TDLOG séance 7 - bases de données relationnelles

Xavier Clerc – xavier.clerc@enpc.fr

24 novembre 2025

Table des matières

Ce document est sensiblement plus long que ceux des autres séances, car il contient des éléments non présentés en cours mais qui peuvent être utiles pour certains projets. Les éléments à connaître en vue des tests sont ceux qui ont été vus en cours et figurent sur les transparents.

1 Introduction

Ce document présente les concepts fondamentaux des bases de données : le lexique des bases de données relationnelles, leur modèle théorique, ainsi que les éléments de base du langage SQL.

Nous allons commencer par présenter le périmètre et les objectifs des bases de données (section 2). Ensuite, nous proposerons une première vision, intuitive, des bases de données sous forme tabulaire pour définir les concepts essentiels (section 3), puis nous présenterons les logiciels de type SGBD (section 4). La section 5 nous permettra de présenter le modèle théorique sous-jacent aux bases de données relationnelles. Les sections 6 et 7 montreront ensuite comment les différents éléments vus précédemment sont implémentés en pratique.

2 Périmètre et objectifs

Nous présentons dans cette section les origines, ainsi que le domaine et les objectifs des bases de données classiques. Par bases de données "classiques", il faut entendre bases de données "relationnelles". En effet, en dépit de mouvements relativement récents vers d'autres types de bases

de données (*cf.* fin de section), les bases de données dites relationnelles restent globalement majoritaires.

2.1 Les débuts : informatique de gestion

Les bases de données sont essentiellement issues de l'informatique de gestion. C'est-à-dire de logiciels, originellement écrits dans des langages tels que Cobol, qui étaient utilisés dans les entreprises pour la gestion du personnel ou la gestion du stock ou de la production.

Il est difficilement contestable que ces applications d'origine ont joué un rôle que l'on distingue encore aujourd'hui dans les bases de données récentes. D'une part, contrairement à la plupart des langages de programmation, les langages utilisés pour l'interrogation de bases de données sont essentiellement déclaratifs (ce qui signifie que l'on exprime ce que l'on souhaite obtenir et non comment on souhaite l'obtenir). D'autre part, il n'y a pratiquement qu'un seul langage de programmation qui est véritablement utilisé dans le domaine des bases de données relationnelles : SQL.

On pourrait conclure de cet état de fait que les bases de données sont un domaine en faible évolution depuis que les fondations relationnelles ont été mises en place (années 1970). Si cela est superficiellement vrai, les bases de données ont évolué en profondeur : elles ont gagné en efficacité (à la fois en temps et en espace) et le langage SQL a peu à peu intégré de nouvelles constructions. Néanmoins, les principes de base restent inchangés, ce qui est en partie permis par le caractère déclaratif du langage (qui masque complètement l'implémentation sous-jacente) et aussi au fait que les données stockées dans une base ont en général une grande valeur et que les utilisateurs ne souhaitent pas migrer vers d'autres systèmes, si cela n'est pas strictement nécessaire.

2.2 Hier : omniprésence

Plus récemment, l'usage des bases de données s'est massivement répandu, notamment avec l'expansion d'Internet. En effet, pratiquement tout site web qui permet l'inscription d'utilisateurs pour profiter de services est fondé sur une base de données. De plus, de nombreuses applications non Internet sont également fondées sur des bases de données :

- gestion de carnet d'adresses ;
- gestion de calendrier ;
- systèmes de sauvegarde des courriers électroniques ;
- *etc*

Comme le montrent les exemples précédents, les bases de données sont rarement autosuffisantes. Elles agissent le plus souvent comme une sous-partie d'un système plus large. Dans ce cadre, leur utilisation est préférée à de simples fichiers car elles apportent les garanties suivantes :

- gestion des accès simultanés ;
- gestion optimisée de l'espace disque ;
- gestion des pannes et de la reprise sur erreur (pour éviter la corruption des données) ;
- accès aux données à l'aide d'un langage plus simple que les langages de programmation habituels.

