)$ est $R(t_1, \ldots, t_l)$, où chaque t_i est une constante ou appartient à $\vec{y}$. Alors $q_\psi \equiv \pi_{\vec{k}}(\sigma_F(R))$, où $\vec{k}$ et F sont choisis en accord avec $\vec{y}$ et $\vec{t}$.

(b) $\psi(y_1, y_2)$ est $y_1 \neq y_2$: q_ψ est $\sigma_{1 \neq 2}(q_{adom} \times q_{adom})$.

(c) $\psi(y_1, y_2, y_3)$ est $\psi'(y_1, y_2) \vee \psi''(y_2, y_3)$: q_ψ est $(q_{\psi'} \times q_{adom}) \cup (q_{adom} \times q_{\psi''})$.

(d) $\psi(y_1, \ldots, y_m)$ est $\neg\psi'(y_1, \ldots, y_m)$: q_ψ est $(q_{adom} \times \cdots \times q_{adom}) - q_{\psi'}$.

□

4 SQL et Requêtes Agrégat

4.0.1 Objectif du chapitre

L'algèbre relationnelle fait partie des fondements des bases de données relationnelles, utilisé en interne par le SGBD pour évaluer les requêtes. Le langage *SQL* quant à lui est destiné aux utilisateurs du SGBD et, à l'heure actuelle, est le langage standard d'interrogation. Il marie d'une certaine façon l'algèbre relationnelle et le calcul relationnel de n-uplets. SQL a été standardisé par l'*ANSI* en 1986 puis par l'ISO en 1989, 1992, 1999, 2003 et 2008. Ces standards successifs ont étendu les capacités du langage. En plus de ces standards, SQL possède de nombreux dialectes spécifiques à chaque SGBD. En effet, le développement des SGBD s'est déroulé en parallèle, voire a précédé la standardisation de la norme SQL. En conséquence, seul un sous-ensemble de la norme est supporté par la grande majorité des SGBD (en particulier les SGBD open-source). Ce sous-ensemble correspond en fait à la norme SQL-92. C'est donc cette version de la norme SQL que nous allons présenter ici.

La norme SQL-92 se décompose en plusieurs parties. L'interrogation des données se fait par l'opération de *sélection* (`SELECT`), dont la sémantique est donnée par une expression du calcul relationel et dont l'exécution peut être faite par une équivalence avec des opérateurs de l'algèbre relationnelle. Le *langage de manipulation de données* (*Data Manipulation Language* ou *DML*) permet de modifier les données grâce à l'*insertion* (`INSERT`), la *mise à jour* (`UPDATE`) et la *suppression* (`DELETE`) de n-uplets. Le *langage de définition de données* (*Data Definition Language* ou *DDL*) se charge de la définition du schéma d'une BD. Enfin, SQL-92 inclut d'autres opérations comme la gestion des droits ou la gestion des transactions.

L'objectif de ce chapitre est de donner au lecteur une connaissance de la syntaxe SQL, qui lui permettra ensuite de tester la compréhension des bases de données en utilisant des SGBD existants, tels que MySQL, Oracle, IBM DB2, Microsoft SQL Server, etc. Le langage SQL étant standardisé, les exemples donnés ici fonctionneront sur n'importe quel SGBD respectant la norme SQL-92 (c'est-à-dire à peu près tous). Nous introduisons également dans ce chapitre le calcul d'agrégats, qui n'est pas traité dans le calcul relationnel simple et l'algèbre relationnelle SPJR.

4.1 Le langage de définition de données

Le langage de définition de données permet la modification du schéma d'une base de données. Il propose trois opérations : la création (`CREATE`), la suppression (`DROP`) et la modification (`ALTER`). Par exemple, la création d'une table a pour syntaxe :

```
CREATE TABLE ⟨table⟩ (⟨définitions de colonnes⟩ [⟨ contraintes de
                                    table⟩]).
```

Chaque définition de colonne comporte le nom de la colonne, son type et éventuellement une spécification de contraintes d'intégrité. Il est également possible de définir des contraintes d'intégrité sur la table. Les types de données supportés sont les suivants :

— chaînes de caractères de taille fixe, complétées à droite par des espaces (`CHAR(`*taille*`)`),
— chaînes de taille variable (`VARCHAR(`*taille*`)`),
— entiers sur 32 bits (`INTEGER`),
— nombres en précision fixe (*nbChiffres* chiffres dont *nbDecimales* après la virgule) (`NUMERIC(`*nbChiffres*`,` *nbDecimales*`)`),
— nombres en virgule flottante simple précision (`REAL`),
— nombres en virgule flottante double précision (`DOUBLE PRECISION`), et
— types date et/ou heure (`DATE/TIME/TIMESTAMP`).

Concernant les contraintes d'intégrité, le principe de base est la *clé* d'une table. La clé primaire d'une table est un ensemble d'attributs (souvent un seul attribut). Une combinaison de valeurs de ces attributs détermine au plus un nuplet de la table. En d'autres termes, deux nuplets de la table ne peuvent avoir la même projection sur ces attributs. Une table peut avoir plusieurs clés. On dira d'un clé d'une table T qu'elle est une *clé primaire* si elle est utilsée dans d'autres tables pour identifier des nuplets de T. (Les autres clés seront dites *clés candidates*.) On appelle dans ce cas *clé étrangère* un ensemble d'attributs de l'autre table faisant référence à la *clé primaire* de T. Si on se réfère à l'exemple initial de la Figure 2.1, on voit que la table COORDONNÉES a comme clé primaire l'attribut *Salle*. Cet attribut devient une clé étrangère dans la table SÉANCE, référençant la table COORDONNÉES. Il faut noter que contrairement à ce qu'on pourrait penser, l'attribut *Titre* n'est pas une clé primaire de la table FILM, puisque sa valuation n'est pas unique sur l'ensemble des nuplets de la table. Ainsi l'attribut *Titre* de la table SÉANCE n'est pas une clé étrangère non plus. Notons qu'il serait possible de *normaliser* le schéma, pour obliger toutes les tables à avoir une clé primaire. Cette opération est hors programme, mais nous allons donner un exemple un peu plus bas de transformations de ces relations pour créer des tables avec des clés primaires.

Ainsi, le langage DDL permet de spécifier qu'un ensemble d'attributs est clé primaire (`PRIMARY KEY`), qu'une séquence d'attributs est clé étrangère (`FOREIGN KEY/REFERENCES`). Il permet aussi de contraindre le domaine d'un attribut (`CHECK`) ou qu'un ensemble d'attributs ne contient pas de valeur manquante (`NOT NULL`). Enfin, il permet, à la manière de `PRIMARY KEY`, de préciser qu'un ensemble d'attributs ne contient pas de doublons (`UNIQUE`) ; à la différence de `PRIMARY KEY`, il est possible de spécifier plusieurs `UNIQUE` sur la même table.

Par exemple, la requête DDL suivante crée la table COORDONNÉES.

```
CREATE TABLE coordonnees(
  salle VARCHAR(30) PRIMARY KEY,
  adresse VARCHAR(255) UNIQUE NOT NULL,
  telephone VARCHAR(10)
);
```

L'attribut `salle` est la clé primaire donc annoté par `PRIMARY KEY`. Les contraintes `UNIQUE` et `NOT NULL` sur `adresse` précisent que cet attribut est une clé candidate.

Le deuxième exemple permet de créer la table FILM. Notons que nous allons sub-divisier la table originale présentée dans la Figure 2.1 en trois tables FILM, ACTEUR et JOUEDANS

```
CREATE TABLE film(
  idfilm INTEGER(4) PRIMARY KEY,
  titre VARCHAR(255) NOT NULL,
  directeur VARCHAR(60) NOT NULL,
  annee YEAR(4) CHECK(annee > 1900),
  titre_vo VARCHAR(255),
  CONSTRAINT film_unique_titre_directeur UNIQUE (titre, directeur)
);
```

```
CREATE TABLE acteur(
  idacteur INTEGER(4) PRIMARY KEY,
  nom VARCHAR(255),
  prenom VARCHAR(60),
  CONSTRAINT acteur_unique_nom_prenom UNIQUE (nom, prenom)
);

CREATE TABLE jouedans(
  idacteur INTEGER(4) REFERENCES acteur(idacteur),
  idfilm INTEGER(4) REFERENCES film(idfilm),
  PRIMARY KEY(idacteur, idfilm)
);
```

La contrainte référentielle précise que les attributs `idacteur` et `idfilm` de la table JOUEDANS sont des clés étrangères des attributs et tables indiquées dans la clause `REFERENCES`. Les contraintes `NOT NULL` sur les attributs `titre` et `directeur` indiquent que ces valeurs doivent nécessairement être renseignées. En revanche, le `titre_vo` peut rester vide (il aura la valeur `NULL`). Notons que des attributs `PRIMARY KEY` sont également `NOT NULL`, en revanche des attributs `UNIQUE` peuvent contenir de multiples valeurs nulles, mais seulement une valeur non nulle donnée.

La contrainte de domaine sur `annee` limite aux valeurs supérieures à 1900. La contrainte de table `PRIMARY KEY(equipe, no)` sur la table JOUEDANS définit la clé primaire. Enfin, les contraintes de table nommées `acteur_unique_nom_prenom` et `film_unique_titre_directeur` précisent d'autres clés candidates.

Exercice : Ecrire la requête DDL pour créer la table SÉANCE.

```
CREATE TABLE seance(
  idfilm INTEGER(4) REFERENCES film(idfilm),
  salle VARCHAR(30) REFERENCES coordonnees(salle),
  horaire TIME,
  PRIMARY KEY(idfilm, salle)
);
```


4.2 Le langage de manipulation de données

Maintenant que le schéma de la BD a été défini grâce au langage DDL, il reste à ajouter les données proprement dites dans la BD. C'est le rôle du langage DML.