2.3 Aujourd'hui : contestation du modèle relationnel

Depuis environ une vingtaine d'années, une nouvelle génération de bases de données émerge, en partie fondée sur les besoins spécifiques liés au *big data*. Ces besoins peuvent grossièrement se résumer à des stockages et transferts de données quelques ordres de grandeur plus importants. En conséquence, certaines opérations au cœur des bases de données relationnelles deviennent déraisonnables dans le cadre du *big data* (*p. ex.* les jointures), et il devient en outre nécessaire de répartir les données sur plusieurs machines. Cette masse de données a également des conséquences indirectes sur le modèle de calcul : il n'est plus raisonnable de considérer que l'on dispose à un instant donné de l'ensemble des données accessibles pour un calcul, et il n'est de même pas possible de bloquer toute modification des données pendant l'exécution d'un calcul.

2.4 Architecture

Le plus souvent, les bases de données sont des programmes autonomes qui se comportent comme des serveurs et auxquels des clients désireux d'accéder aux données se connectent. Ainsi, l'utilisation d'une base de données place le plus souvent dans le cadre d'une architecture de type client-serveur (*cf.* séance sur l'architecture), indépendamment du type de l'application qui joue le rôle de client (*p. ex.* serveur web ou application bureautique).

En outre, la base de données se comportant comme un serveur, elle peut être accédée simultanément par de nombreux clients. Cela est particulièrement utile pour les applications partagées, telles que les applications Internet.

Enfin, dans cette architecture, le langage SQL est utilisé par tous les clients pour accéder aux données lorsqu'elles sont stockées dans une base de données relationnelle. Cela signifie que des clients de natures différentes peuvent facilement y accéder et que, quel que soit le langage utilisé pour développer l'application elle-même, l'accès aux données se fait toujours par le biais du même langage, SQL.

3 Concepts élémentaires

Nous présentons dans cette section, de manière intuitive, l'ensemble des notions servant de fondations aux bases de données relationnelles.

3.1 Représentation tabulaire

Les bases de données sont fréquemment représentées sous forme de tables en deux dimensions. Les tables contiennent les informations de la base de données, organisées de la manière suivante :

- chaque table regroupe les instances d'un type d'entités ;
- chaque ligne d'une table représente une instance du type d'entité correspondant à la table ;
- chaque colonne correspond à un attribut (ou propriété) des instances d'entités correspondant à la table.

Ainsi, la figure Figure 3.1 représente-t-elle une portion d'une base de données sur le cinéma. La table "Films" possède trois colonnes correspondant aux trois attributs d'une instance de "Films". Chaque ligne correspond à un film donné. La table regroupe l'ensemble des films connus de la base de données.

En outre, chaque colonne ou attribut est d'un type donné, ici :

— "Titre" est une chaîne de caractères donnant le titre du film ;
— "Année" est un nombre donnant l'année de sortie du film ;
— "Metteur en scène" est une chaîne de caractères donnant le nom du réalisateur du film.

Titre	Année	Metteur en scène
Le parrain	1972	Francis Ford Coppola
Scarface	1983	Brian de Palma
Scarface	1932	Howard Hawks
(…)		
Inglorious Basterds	2009	Quentin Tarantino

FIGURE 3.1 – Base de données sous forme tabulaire de "Films".

Les lignes et colonnes jouent donc des rôles différents au sein de la base de données. Par référence à la programmation objet, les tables définissent les différentes classes tandis que les lignes sont les instances et les colonnes les attributs des instances.

En conséquence, les colonnes sont relativement stables (*i. e.* les ajouts, suppression et renommages sont peu fréquents) tandis que les lignes sont fréquemment modifiées (ajout, suppression ou modification de valeurs).

Enfin, on notera que les lignes d'une table ne sont pas ordonnées. Pour des raisons pratiques, on les affiche dans un certain ordre, mais aucune garantie n'est donnée sur l'ordre d'affichage des lignes. L'ordre d'affichage des colonnes peut correspondre à celui spécifié lors de la création de la table, mais sans revêtir de caractère sémantique fort.

3.2 Valeurs `NULL` et `unknown`

En bases de données, une valeur particulière permet de dénoter l'absence de valeur dans une case de la table, il s'agit de la valeur `NULL`. Cette valeur est utilisée à la fois pour constater l'absence de valeur pertinente pour un attribut d'une ligne donnée et pour signifier que ladite valeur n'est pas encore connue.