Une insertion de n-uplets a la syntaxe suivante : `INSERT INTO ⟨table⟩ [(⟨liste d'attributs⟩)] VALUES (⟨liste de valeurs⟩) | ⟨requete⟩`. On peut préciser la liste des attributs à renseigner : les autres valeurs sont alors fixées à la valeur par défaut de l'attribut ou à `NULL`. La clause `VALUES` ajoute un n-uplet à la fois alors que l'utilisation d'une requête à la place de `VALUES` permet d'insérer plusieurs n-uplets en une seule opération (*Bulk insert*).

La requête suivante permet l'ajout du n-uplet ⟨'Le Champo', '51 rue des Ecoles', '0143545160'⟩ dans la relation COORDONNÉES :

```
INSERT INTO coordonnees
VALUES('Le Champo', '51 rue des Ecoles', '0143545160');
```

La deuxième requête insère un n-uplet en ne précisant que les attributs `salle` et `adresse`. Ce n-uplet n'aura pas de valeur pour l'attribut `telephone` (`NULL`). Notons que l'ordre des attributs n'est pas forcément celui de la définition dans la table.

```
INSERT INTO coordonnees(adresse, salle)
VALUES('31 boulevard des Italiens', 'Gaumont Op\'era');
```

La troisième requête insere un n-uplet dans la table FILM :

```
INSERT INTO film(idfilm, titre, directeur, annee, titre_vo)
VALUES(1, 'Les Oiseaux', 'Hitchcock', 1963, 'The Birds');
```

Enfin, l'insertion peut utiliser des requêtes que nous considérerons dans la prochaine section. Par exemple, la requête suivante insère dans la table SÉANCE tous les films de Hitchcock au cinéma "Le Champo".

```
INSERT INTO seance(salle, idfilm)
  SELECT 'Le Champo', f.idfilm
  FROM film f
  WHERE directeur = 'Hitchcock'
```

L'instruction `UPDATE` modifie les valeurs des n-uplets d'une table : `UPDATE ⟨table⟩ SET ⟨liste d'affectations⟩ [WHERE ⟨condition⟩]`. L'utilisation d'une liste de valeurs permet de modifier la valeur de plusieurs attributs. La clause `WHERE` définit les n-uplets qui seront mis à jour.

La requête suivante modifie l'horaire du film 1 de la salle "Le Champo".

```
UPDATE seance
   SET horaire = '21:00'
 WHERE salle = 'Le Champo' AND idfilm = 1;
```

L'instruction `DELETE` supprime des n-uplets d'une table : `DELETE FROM ⟨table⟩ [WHERE ⟨condition⟩]`. La clause `WHERE` précise les n-uplets à supprimer.

Par exemple, la requête suivante supprime le film 1 de la liste des films projetés au "Champo".

```
DELETE FROM seance
 WHERE idfilm=1 and salle='Le Champo';
```

Sans clause `WHERE`, tous les n-uplets de la table sont supprimés. L'instruction suivante effacerait tout le contenu de la table SÉANCE.

```
DELETE FROM seance;
```

4.3 L'interrogation des données

4.3.1 La syntaxe SQL du SELECT, FROM, WHERE

Le dernier aspect du langage SQL que nous étudierons est l'interrogation des données grâce à l'instruction `SELECT` :

```
SELECT ⟨liste d'expressions⟩
FROM ⟨liste de tables⟩
WHERE ⟨conditions⟩
GROUP BY ⟨liste d'attributs⟩
HAVING ⟨conditions⟩
ORDER BY ⟨liste d'attributs⟩
```

Les rôles des trois prières lignes sont les suivants :

— La clause `SELECT` spécifie le schéma de sortie (projection).

— La clause `FROM` précise les tables impliquées et leurs liens (produit cartésien et jointures).

— La clause `WHERE` fixe les conditions que doivent remplir les n-uplets résultats (sélection).

Les trois dernières clauses nous font sortir du calcul et de l'algèbre relationnelle :

— La clause `GROUP BY` comment regrouper des nuplets (agrégation) ?

— La clause `HAVING` impose une condition sur les groupes (i.e. permet d'éliminer l'intégralité d'un groupe, en se basant sur la valeur d'un agrégat calculé sur ce groupe).

— Enfin `ORDER BY` définit les critères de tris des résultats.

La clause `SELECT` est suivie d'une liste d'expressions. Chaque expression peut être un attribut, un littéral entre guillemet ou une expression calculée. Le caractère '*' est un joker signifiant « tous les attributs de la table ». Il est à noter que la clause `SELECT` n'interdit pas a priori les doublons, i.e. plusieurs n-uplets égaux. Alors que le modèle relationnel et l'algèbre manipulent des ensembles, en SQL, une table n'est pas un ensemble de n-uplets mais un *multi-ensemble*. La raison en est simple : éliminer les doublons est coûteux. Les requêtes *SELECT/FROM/WHERE* sont définies sur des multi-ensembles par défaut. Pour se ramener à un ensemble et donc éliminer les doublons, il faut alors utiliser `DISTINCT` dans la clause `SELECT`.

4.3.2 Traduction en calcul relationnel

La traduction en calcul relationnel est directe, et se fait de la manière automatique suivante. Soit une requête SQL Q de la forme :

SELECT $A_1, A_2, \ldots, A_n$
FROM $R_1, R_2, \ldots, R_m$
WHERE CONDITION$(A_1, \ldots, A_n, A_{n+1}, \ldots, A_{n+k})$

Une expression de calcul relationnel correspondante se déduit directement par la relation suivante :

$$res = \{A_1, A_2, \ldots, A_n | \exists A_{n+1}, \ldots, A_{n+k}, (CONDITION_{calcul}(A_1, \ldots, A_{n+k})\}$$

où $CONDITION_{calcul}$ lie les attributs aux tables correspondantes, ce qui de fait génère les jointure, ainsi que les conditions de restriction de la clause `WHERE`.

Notons pour être précis que la requête SQL a pour résultat un multi-ensemble, contenant peut-être des doublons. Il faudrait utiliser `DISTINCT` pour les éliminer.

4.3.3 Traduction en algèbre relationnelle

La traduction en algèbre relationnelle se fait de la manière suivante :
$\pi_{A_1,\ldots,A_n}(\sigma_{CONDITION}(R_1 \times R_2 \times \ldots \times R_m)$
Notons que nous considérons ici dans la clause `FROM` uniquement des tables et donc le produit cartésien. Nous verrons plus tard qu'il est possible d'indiquer directement dans cette clause `FROM` des jointures, ce qui change légèrement la traduction.

4.3.4 Exemple de requêtes

1. Nom et prénom des acteurs

SQL	**SELECT** nom, prenom **FROM** acteur;
Algèbre	$\pi_{nom,prenom}(\text{ACTEUR})$
Calcul	$res = \{x_{nom}, x_{prenom} \mid \exists x_{id}, \text{ACTEUR}(x_{id}, x_{nom}, x_{prenom})\}$

2. Tous les attributs des acteur

SQL	**SELECT** * **FROM** acteur;
Algèbre	$\pi_{idacteur,nom,prenom,age}(\text{ACTEUR})$
Calcul	$res = \{x_{id}, x_{nom}, x_{prenom} \mid \text{ACTEUR}(x_{id}, x_{nom}, x_{prenom})\}$

3. Noms des acteurs (sans doublons) pour SQL. L'algèbre et le calcul travaillent directement sur des ensembles et non des multi-ensembles.

SQL	**SELECT DISTINCT** nom **FROM** acteur;

La clause `FROM` accepte en paramètre une liste de tables. La requête porte alors sur le produit cartésien de ces tables. Pour effectuer une jointure entre les tables, il convient d'utiliser les opérateurs de jointure du langage. Le mot-clé `JOIN` placé entre deux tables indique qu'il faut en faire la jointure. La condition de jointure est précisée avec le mot-clé `ON`. Une jointure naturelle s'exprime avec le mot-clé `NATURAL JOIN`.

L'exemple suivant montre deux requêtes de jointure.

1. Titre d'un film avec la liste des identifiants de ses acteurs. Dans le cas de l'algèbre, on utilise la jointure naturelle, c'est-à-dire que ce sont les attributs ayant le même nom dans les deux tables jointes qui sont choisis, et l'opérateur utilisé est l'égalité.

SQL	**SELECT** f.titre **AS** titre, jd.idacteur **AS** id **FROM** film f **JOIN** jouedans jd **ON** f.idfilm = jd.idfilm;
ou bien	**SELECT** f.titre **AS** titre, jd.idacteur **AS** id **FROM** film f jouedans jd **WHERE** f.idfilm = jd.idfilm;
Algèbre	$\pi_{titre,idacteur}(\text{FILM} \bowtie \text{JOUEDANS})$
ce qui signifie	$\pi_{titre,idacteur}(\text{FILM} \bowtie_{\text{FILM}.idfilm=\text{JOUEDANS}.idfilm} \text{JOUEDANS})$
Calcul	$res = \{titre, idacteur \mid \exists(idfilm, directeur, annee),$ $\text{FILM}(idfilm, titre, directeur, annee)$ $\wedge \text{JOUEDANS}(idfilm, idacteur)\}$

2. Les noms des salles projetant un film de Hitchcock.

SQL	**SELECT** f.titre **AS** titre, s.salle **AS** cinema **FROM** film f, seance s **WHERE** f.idfilm = s.idfilm **AND** f.directeur = 'Hitchcock' **AND** s.horaire IS **NOT NULL**;
Algèbre	$\pi_{titre,salle}((\sigma_{directeur=\text{“}Hitchcock\text{”}}\text{FILM}) \bowtie \text{SÉANCE})$
Calcul	$res = \{titre, salle \mid \exists(idfilm, annee, horaire),$ $\text{FILM}(idfilm, titre, \text{“}Hitchcock\text{”}, annee)$ $\wedge \text{SÉANCE}(idfilm, salle, horaire)\}$

On pourra s'interroger sur l'optimalité de l'expression d'algèbre relationnelle du 2^e exemple.

Dans l'exemple, des alias d'attributs et de tables sont utilisés : la syntaxe `AS cinema` dans la clause `SELECT` permet de renommer la colonne en sortie, la syntaxe `film f` définit `f` comme alias de la table `film`.

La clause `WHERE` permet d'exprimer la sélection. SQL supporte des opérateurs de comparaison (`<, <=, =, >=, >, <>`), logiques (`AND, OR`), de test de valeur manquante (`IS NULL, IS NOT NULL`), de recherche textuelle (`LIKE`), de sélection d'intervalle (`BETWEEN`), de liste (`IN`), et d'imbrication de blocs (`IN, EXIST, NOT EXIST, ALL, SOME, ANY`).

Les requêtes suivantes illustrent ces notions.

1. Titre des films datant d'avant 2000

SQL	**SELECT** titre **FROM** film **WHERE** annee < 2000;
Algèbre	$\pi_{titre}(\sigma_{annee<2000}(\text{FILM}))$
Calcul	$res = \{titre \mid \exists(idfilm, directeur, annee),$ $\text{FILM}(idfilm, titre, directeur, annee)$ $\wedge(annee < 2000)\}$

2. Titre et directeur des films produits entre 2003 et 2013 dont le titre commence par 'L'

SQL	**SELECT** titre, directeur **FROM** film **WHERE** annee **BETWEEN** 2003 **AND** 2013 **AND** titre **LIKE** 'L%';
Algèbre	$r_1 = \sigma_{annee>2013 \wedge annee<2000 \wedge titre\ LIKE\ 'L\%'}(\text{FILM})$ $\pi_{titre,directeur}(r_1)$
Calcul	$res = \{titre, directeur \mid \exists(idfilm, annee),$ $\text{FILM}(idfilm, titre, directeur, annee)$ $\wedge(annee > 2000)$ $\wedge(annee < 2013)\}$

4.4 Requêtes agrégats

Cette section discute de traitements d'agrégation (e.g. calcul de moyenne) qui ne sont pas couverts par l'algèbre relationnelle classique et le calcul relationnel. Nous expliquons ici le principe sur SQL, et étendons l'algèbre pour représenter ces concepts. Nous ne considérons donc plus ici le calcul relationnel.

Les fonctions d'agrégation, les clauses `GROUP BY` et `HAVING` permettent de « résumer » un ensemble de n-uplets. La clause `GROUP BY` précise les attributs de groupement. Les fonctions d'agrégation sont ensuite appliquées sur chaque groupe. Enfin, la clause `HAVING` permet de spécifier des conditions sur les groupes. Il s'agit donc ici d'une condition *globale* à un groupe. Ces conditions peuvent être la valeur min, max, moyenne, ou la somme d'un attribut a sur un groupe (`MIN(a)`, `MAX(a)`, `AVG(a)`, `SUM(a)`), le nombre de lignes d'un groupe (`COUNT(*)`), ou le nombre de lignes d'un groupe ayant une valeur distincte d'un attribut `a` particulier (`COUNT(distinct a)`).

— Année de production moyenne des films

SQL	**SELECT AVG**(annee)**FROM** film;
Algèbre	$\gamma_{AVG(annee)}(\text{FILM})$

— Nombre total de films dans lequel chaque acteur à joué

SQL	**SELECT** idacteur **AS** id, **COUNT**(*)**AS** Nb_films **FROM** film f **NATURAL JOIN** jouedans jd **GROUP BY** idacteur
Algèbre	$r_1 = \text{FILM} \bowtie \text{JOUEDANS}$ $\pi_{idacteur}\gamma_{COUNT(*)}(r_1)$

— Année de production la plus ancienne pour chaque acteur ayant joué dans au moins 3 films de Hitchcock.

SQL	**SELECT** idacteur, **MIN**(annee)**AS** annee_debut **FROM** film f **NATURAL JOIN** jouedans jd **WHERE** f.directeur='Hitchcock' **GROUP BY** idacteur **HAVING COUNT**(*)> 2;
Algèbre	$r_1 = \sigma_{directeur=\text{“}Hitchcock\text{”}}\text{FILM} \bowtie \text{JOUEDANS}$ $r_2 = \rho_{MIN(annee)\rightarrow annee\_debut}({}_{idacteur}\gamma_{MIN(annee),COUNT(*)}(r_1))$ $\pi_{annee\_debut,idacteur}\sigma_{COUNT(*)>2}(r_2)$

4.5 Requêtes ensemblistes

SQL permet également de regrouper les résultats de plusieurs requêtes à l'aide d'opérations ensemblistes. La requête ⟨*requete*1⟩ `UNION` ⟨*requete*2⟩ retourne l'union des résultats des deux requêtes. La requête ⟨*requete*1⟩ `INTERSECT` ⟨*requete*2⟩ retourne l'intersection des résultats des deux requêtes. La requête ⟨*requete*1⟩ `EXCEPT` ⟨*requete*2⟩ retourne la différence des résultats des deux requêtes. Pour ces trois opérations, les schémas des requêtes doivent être compatibles (même nombre d'attributs, même types). Ces opérations s'effectuent sur des ensembles, i.e. les doublons sont éliminés. Le mot-clé `ALL` ajouté après un opérateur ensembliste évite l'élimination des doublons et permet donc de manipuler des multi-ensembles.

4.6 Tri

Une dernière clause `ORDER BY` permet de spécifier un tri des résultats. Donc nous sommes passés des ensensembles, aux multi-ensembles, pour atteindre les listes. On notera que les tables stockées sont des ensembles et pas des multi-ensembles ou des listes.

La clause `ORDER BY` est suivie d'une liste d'attributs pour lesquels on précise l'ordre de tri (`ASC` pour croissant, `DESC` pour décroissant). Cette clause, qui n'a pas d'équivalent en algèbre relationnelle, ne doit apparaître qu'une seule fois dans une requête et obligatoirement à la fin. En effet, elle transforme un ensemble de n-uplets en liste de n-uplets pour représenter l'ordre. Il n'est donc pas possible d'appliquer un autre opérateur sur le résultat de cette clause.

Par exemple,

— Nom et prénom des acteurs triés en ordre croissant selon le nom puis le prénom

```
SELECT nom, prenom FROM acteur
ORDER BY nom ASC, prenom ASC;
```

5 Exercices

5.1 Objectif du chapitre

Le programme officiel propose trois activités (ou exercices) que nous détaillerons dans la Section 5.2 :

— utiliser une application de création et de manipulation de données, offrant une interface graphique, notamment pour créer une base de données simple, ne comportant pas plus de trois tables ayant chacune un nombre limité de colonnes. L'installation et l'exploitation d'un serveur SQL ne fait pas partie des attendus.

— lancer des requêtes sur une base de données de taille plus importante, comportant plusieurs tables, que les étudiants n'auront pas eu à construire, à l'aide d'une application offrant une interface graphique

— enchaîner une requête sur une base de données et un traitement des réponses enregistrées dans un fichier.

On constate que le programme commence directement par des activités sur machine. De notre point de vue, il peut être préférable de consacrer au moins une séance à la réflexion, sur papier, en particulier si on souhaite travailler sur les expressions du calcul et de l'algèbre. Pour le travail sur machine, il est important de noter que la plpupart des SGBDs disponibles sur le marché (même ceux qui ne sont pas en open source) sont gratuits dans le cadre d'une utilisation non commerciale. Ainsi par exemple Oracle, qui est le SGBD le plus utilisé, peut être téléchargé et utilisé gratuitement, tout comme d'autres logiciels concurrents tels Microsoft SQL Server ou IBM DB2. Pour l'utilisation qui en sera faite, et parce qu'ils sont en général plus simples à installer, nous recommandons plutôt l'utilisation d'un logiciel libre de type MySQL ou PostgreSQL, installables par exemple avec le package EasyPHP [1]. Il faut noter que les deux premières activités sont identiques : en effet, une requête ne dépend pas de la taille de la base de données, ni de son contenu, mais uniquement de sa structure ! Le seul intérêt de lancer des requêtes sur une "petite" base de données, est qu'il est possible de vérifier à la main que les résultats sont corrects, ce qui est très utile surtout pour débuter. Concernant le schéma de la base de données, en réalité le nombre de tables ne rend pas plus difficile la réflexion, car on se ramène toujours à la même procédure par le biais des jointures. Avec un schéma compliqué, il faudra simplement faire plus de jointures. Aussi, dans les exercices proposés ici, nous nous restreignons à un schéma de trois tables.

Le programme officiel ne donne aucun exemple concrêt d'application : en effet, les bases de données étant partout, *n'importe quelle application* peut convenir : salles de cinéma, horaires de trains, rencontres de championnat de foot, comptes bancaires, résultats de manipulations d'expériences, bibliothèque, etc... Dans la Section **??** nous allons discuter de la conception des schémas de bases de données et des requêtes adaptées aux questions que peuvent résoudre les élèves de CPGE, et nous donnerons quelques exemples corrigés.

1. `http://www.easyphp.org/`

Enfin, en Section **??** nous proposerons quelques exercices à faire sur papier, dans ce qui pourrait être une première séance.