On dit que la valeur `NULL` se propage, au sens où toute expression de calcul dans laquelle elle apparaît a pour valeur `NULL`. En pratique, cela signifie que `NULL` apparaît comme un élément absorbant dans les calculs.

En revanche, lors d'une comparaison, `NULL` n'est pas absorbant mais toute comparaison entre une valeur quelconque et `NULL` a pour résultat `unknown`. Cette valeur `unknown`, à placer au même niveau que les valeurs booléennes `true` et `false`, signifie que la logique manipulée dans les bases de données n'est pas la logique binaire classique mais une logique ternaire.

Attention, la comparaison d'une valeur à `NULL` renvoie toujours `unknown`, y compris si la dite valeur est `NULL`. Ainsi `NULL = NULL` a pour résultat `unknown`. Pour tester si une valeur `v` est égale à (respectivement, différente de) `NULL`, il faut utiliser la forme `v IS NULL` (respectivement, `v IS NOT NULL`).

Bien que relativement peu fréquente, l'utilisation d'une logique ternaire est assez simple. Là où l'on avait deux valeurs de vérité `false` et `true`, on y ajoute une troisième appelée `unknown`. Il existe plusieurs manières de définir les opérations sur les valeurs de vérité dans le cadre d'une logique ternaire. Une possibilité consiste à poser que `false` $<$ `unknown` $<$ `true` (en partant

du principe que l'on a `false` $<$ `true` dans une logique binaire). Dès lors, on peut poser que le *et* logique correspond au minimum et que le *ou* logique correspond au maximum. En SQL, suivant cette idée, les opérations logiques sont définies par les tables de vérité données par les tables 3.1.

or	`true`	`unknown`	`false`
`true`	`true`	`true`	`true`
`unknown`	`true`	`unknown`	`unknown`
`false`	`true`	`unknown`	`false`

and	`true`	`unknown`	`false`
`true`	`true`	`unknown`	`false`
`unknown`	`unknown`	`unknown`	`false`
`false`	`false`	`false`	`false`

TABLE 3.1 – Tables de vérité des *ou* et *et* logiques en SQL.

3.3 Notion de clef primaire

La clef primaire d'une table est une colonne, ou une liste de colonnes, telle que sa valeur permet de déterminer de manière unique la ligne de la table. Dans l'exemple de la figure Figure 3.1, on voit que la clef primaire ne peut être le titre, puisque deux films distincts peuvent avoir le même titre.

En raisonnant sur le contenu possible de la table, il paraît évident que les autres colonnes ne peuvent pas non plus servir de clef primaire ; en effet :

— plusieurs films sont réalisés chaque année ;
— un metteur en scène peut réaliser plusieurs films dans sa carrière.

Il n'existe donc pas pour notre table de clef primaire *simple* (*i. e.* composée d'une seule colonne).

En revanche, il est peu fréquent que deux films de même titre sortent la même année. On pourrait donc utiliser le couple ⟨ Titre, Année ⟩ comme clef primaire *composée* (ou *composite*).

Cependant, pour se prémunir contre la possibilité de deux remakes la même année [1], on va préférer ajouter une nouvelle colonne et attribuer à chaque film un identifiant dont on garantira qu'il est unique.

Cela correspond à la manière dont on procède en pratique :

1. on cherche une clef primaire simple ;
2. s'il n'en existe pas, on cherche une clef primaire composée ;
3. s'il n'en existe pas, on définit une clef arbitraire par ajout d'une colonne.

La figure Figure 3.2 montre la table augmentée d'un identificateur jouant le rôle de clef primaire. La convention veut que l'on souligne la clef primaire afin de la mettre en évidence.

3.4 Notion de clef étrangère

La clef primaire, on l'a dit, permet d'identifier de manière unique une ligne d'une table donnée. La clef étrangère est simplement la référence à une clef primaire depuis une colonne d'une autre table. Supposons que dans notre exemple sur les films, nous souhaitions disposer des titres des films non pas seulement en français, mais dans plusieurs langues. On pourrait envisager d'ajouter des colonnes pour stocker cette information ; cependant, cette solution n'est pas idéale car nous aurions à modifier la structure de la table à chaque fois que nous voudrions gérer une nouvelle

1. Ce qui, par parenthèse, s'est produit en 2011 en France.

Ident	Titre	Année	Metteur en scène
04143	Le parrain	1972	Francis Ford Coppola
01345	Scarface	1983	Brian de Palma
19045	Scarface	1932	Howard Hawks
	(…)		
08432	Inglorious Basterds	2009	Quentin Tarantino

FIGURE 3.2 – Table "Films" avec identifiant explicite.

langue. De plus, quand nous ne disposons pas du titre dans une langue donnée, nous devrions utiliser la valeur `NULL`.