5.2 Activités du Programme Officiel

Il est possible de commencer soit (a) par la création d'un schéma de BD, son remplissage et son interrogation, ou bien (b) d'arriver sur une base de données déjà créé et remplie et de simplement l'interroger. Nous suggérons de commencer *avec les élèves* sur une base de données déjà créée et remplie, puisque comme la conception de schémas n'est pas au programme, la seule activité qu'on peut proposer est de recopier en quelque sorte un schéma déjà fait sur papier, ce qui n'est pas forcément très accrocheur. Toutefois, pour cela, il faut disposer d'un schéma déjà fait et de données déjà préparées. Nous mettons à disposition à cette adresse [2] le fichier `bd-banque-simple.sql` qui correspond à la BD nommée *banque-simple* que nous utiliserons dans la suite.

5.2.1 Installation du SGBD *MySQL*, du serveur web *Apache*, de l'application *PHPMyAdmin* et de la BD *banque-simple*

L'architecture mise en place est une architecture trois-tiers (trois niveaux) :

— le premier niveau correspond au SGBD *MySQL* qui contiendra les données et exécutera les requêtes

— le deuxième niveau correspond au serveur Web Apache, qui exécute l'application web PHPMyAdmin (un ensemble de pages Web et de programmes écrits en langage PHP)

— le troisième niveau correspond au navigateur Web de l'utilisateur qui interagit avec le premier niveau *via* l'application PHPMyAdmin.

En effet, le navigateur Web de l'utilisateur ne se connecte jamais directement à la base de données, il doit passer par le serveur Web pour poser des requêtes. Le serveur Web peut servir à autre chose, comme par exemple gérer un site Web, nous le verrons dans la Section **??**.

Nous prenons comme exemple l'installation de la suite *EasyPHP* sur windows, qui contient à la fois MySQL, Apache, et PHPMyAdmin. L'installation a été faite en janvier 2014 avec la version 13.1 de EasyPHP DevServer, sur Windows. Rendez-vous à l'adresse : `http://www.easyphp.org/` et téléchargez la version 13.1 VC 11 si vous avez Windows 7 ou 8 et VC 9 si vous avez une version précédente. Nous espérons que les utilisateurs de Linux sauront installer correctement un package similaire (par exemple le package *LAMP* [3]). Exécutez ensuite le fichier téléchargé (dans cette version : `EasyPHP-DevServer-13.1VC9-setup.exe`). L'installeur demande le choix d'une langue, vous pouvez choisir Français, puis quelques clics plus loin vous demande un répertoire d'installation. Vous pouvez le placer où vous voulez, mais notez bien où se trouve ce répertoire d'installation [4], vous en aurez besoin par la suite pour l'installation de PHPMyadmin, ou encore si vous voulez créer un site Web ! Au bout de quelques minutes, votre installation est terminée. Vous pouvez ensuite lancer EasyPHP, qui une fois en route, se présenté sous la forme d'une lettre "e" noire dans la barre des tâches. Un clic droit sur cette icône ouvre un menu contextuel et un clique gauche ouvre une fenêtre permettant de voir l'état des serveurs (serveur MySQL et serveur Apache), de les relancer, de modifier leur configuration, etc... Effectuons un clic droit sur l'icône, puis choisissons "Web Local". Notre navigateur est lancé, et nous arrivons sur la page Web indiquée dans la Figure 5.1. Cette page correspond au contenu du répertoire publié par le serveur Web Apache, qui se trouve dans le sous répertoire

2. `http://cassioppee.prism.uvsq.fr/cpge/bd-banque-simple.sql`
3. http ://en.wikipedia.org/wiki/LAMP_%28software_bundle%29
4. Dans notre exemple, nous supposerons que ce répertoire est `D:/EasyPHP/`

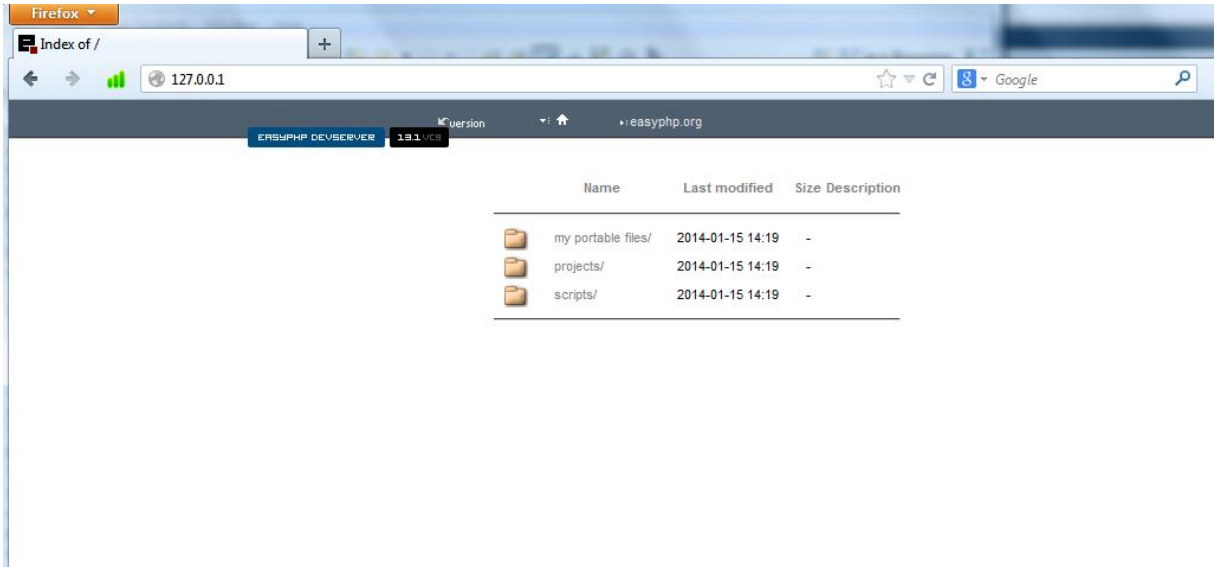

FIGURE 5.1– Page de présentation du serveur web local

`data/localweb/` du répertoire d'installation [5]. Notez que l'URL pour accéder au serveur local est le `http://127.0.0.1` également accessible via l'URL `http://localhost` .

Il faut encore installer l'interface PHPMyAdmin, ce qui se fait très simplement : il faut copier le répertoire nommé `phpmyadminXXXXXXX` qui se trouve dans le sous-répertoire `modules` du répertoire d'installation d'EasyPHP vers le sous-répertoire `data/localweb/` (voir Figure 5.2).

Une fois le répertoire copié, rechargez la page 127.0.0.1, vous devriez voir apparaitre de nouveau répertoire sur la page web (Figure 5.3) . Cliquez dessus, vous arrivez sur la page de présentation du logiciel `PHPMyAdmin` (Figure 5.4). Notez que nous nous sommes connecté à la base sans entrer le moindre mot de passe, ni d'identifiant utilisateur. Dans la configuration installée de base (qui peut être changée), l'utilisateur est `root` (il possède tous les droits) et n'a pas de mot de passe ! Sur la gauche de l'écran, dans la zone cerclée sur la figure, on peut voir quatre lignes, qui correspondent chacune à une base de données. Si on clique sur l'un de ces noms, on pourra accéder au contenu de ces bases de données. Comme il s'agit de bases de données système et de la base de données de PHPMyAdmin, il ne faut pas les modifier.

À partir ce cette page, nous pourrions créer une base de données, en créant des tables "à la main". De manière plus simple, nous allons en importer une déjà existante, en utilisant le fichier `bd-banque-simple.sql` téléchargeable via `http://cassioppee.prism.uvsq.fr/cpge/bd-banque-simple.sql` en notant bien où il est téléchargé sur votre ordinateur. Tout d'abord, nous effectuons une manipulation spécifique de MySQL, qui consiste en la création d'un espace portant un nom spécifique (nous allons choisir le nom *banque*) afin d'y stocker les tables. Notons que l'utilité de cet espace (appelé "base de données" par les développeurs de MySQL) est de permettre une gestion des droits des utilisateurs sur les tables qu'il contient. Nous indiquons dans la table 5.5 la marche à suivre pour créer une base de données.

Une fois la base de données créée, elle apparait dans la liste des bases de données présentes sur le serveur, dans le menu sur la gauche de la fenêtre. Il faut donc *cliquer* sur son nom dans la fenêtre de gauche pour la sélectionner, avant d'exécuter la liste des instructions présentes dans le fichier `bd-banque-simple.sql`, comme indiqué dans la Figure 5.6.

Exemple 5.2.1: Le fichier `bd-banque-simple.sql` contient les instructions suivantes qui permettent de créer le schéma de la base de données, ainsi que les deux clés étrangères liant la table `compte` à `client` par le biais de la contrainte nommée `compte_ibfk_1` et qui impose

5. Soit `D:/EasyPHP/data/localweb` dans notre exemple.

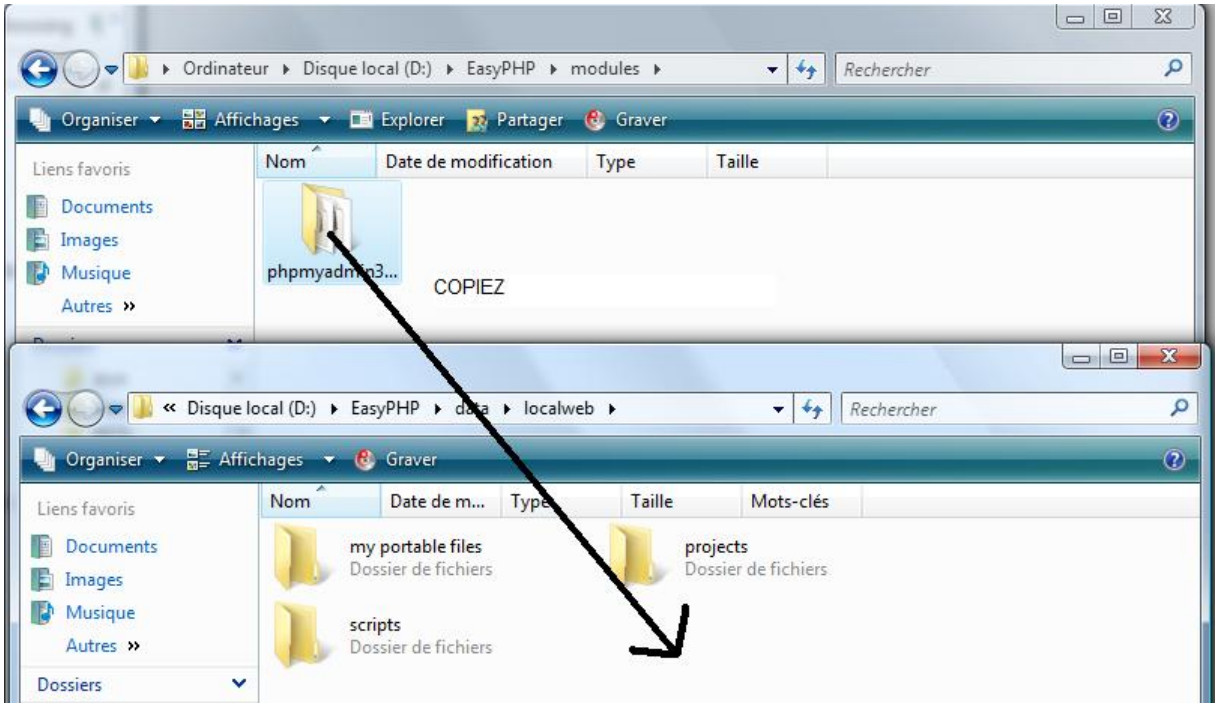

FIGURE 5.2– Copie du répertoire PHPMyAdmin

5

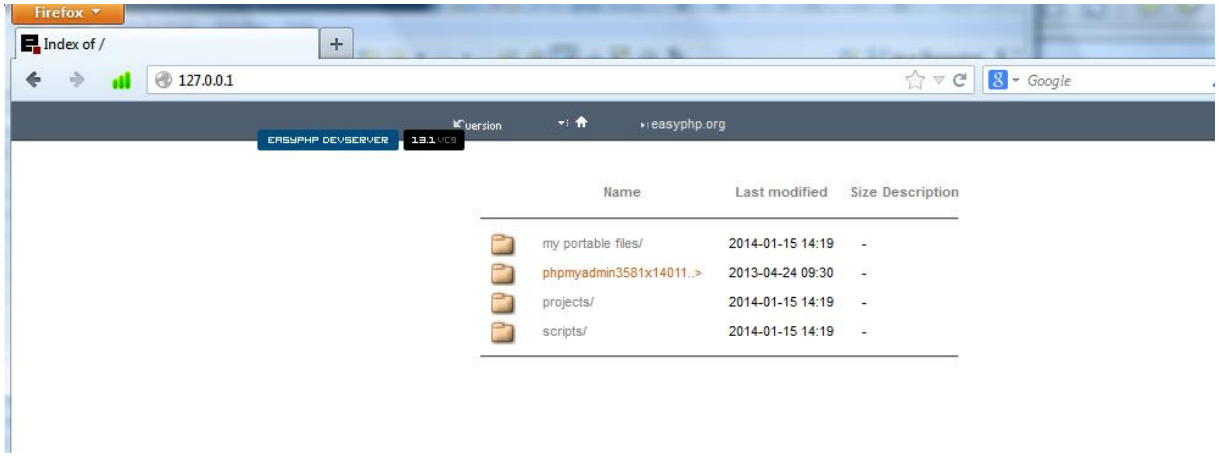

FIGURE 5.3– Page de présentation une fois PHPMyAdmin installé

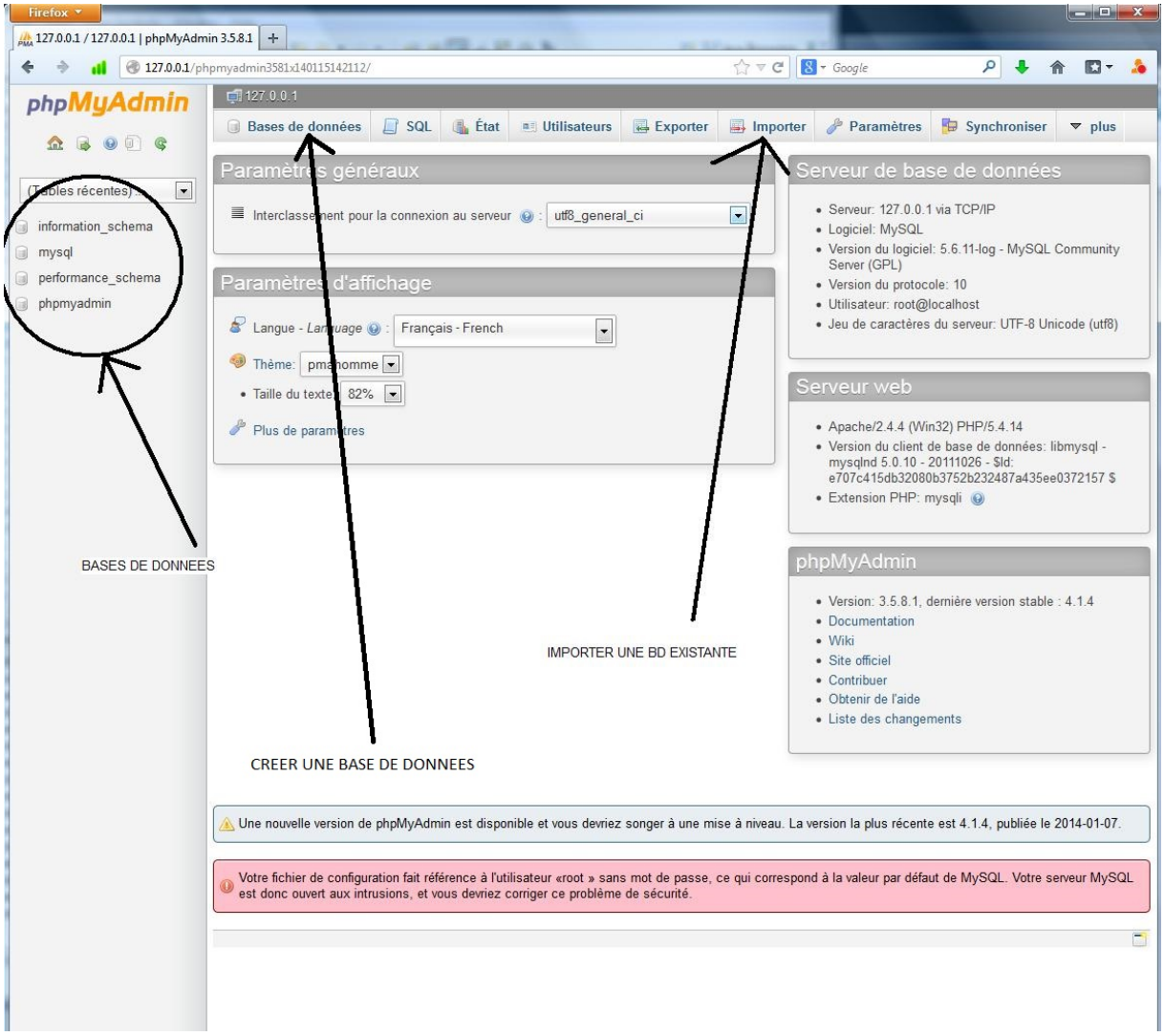

FIGURE 5.4– Première page de PHPMyAdmin

FIGURE 5.5– Création d'une Base de Données

que `compte.idproprietaire = client.idclient` et la table `operation` à `compte` par le biais de la contrainte nommée `operation_ibfk_1` et qui impose que les valeurs prises par `operation.idcompte` soient les mêmes que les valeurs de `compte.idcompte`.

```
DROP TABLE IF EXISTS operation;
DROP TABLE IF EXISTS compte;
DROP TABLE IF EXISTS client;

CREATE TABLE IF NOT EXISTS client (
  idclient int(5) NOT NULL,
  nom varchar(128) NOT NULL,
  prenom varchar(128) NOT NULL,
  ville varchar(128) NOT NULL,
  PRIMARY KEY (idclient)
);


CREATE TABLE IF NOT EXISTS compte (
  idcompte int(5) NOT NULL,
  idproprietaire int(5) NOT NULL,
  type varchar(128) NOT NULL,
  PRIMARY KEY (idcompte),
  KEY idproprietaire (idproprietaire)
);


CREATE TABLE IF NOT EXISTS operation (
  idop int(5) NOT NULL,
  idcompte int(5) NOT NULL,
  montant decimal(10,2) NOT NULL,
  informations text NOT NULL,
  PRIMARY KEY (idop),
  KEY idcompte (idcompte)
);


ALTER TABLE compte
  ADD CONSTRAINT compte_ibfk_1 FOREIGN KEY (idproprietaire)
  REFERENCES client (idclient) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE operation
  ADD CONSTRAINT operation_ibfk_1 FOREIGN KEY (idcompte)
  REFERENCES compte (idcompte) ON DELETE CASCADE ON UPDATE CASCADE;
```

La table `client` contient des informations de base sur les clients de la banque. La table `compte` indique qui est le propriétaire de chaque compte dans la banque. La table `operation` contient toutes les opérations sur les comptes de la banque. Les contraintes de clé étrangères imposent que l'identifiant du client possédant un compte donné doit déjà appartenir à la banque, et que les opérations doivent se faire sur un compte déjà existant également. En effet, si on tente par exemple de créer directement un compte sans avoir préalablement créé son propriétaire, l'opération sera refusée par le SGBD.