Une meilleure solution est de séparer nos informations en deux tables :

— une table "Films" qui contient l'identifiant, l'année de sortie et le nom du réalisateur ;
— une table "Titres" qui contient pour les films leurs titres dans diverses langues.

La figure Figure 3.3 montre ces deux tables.

table "Films"

Ident	Année	Metteur en scène
04143	1972	Francis Ford Coppola
01345	1983	Brian de Palma
19045	1932	Howard Hawks
	(…)	
08432	2009	Quentin Tarantino

table "Titres"

Ident	Langue	Titre
04143	FR	Le parrain
04143	EN	The Godfather
01345	EN	Scarface
19045	EN	Scarface
	(…)	
08432	EN	Inglorious Basterds

FIGURE 3.3 – Tables "Films" et "Titres" avec clefs primaires.

La table "Films" a, on l'a vu, pour clef primaire l'attribut "Ident". La table "Titres" a quant à elle pour clef primaire le couple d'attributs ⟨ Ident, Langue ⟩. En effet, dans cette seconde table, si l'on connaît à la fois l'identificateur du film et le code de la langue, on détermine de manière unique le titre.

Enfin, "Ident" au sein de la table "Titres" est une clef étrangère, car il s'agit d'une référence à la clef primaire d'une autre table, en l'occurrence "Films". Les deux types de clefs jouent des rôles différents :

— la clef primaire permet d'identifier une ligne au sein d'une table ;
— la clef étrangère permet de faire référence à une ligne d'une autre table.

Une clef étrangère est une forme de *contrainte d'intégrité* (*i. e.* une contrainte garantissant que les tables contiennent des données cohérentes) en ce sens qu'elle garantit qu'on ne fait pas référence depuis une table à une information inexistante dans une autre.

3.5 Notion de schéma

Le schéma d'une base de données est tout simplement sa structure, indépendamment des informations qui y sont stockées. Par rapport aux éléments que nous avons déjà vus, cela signifie que le schéma est la collection des informations que l'on possède sur les tables.

Ainsi, pour chaque table, son nom augmenté de la liste des attributs et de leurs types constitue une partie du schéma. De même, les clefs primaires et étrangères font partie du schéma de la table en ce qu'elles contraignent les contenus possibles.

La figure Figure 3.4 propose une manière graphique de représenter le schéma de la base de données prise en exemple. Les clefs primaires (simples ou composites) voient leurs attributs soulignés, tandis que les clefs étrangères sont matérialisées par une flèche depuis la colonne effectuant la référence jusqu'à la clef primaire référencée.

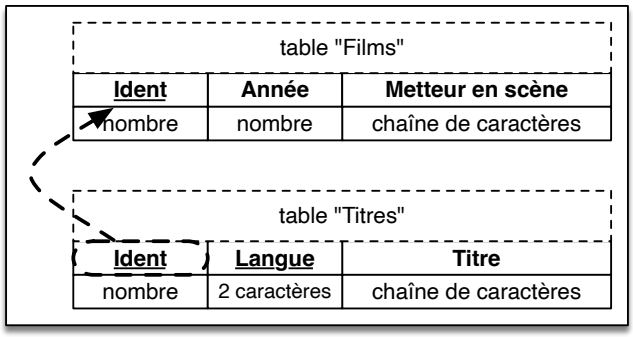

FIGURE 3.4 – Schéma de la base de données sur les films.

3.6 Notion d'index

La notion d'index est centrale dans l'utilisation pratique des bases de données, mais n'a pas d'impact sémantique sur le schéma d'une base de données. Un index est une structure de la base de données utilisée pour accéder plus rapidement aux lignes d'une table en fonction de la valeur d'une colonne (typiquement une clef).