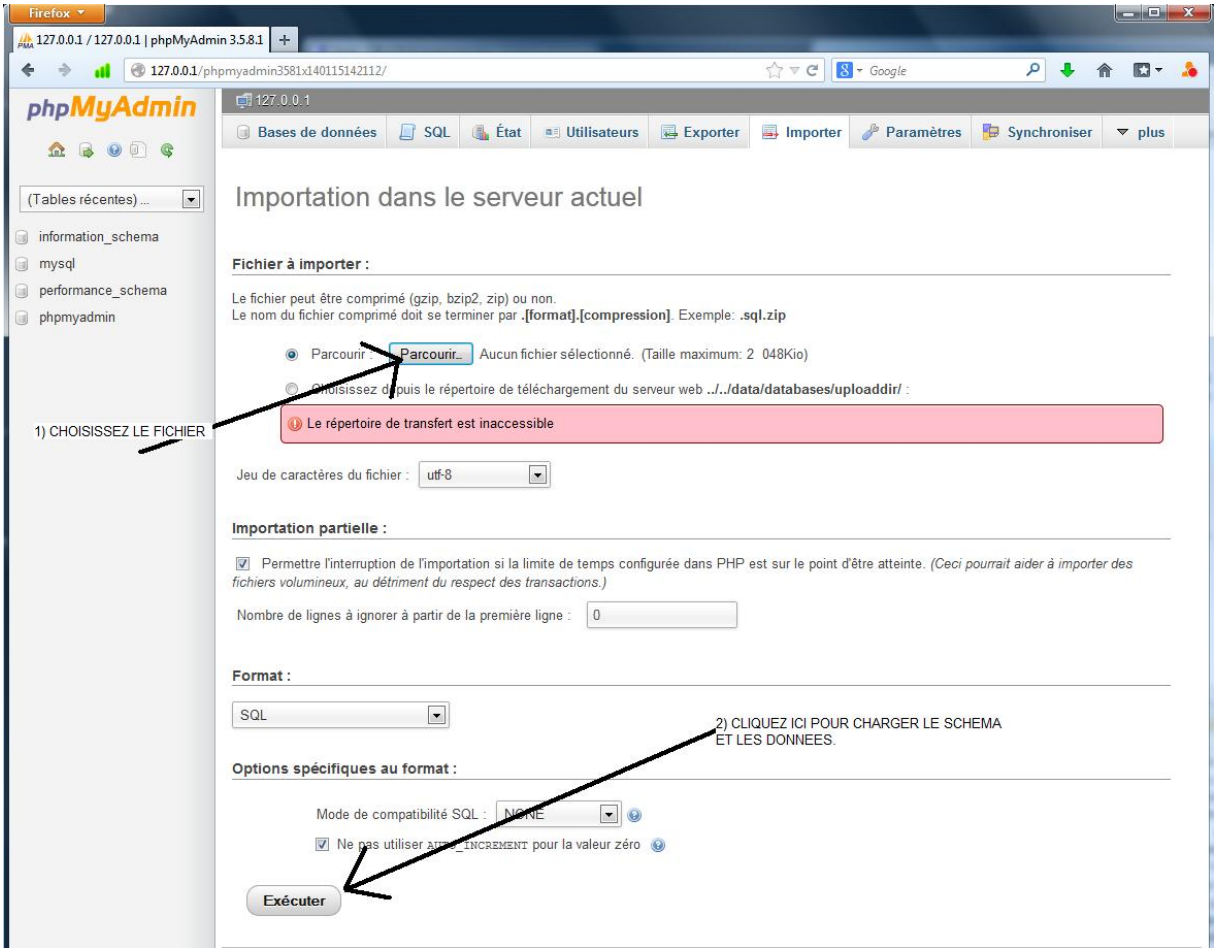

FIGURE 5.6– Chargement d'un schéma et des données d'une BD

Il faut ensuite remplir la base avec des données. Ceci peut être fait manuellement, ou bien en utilisant un fichier avec les données. Dans ce premier TP, nous allons utiliser des données déjà constituées et disponibles à l'URL : `http://cassioppee.prism.uvsq.fr/cpge/bd-banque-simple-data.sql` sous le format d'instructions SQL de type `INSERT INTO ...`. Il est également très simple d'insérer des données dans la base, en utilisant l'interface ou en écrivant les requêtes SQL correspondantes. L'opération de chargement se fait par le même menu *Importer* que précédemment. Il faut simplement choisir le fichier `bd-banque-simple-data.sql` à la place.

5.2.2 De la présentation des exercices

Une fois le schéma de la base de données créé et qu'elle a été peuplée par des données, nous sommes prêts à commencer les exercices ! Remplir la base de données prend du temps, si c'est fait à la main, ce qui peut être le cas pour les premiers exercices, puisque c'est intéressant de permettre aux élèves de voir les données pour se convaincre que le résultat de leur requête est correct. Toutefois, une fois ces premiers exemples passés, nous suggérons d'utiliser une base de données suffisamment grosse, d'une part pour plus de réalisme, et d'autre part puisque sur des jeux de données très petits il est parfois possible d'obtenir la bonne réponse avec une requête fausse. Il est possible de fournir le resultat de la requête à l'élève, ce qui lui permet de contrôler que sa requête est correcte. Sur une base de données avec des milliers de lignes il y a de fortes chances que si les résultats sont les mêmes, la requête soit bonne.

Notons qu'il est possible d'utiliser des logiciels pour générer automatiquement des données, tels que `http://www.generatedata.com/` (logiciel sous licence GNU) qu'il faudra ensuite charger dans la base de données, en utilisant la fonction d'importation. Ces logiciels peuvent générer des données dans plusieurs formats (SQL, CSV, XML...) en général tous ces formats sont acceptés par l'import de MySQL. Nous fournissons, à titre d'exemple, les fichiers suivants générés automatiquement avec ce logiciel. Dans les exercices, nous utiliserons les données générées automatiquement pour tester les résultats de nos requêtes.

— `http://cassioppee.prism.uvsq.fr/cpge/bd-banque-simple-random-data-client.sql`

— `http://cassioppee.prism.uvsq.fr/cpge/bd-banque-simple-random-data-compte.sql`

— `http://cassioppee.prism.uvsq.fr/cpge/bd-banque-simple-random-data-operation.sql`

5.2.3 Méthodologie pour répondre à une question

Nous proposons une méthodologie simple pour répondre à une requête basique exprimée "en français". Il peut être nécessaire d'appliquer plusieurs fois cette méthodologie sur des parties de la requête, puis de les recombiner ensuite à l'aide d'opérateurs ensemblistes. E.g. la requête "Trouver les noms de personnes ne possédant pas de livret A" peut se décomposer en deux requêtes basiques R1 = "Trouver les noms de toutes les personnes" et R2 = "Trouver les noms de toutes les personnes possédant un livret A" combinées par l'opérateur `différence ensembliste`[6]. Notons que cette requête pourrait également s'écrire directement, mais dans ce cas, la clause `WHERE` serait plus compliquée, puisqu'elle devrait exprimer le fait qu'une personne donnée *ne possède aucun Livret A*, ce qui se fera également avec une sous requête.

Cette méthodologie se propose de remplir une requête de type `SELECT ...FROM ...WHERE...` de la manière suivante.

1. Construction de la clause SELECT
   (a) Identifier les attributs qui doivent être affichés, et les tables les contenant.
   (b) Mettre ces tables dans la clause `FROM` et donner un nom de vari able à chaque table.
   (c) Mettre les attributs à affichés, préfixés par le nom de variable de la table correspondante dans la clause `SELECT`
   (d) Identifier les fonctions à calculer, et les attributs nécessaires au calcul de ces fonctions, puis si nécessaire, ajouter des tables supplémentaires dans la clause `FROM`, et enfin ajouter la fonction dans la clause `SELECT`
2. Construction des Groupes
   (a) Identifier les attributs utilisés pour le partitionnement et les mettre dans la clause `GROUP BY`.
   (b) Compléter si nécessaire les tables de la clause `FROM`.
   (c) Compléter la clause `SELECT` en rajoutant les attributs utilisés pour le partitionnement. En général, ces attributs auront déjà été identifiés car ils vont a priori apparaitre dans le résultat final, puisqu'ils s'agissent de critères de regroupement. En effet, est ce qu'une requête qui calculerait le revenu moyen en fonction de la ville et qui n'afficherait pas le nom de la ville de chaque groupe aurait une quelconque utilité ?
3. Construction des restrictions
   (a) Regarder sur quels attributs (ou groupes d'attributs) de quelles tables portent les restrictions.

6. Malheureusement, cet opérateur n'est pas implémenté dans MySQL !

(b) Rajouter ces tables, si elles ne sont pas encore présentes, dans la clause `FROM`.

(c) Rajouter les restrictions sur les attributs en question dans la clause `WHERE`. Attention, il est capital de noter qu'une condition de restriction *peut tout à fait* s'exprimer en utilisant une sous-requête. E.g. Voir exemple "idproprietaire sans livret A"

(d) Rajouter les restrictions qui portent sur des valeurs agrégées dans la clause `HAVING`.

4. Construction des jointures

(a) Rajouter dans la clause `WHERE` les conditions de jointure, permettant de lier toutes les tables de la clause `FROM` les unes aux autres. Une telle jointure se fait en utilisant les clés étrangères des tables en question. Il peut arriver qu'il ne soit pas possible de lier une table aux autres. Cela signifie qu'il faudra introduire une ou plusieurs autres tables intermédiaires permettant de lier cette table par le biais d'un chemin composé de clés étrangères. En général, une clé étrangère est mono-valuée, donc si on a n tables dans la clause `FROM` il faudra avoir $n-1$ conditions de jointure dans la clause `WHERE`.

5. Construction des sous-requêtes

(a) Pour chaque sous requête utilisée dans la clause `WHERE`, vérifier qu'elle a besoin ou non d'être paramétrée. Une sous requête non paramétrée signifie que sa valeur ne dépend pas du moment où elle est calculée e.g. une sous requête peut être utilisée pour calculer le salaire moyen. Une sous requête paramétrée signifie que sa valeur dépend de la ligne qui est en train d'être traitée e.g. le fait qu'un idproprietaire donné possède un compte de type Livret A est une sous requête paramétrée. Dans le cas d'une sous requête paramétrée, celle-ci devra avoir une condition de paramétrisation dans la clause `WHERE`. Cette condition de paramétrisation fera intervenir un identifiant de table de la requête extérieure. (Voir exemples plus bas).

5.2.4 Requêtes sur une base de données existante

Après avoir installé le serveur de base de données, chaque élève peut l'interroger via le web. Attention, nous n'avons pas configuré les droits d'accès , ce qui signifie que si vous donnez le login et mot de passe qui a été utilisé pour créer la base, un élève pourrait tout effacer !

L'interrogation se fait en ouvrant la fenêtre de requête (voir Figure 5.7) et en tapant le code de la requête dans cette fenêtre. Les résultats apparaissent dans la fenêtre principale du navigateur.

Questions sans jointure

Pour tous les exercices, on peut demander l'écriture avec le calcul, l'algèbre ou SQL. Notons que pour les requêtes avec des agrégats, on ne peut demander que l'écriture algébrique ou SQL. Dans nos corrections, nous donnons la version SQL, qui correspond à ce qui est demandé dans le programme. Notons qu'il est possible de complexifier les questions à loisir en rajoutant des tables et des conditions de restriction variées. Pour plus de clarté dans la difficulté de chaque exercice, nous nous restreignons à un schéma avec seulement trois tables.

Exercices sans jointure.

Exercice 1

Donnez les nom et prenom de tous les clients de la base.
indication : Ceci est une petite indication

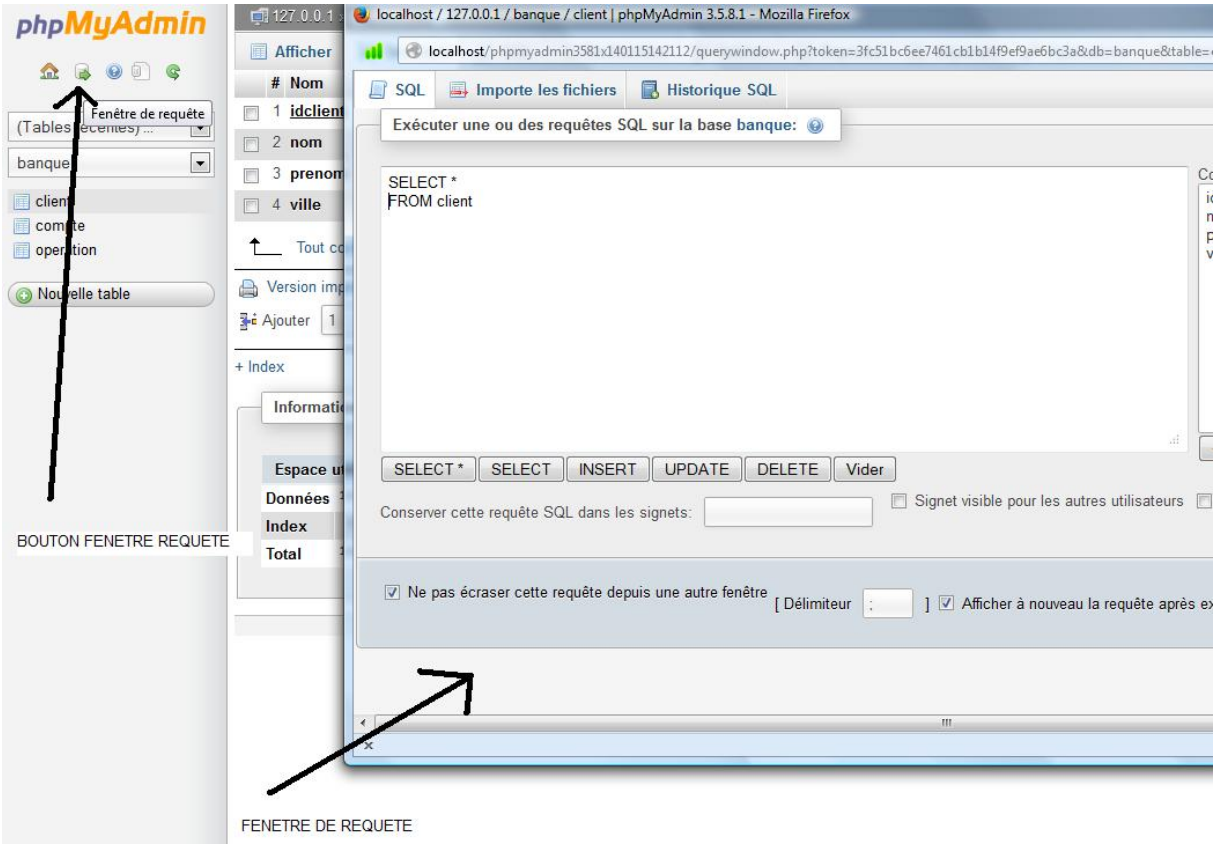

FIGURE 5.7– Ouverture de la fenêtre de requêtes

Exercice 2

Donnez les nom et prenom de tous les clients de la base habitant à “Paris”.

Exercice 3

Donnez les identifiants de tous les comptes de type “Livret A”

Exercice 4

Donnez les idop de type débit sur le compte d’idcompte 1

Exercice 5

Donnez les idproprietaire des personnes possédant un Livret A

Exercice 6

Donnez les idproprietaire des personnes ne possédant aucun Livret A.

Exercices utilisant des fonctions

Exercice 7

Donnez le solde (= somme de toutes les opérations effectuées) de chaque compte

Exercice 8

Donnez le solde de chaque compte, uniquement pour les comptes débiteurs.

Exercice 9

Donnez le solde des comptes d'identifiant compris entre 1 et 10 qui sont débiteurs.

Exercice 10

Calculez le nombre moyen de comptes possédé par un client.

Exercice 11

Affichez pour chaque opération son montant en dollars, en supposant que le taux de conversion soit une constante valant 1.3. Nommez cette colonne `montantUSD`

Exercices mettant en jeu une ou plusieurs jointures.

Exercice 12

Donnez, pour chaque personne définie par son couple nom, prenom, la liste de ses comptes (identifiés par leur idcompte).
indication : On voit ici qu'on a besoin d'afficher des informations présentes dans deux tables : client et compte. Il faut identifier également l'attribut qui va permettre de faire le lien (jointure) entre une ligne de la première table et une ligne de la deuxième.

Exercice 13

Donnez la liste des idcompte qui ont pour propriétaire une personne sont le prénom est "Marie".
indication : Ici, bien qu'on n'ait pas besoin d'afficher des informations de la table client, on souhaite faire un test sur ces données :il faudra donc intégrer cette table à la requête, puisqu'on en a besoin dans la clause WHERE.

Exercice 14

Donnez la liste des opérations (`montant` et `informations`) sur chaque compte de Sylvie Moulin.
indication : Les informations recherchées se trouvent dans la table `operation`, et les valeurs à tester dans la table`client`. Il faut de plus réussir à relier ces informations. Ce n'est possible qu'en utilisant la table `compte`.

Exercice 15

Donnez le nom et prénom des clients ayant fait au moins une opération de débit sur l'un de leurs comptes.
indication : Les informations recherchées se trouvent dans la table `operation`, et les valeurs à tester dans la table`client`. Il faut de plus réussir à relier ces informations. Ce n'est possible qu'en utilisant la table `compte`.

Exercice 16

Donnez pour chaque client identifié par son `nom` et `prenom` le nombre de comptes qu'il possède.
indication : Il s'agit ici de compte le nomdre d'idcompte distinct associé à chaque client.

Exercice 17

Donnez pour chaque type de compte la somme totale disponible. Que faire si un type n'est associé à aucune opération ?

Exercices mettant en jeu une auto-jointure

Exercice 18

Donnez pour chaque `idcompte` la valeur maximale de dépôt et de débit. S'il n'y a aucune valeur de dépôt ou de débit, alors le champ devra être `NULL`.

Exercices mettant en jeu une table ou requête imbriquée pour un calcul annexe

Exercice 19

Donnez l'`idproprietaire` possédant le compte avec le plus grand numéro d'`idcompte`.
indication : Il faut que la requête fonctionne quel que soit le contenu de la base ! Il ne s'agit donc pas de regarder manuellement la plus grande valeur (e.g. 9) et de la mettre en dur dans la requête.

Exercice 20

Donnez les `idproprietaire` des personnes possédant moins de comptes que la moyenne calculée sur l'ensemble de la base de données.
indication : Comme la valeur qu'on veut tester (le nombre de comptes) est un agrégat, il faudra utiliser la clause `HAVING`.