Les index sont créés explicitement par les utilisateurs de la base de données afin d'en améliorer les performances, et souvent implicitement créés pour les clefs primaires. Ils sont souvent incontournables si l'on souhaite obtenir des performances acceptables lors de l'utilisation d'une base de données de grande taille.

Cependant, nous avons décidé de ne pas traiter plus avant la notion d'index ici pour plusieurs raisons :

- ils n'ont pas d'impact sur la sémantique du schéma de la base de données ;
- ils ne modifient pas les résultats obtenus lors de l'interrogation d'une base de données ;
- pour ressentir leur impact en termes de performances sur les machines actuelles, il faut des requêtes mettant en jeu au minimum quelques millions de lignes.

Nous nous contentons donc seulement de souligner leur importance pratique, et le fait qu'ils constituent le levier le plus simple à activer lorsque l'on souhaite optimiser les performances d'une base de données comportant de (très) nombreuses lignes. Pour plus d'informations sur la manière de définir (et de dimensionner) des index, on peut se reporter à l'ouvrage "Bases de données -- Concepts, utilisation et développement" mentionné dans les références.

4 SGBD

4.1 Définition

Les SGBD (pour systèmes de gestion de bases de données) ou SGBDR (pour systèmes de gestion de bases de données relationnelles) sont des logiciels chargés exclusivement du stockage et des accès à une base de données.

Comme on l'a dit, ils sont responsables, outre le stockage, de la gestion des accès simultanés et de la gestion des pannes. Un des points forts des SGBD est qu'ils sont en effet capables d'assurer l'intégrité des données même en cas de panne. Pour cela, ils reposent en général sur deux mécanismes complémentaires :

— des *snapshots*, pris à intervalle régulier qui consistent en une copie du contenu de la base de données ;
— un *log* ou journal, qui enregistre toutes les modifications apportées à la base de données.

Ainsi, en cas de panne, il "suffit" de recharger l'état du dernier *snapshot* et de "rejouer" toutes les modifications enregistrées ultérieurement dans le journal pour retrouver le dernier état de la base de données.

Les SGBD gèrent également les aspects liés à la sécurité de la base de données. Chaque utilisateur de la base de données peut ainsi recevoir des droits de lecture et/ou modifications des tables de la base de données. Cela permet à l'administrateur de la base de données de répartir ces privilèges de la même manière que cela est possible dans un système de fichiers.

Enfin, un intérêt crucial des SGBD est qu'ils permettent l'accès aux bases de données à l'aide d'un langage déclaratif (typiquement le langage SQL). Outre que ce langage est facile à apprendre y compris pour des non-informaticiens, son caractère déclaratif permet aux SGBD d'optimiser de manière transparente les requêtes qui sont faites. En effet, comme l'utilisateur ne fournit qu'une caractérisation du résultat et non la méthode de calcul, le moteur du SGBD est libre d'adopter la stratégie de calcul qui lui semble la plus efficace.

4.2 Principales implémentations

Il existe de nombreuses implémentations de SGBDR. Comme la plupart de ces implémentations utilisent le langage SQL, il est relativement facile de passer d'une implémentation à l'autre sans avoir à réécrire les requêtes portant sur la base de données. De même, la migration des données se fait en général sans difficulté majeure car il est possible (bien que parfois pénible) d'exporter les données d'un SGBD et de les charger dans un autre SGBD.

Implémentations commerciales

Les principaux SGBDR commerciaux sont :

— DB2 d'IBM, disponible sur la plupart des systèmes d'exploitation ;
— Oracle Database d'Oracle, disponible sur la plupart des systèmes d'exploitation ;
— SQL Server de Microsoft, disponible uniquement sous Windows.

DB2 est disponible depuis le début des années 80 et Oracle (Database) depuis la fin des années 70. SQL Server quant à lui n'est disponible que depuis la toute fin des années 80. Des trois, c'est Oracle qui est à la fois le plus connu et souvent considéré comme le plus performant. Il tire ses excellentes performances en particulier de deux éléments :

— une meilleure gestion de l'espace et de l'organisation des disques durs (y compris en passant outre les systèmes de fichiers, en travaillant sur une partition dédiée) ;
— une excellente optimisation des requêtes soumises par l'utilisateur (en particulier, Oracle fonde l'application des optimisations sur des éléments statistiques collectés au fil de l'utilisation).