5.3 Corrigés des exercices

Corrigé de l'exercice 1 :

```
SELECT c.nom, c.prenom
FROM client c
```

Corrigé de l'exercice 2 :

```
SELECT nom, prenom
FROM client
WHERE ville = 'Paris'
```

Corrigé de l'exercice 3 :

```
SELECT idcompte
FROM compte
WHERE type = 'Livret A'
```

Corrigé de l'exercice 4 :

```
SELECT idop
FROM operation
WHERE montant < 0
AND idcompte = 1
```

Corrigé de l'exercice 5 :

```
SELECT idproprietaire
FROM compte c1
WHERE c1.type = 'LivretA'
```

Corrigé de l'exercice 6 : Solution 1 : en utilisant ALL

```
SELECT idproprietaire
FROM compte c1
WHERE 'Livret A' <> ALL (
SELECT type
FROM compte c2
WHERE c1.idproprietaire =c2.idproprietaire
)
```

Il faut comprendre que la sous requête est exécutée est exécuté pour chaque ligne de la table compte, et qu'à chaque fois, la sous requête va prendre une valeur différente pour c1.idproprietaire, qui est la *valeur de la ligne courante* ! C'est une sous requête paramétrée. Solution 2 : avec une différence. On construit l'ensemble de tous les idproprietaires, puis l'ensemble des idproprietaires ayant un Livret A et on soustrait.

```
SELECT idproprietaire
FROM compte
EXCEPT
SELECT idproprietaire
FROM compte
WHERE type = 'Livret A'
```

Solution 3 : en utilisant NOT IN (ce qui est une autre manière d'écrire la solution 2, cette fois acceptée par MySQL). Notons que cette sous requête a toujours la même valeur, quel que soit le moment où elle est calculée : c'est une sous requête non-paramétrée.

```
SELECT idproprietaire
FROM compte c1
WHERE c1.idproprietaire NOT IN (
SELECT idproprietaire
FROM compte c2
WHERE c2.type = 'LivretA'
)
```

Corrigé de l'exercice 7 :

```
SELECT idcompte, SUM(montant)
FROM operation
GROUP BY idcompte
```

Corrigé de l'exercice 8 : Il est important de noter que le fait qu'un compte soit débiteur est une condition sur un agrégat de lignes, et s'écrit donc à l'aide de la clause HAVING. La clause WHERE ne peut s'évaluer que sur une seule ligne.

```
SELECT idcompte, SUM(montant)
FROM operation
GROUP BY idcompte
HAVING SUM(montant) < 0
```

Corrigé de l'exercice 9 : Solution 1 : en énumérant les valeurs possibles.

```
SELECT idcompte, SUM(montant)
FROM operation
WHERE idcompte IN (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
GROUP BY idcompte
HAVING SUM(montant) < 0
```

Solution 2 : avec des inégalités.

```
SELECT idcompte, SUM(montant)
FROM operation
WHERE idcompte >= 1
AND idcompte <= 10
GROUP BY idcompte
HAVING SUM(montant) < 0
```

Corrigé de l'exercice 10 :

```
SELECT COUNT(DISTINCT co1.idcompte) / COUNT(DISTINCT co1.
   idproprietaire)
FROM compte co1
```

Corrigé de l'exercice 11 :

```
SELECT op.idop, op.montant1.3 as montantUSDFROM operation op
```

Corrigé de l'exercice 12 :

```
SELECT cl.nom, cl.prenom, co.idcompte
FROM client cl, compte co
WHERE cl.idclient = co.idproprietaire
```

Corrigé de l'exercice 13 :

```
SELECT co.idcompte
FROM client cl, compte co
WHERE cl.idclient = co.idproprietaire
AND cl.prenom = 'Marie'
```

Corrigé de l'exercice 14 :

```
SELECT op.idcompte, op.montant, op.informations
FROM client cl, compte co, operation op
WHERE cl.idclient = co.idproprietaire
AND co.idcompte = op.idcompte
AND cl.prenom = 'Sylvie'
AND cl.nom = 'Moulin'
```

Corrigé de l'exercice 15 :

```
SELECT DISTINCT cl.nom, cl.prenom
FROM client cl, compte co, operation op
WHERE cl.idclient = co.idproprietaire
AND co.idcompte = op.idcompte
AND op.montant < 0
```

Corrigé de l'exercice 16 :

```
SELECT cl.nom, cl.prenom, COUNT(DISTINCT idcompte)
FROM client cl, compte co
WHERE cl.idclient = co.idproprietaire
GROUP BY cl.nom, cl.prenom
```

Corrigé de l'exercice 17 : Solution 1 :

```
SELECT co.type, SUM(op.montant)
FROM compte co, operation op
WHERE co.idcompte = op.idcompte
GROUP BY co.type
```

Cette “solution” présente un problème : si un type de compte n'a aucune opération dessus, il n'apparaitra pas. Par exemple le type Plan Epargne Actions est filtré par la jointure. Il faut donc utiliser une jointure externe (gauche) pour conserver tous les types de comptes. Toutefois la valeur apparaissant sera NULL. Solution 2 :

```
SELECT co.type, SUM(op.montant)
FROM compte co LEFT OUTER JOIN operation op on co.idcompte = op.
   idcompte
GROUP BY co.type
```

Solution 3 : construire une solution en faisant l'union de la solution 1, et d'une autre table qu'on aura construite à la main qui contiendra les types de comptes n'apparaissant pas, et la constante 0.

```
SELECT co.type, SUM(op.montant)
FROM compte co, operation op
WHERE co.idcompte = op.idcompte
GROUP BY co.type
UNION
SELECT co.type, 0
FROM compte co
WHERE co.type NOT IN (
\> SELECT co1.type
\> FROM compte co1, operation op
\> WHERE co1.idcompte = op.idcompte)
```

Corrigé de l'exercice 18 : Idée 1 : Problème : on prend le min et le max algébriques, donc on ne teste pas pour savoir si la valeur est nulle ou pas.

```
SELECT op1.idcompte , MAX(op1.montant), MIN(op2.montant)
FROM operation op1, operation op2
WHERE op1.idop = op2.idop
GROUP BY op1.idcompte
```

Solution 2 : On va faire deux requêtes, l'une qui calcule le MIN sur les valeurs négatives, et l'autre qui calcule le max sur les valeurs positives, puis faire la jointure !

```
SELECT T1.id, T1.maxi, T2.mini
FROM
(SELECT op1.idcompte as id, MAX(op1.montant) as maxi
FROM operation op1
WHERE op1.montant >0
GROUP BY op1.idcompte) AS T1 FULL OUTER JOIN
(SELECT op2.idcompte as id, MIN(op2.montant) as mini
FROM operation op2
WHERE op2.montant <0
GROUP BY op2.idcompte) AS T2 on T1.id = T2.id
```

Seul petit soucis ... le FULL OUTER JOIN n'est pas supporté par MySQL, en tous cas dans sa version actuelle ! (C'est une lacune). Comme LEFT OUTER JOIN et RIGHT OUTER JOIN sont supportés, on peut néanmoins proposer une solution, qui est la suivante : Solution 3 : Une manière de réécrire le FULL OUTER JOIN est de faire l'UNION (ensembliste) du LEFT OUTER JOIN et du RIGHT OUTER JOIN (NB l'union multi-ensembliste est l'opérateur UNION ALL), en prenant bien garder d'aller chercher idcompte dans la bonne table (celle du côté de la jointure externe.)

```
SELECT T1.id, T1.maxi, T2.mini
FROM
(SELECT op1.idcompte as id, MAX(op1.montant) as maxi
FROM operation op1
WHERE op1.montant >0
GROUP BY op1.idcompte) AS T1 LEFT OUTER JOIN
(SELECT op2.idcompte as id, MIN(op2.montant) as mini
FROM operation op2
WHERE op2.montant <0
GROUP BY op2.idcompte) AS T2 on T1.id = T2.id
UNION
SELECT T2.id, T1.maxi, T2.mini
FROM
(SELECT op1.idcompte as id, MAX(op1.montant) as maxi
FROM operation op1
WHERE op1.montant >0
GROUP BY op1.idcompte) AS T1 RIGHT OUTER JOIN
(SELECT op2.idcompte as id, MIN(op2.montant) as mini
FROM operation op2
WHERE op2.montant <0
GROUP BY op2.idcompte) AS T2 on T1.id = T2.id
```

Corrigé de l'exercice 19 :

```
SELECT idproprietaire
FROM compte co
WHERE co.idcompte = (
\> SELECT MAX(co1.idcompte)
\> FROM compte co1)
```

Corrigé de l'exercice 20 :

```
SELECT idproprietaire
FROM compte co
GROUP BY idproprietaire
HAVING COUNT(DISTINCT idcompte) < (
\> SELECT COUNT(DISTINCT co1.idcompte) / COUNT(DISTINCT co1.
   idproprietaire)
\> FROM compte co1)
```

5.4 Exercices hors programme

Bibliographie

[ABB+11] J.-P. Archambault, E. Baccelli, S. Boldo, D. Bouhineau, P. Cégielski, T. Clausen, I. Guessarian, S. Lopès, L. Mounier, B. Nguyen, F. Quessette, A. Rasse, B. Rozoy, C. Timsit, T. Viéville, and J.-M. Vincent. *Introduction à la science informatique pour les enseignants de la discipline au lycée*. CRDP Paris, 2011.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley, 1995. http ://www.webdam.inria.fr/Alice.

[Che76] Peter P.-S. Chen. The Entity-Relationship Model, Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1), 1976.

[Cod70] E.F. Codd. A relational model of Data For Large Shared Data Banks. *Communications of the ACM (CACM)*, 13(6), 1970.