Enfin, Oracle est connu pour son excellente résistance aux pannes.

Implémentations libres

Les principaux SGBDR libres sont :

— MySQL, aujourd'hui propriété d'Oracle mais distribué sous licence GPL ;
— PostgreSQL, distribué sous licence BSD ;
— SQLite, qui est dans le domaine public [1].

PostgreSQL et MySQL, tous deux lancés au milieu des années 90, se font fait connaître et se sont affrontés lors de l'explosion du nombre de sites Internet et des besoins induits en bases de données (si possible gratuites). Dans ce contexte, MySQL a su tirer son épingle du jeu du fait de meilleures performances et d'une plus grande simplicité d'utilisation. Cependant, ces performances supérieures étaient obtenues en échange de garanties moindres quant à la robustesse du système.

Depuis, MySQL a vu ses performances se rapprocher de celles de PostgreSQL (notamment au fil de l'ajout de fonctionnalités) et les deux solutions sont aujourd'hui essentiellement équivalentes en termes de possibilités et de performances. Cependant, MySQL reste aujourd'hui plus connu, en dépit de son rachat par Oracle qui pose un certain nombre de questions quant à son avenir.

SQLite est très différent de tous les autres SGBD mentionnés ici car il ne fonctionne pas sur un mode client-serveur comme les autres, mais sous forme de librairie directement appelée par un programme écrit en C, Python, *etc* Le principal intérêt de SQLite est, comme son nom l'indique, qu'il est extrêmement léger, à tel point qu'il peut être utilisé dans des applications embarquées (*p. ex.* dans la téléphonie). En revanche, il ne supporte qu'une sous-partie du langage SQL et offre moins de garanties quant à la résistance aux pannes.

5 Modèle de l'algèbre relationnelle

Le modèle relationnel a vu ses fondations théoriques définies dans les années 1970 par Edgar Frank Codd. Cette théorisation des éléments vus à la section 3 repose sur la théorie des ensembles, et est inchangée depuis la proposition initiale. Elle définit ce que l'on appelle l'*algèbre relationnelle*.

1. Ce qui signifie qu'il n'existe de fait aucune restriction à son utilisation.

5.1 Domaine, attribut, tuple et relation

Tout d'abord, Codd a défini rigoureusement les intuitions derrière la représentation tabulaire vue précédemment. Ainsi, chaque colonne ou attribut est associée à un domaine qui correspond à l'ensemble des valeurs qui peuvent apparaître dans la colonne. Pour reprendre notre exemple, on pourra noter Année $\in \mathbb{N}$ pour indiquer que l'année de sortie d'un film est un entier naturel.

Ensuite, une ligne d'une table est vue comme un n-uplet ou tuple d'éléments dont chacun appartient au domaine de la colonne. La première ligne de la table film est ainsi représentée par la valeur :

$$\langle 04143, 1972, "Francis\ Ford\ Coppola" \rangle$$

où :
- $04143 \in \mathbb{N}$;
- $1972 \in \mathbb{N}$;
- $"Francis\ Ford\ Coppola" \in chaines\_caracteres$.

Une table est vue comme une relation au sens mathématique du terme. De manière équivalente, une table peut être vue comme un sous-ensemble du produit cartésien des domaines des attributs, soit :

$$Films \subseteq \mathbb{N} \times \mathbb{N} \times chaines\_caracteres$$

Cela signifie en particulier qu'il ne peut y avoir dans ce modèle de doublon dans une table.

L'usage veut que l'on fasse figurer dans la définition d'une table la liste des attributs avec leurs domaines, en soulignant les clefs primaires. Soit dans notre exemple :
- $Films(\underline{Ident : \mathbb{N}}, Annee : \mathbb{N}, MetteurEnScene : chaines\_caracteres)$;
- $Titres(\underline{Ident : \mathbb{N}, Langue : chaines\_caracteres}, Titre : chaines\_caracteres)$.

La référence à un attribut se fait par son seul nom s'il n'y a pas d'ambiguïté, et en préfixant par le nom de la table s'il y a une ambiguïté (*p. ex.* $Annee$ mais $Films.Ident$).

Enfin, une clef étrangère est simplement une contrainte d'inclusion, soit dans notre exemple $Titres.Ident \subseteq Films.Ident$.

Ces différents éléments suffisent à définir le schéma d'une base de données relationnelle. Nous allons maintenant voir les différentes opérations qui permettent de faire des requêtes sur une base de données afin de répondre aux questions des utilisateurs. On notera que toutes les opérations présentées travaillent sur des ensembles mathématiques et donc ne modifient pas la base de données, elles permettent seulement de faire des calculs sur son contenu.

5.2 Projection

La projection est l'opération qui permet de supprimer et/ou réordonner des colonnes d'une relation. La projection est en général notée $\Pi_{a_0,\ldots,a_n}(R)$ où les a_i sont les noms des attributs que l'on souhaite garder dans le résultat et R la relation sur laquelle on souhaite effectuer la projection.

Par exemple, si l'on souhaite obtenir la liste des metteurs en scène avec les années pour lesquelles ils ont réalisé un film, on écrira $\Pi_{MetteurEnScene,Annee}(Films)$. La figure Figure 5.1 montre le résultat de la projection. La figure Figure 5.2 montre quant à elle le formalisme graphique utilisé pour représenter une projection ; le trapèze symbolise le fait que le résultat est "plus petit en largeur" que l'opérande.

Metteur en scène	Année
Francis Ford Coppola	1972
Brian de Palma	1983
Howard Hawks	1932
(...)	
Quentin Tarantino	2009

FIGURE 5.1 – Exemple de projection de la table "Films".

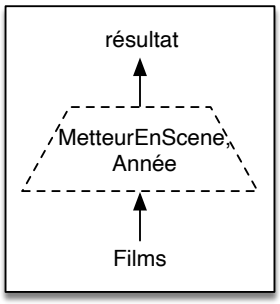

FIGURE 5.2 – Représentation graphique d'une projection.

Attention : les relations ayant une sémantique ensembliste, la projection $\Pi_{Langue}(Titres)$ renverrait $"Fr", "En"$, faisant disparaître tous les doublons.

5.3 Sélection

La sélection est l'opération qui permet de sélectionner des tuples d'une relation. La sélection est en général notée $\sigma_{cond}(R)$ où $cond$ est une condition faisant apparaître des attributs de la relation R et/ou des constantes, liés par des opérateurs de comparaison (typiquement $=$, $\neq$, $<$, $\leq$, $>$, $\geq$) et des opérateurs logiques (typiquement ou, et, non).

Par exemple, si l'on souhaite obtenir la liste des metteurs en scène ayant réalisé un film après l'an 2000, on écrira une combinaison de sélection et de projection : $\Pi_{MetteurEnScene}(\sigma_{Annee \geq 2000}(Films))$. Comme on l'a dit, les éventuels doublons seront éliminés puisque l'on travaille sur des ensembles. Les figures Figure 5.3 et Figure 5.4 montrent respectivement le résultat de ce calcul et le formalisme graphique utilisé pour représenter une sélection (le trapèze symbolise que le résultat est "plus petit en hauteur" que l'opérande).

Metteur en scène
Quentin Tarantino

FIGURE 5.3 – Exemple de sélection et projection de la table "Films".

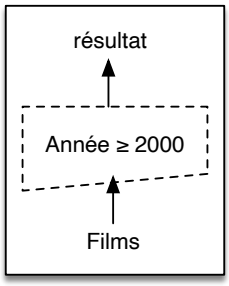

FIGURE 5.4 – Représentation graphique d'une sélection.

5.4 Intersection, union et différence

Puisque les relations sont de simples ensembles, il est naturel de disposer des opérations ensemblistes classiques sur les relations. Ainsi, les opérations d'intersection, d'union et de différence sont-elles définies sur les relations. La seule restriction portant sur leur utilisation est que les deux opérandes d'un tel opérateur doivent être des relations de même type, ce qui signifie que leurs attributs doivent avoir les mêmes domaines, et ce dans le même ordre. Si les relations ont les mêmes domaines à l'ordre près, il suffit d'utiliser une opération de projection pour réordonner les attributs.

Ces opérateurs permettent de répondre à des questions telles que :

1. quels sont les réalisateurs ayant réalisé des films dans les années 70 ou les années 2000 ?
2. quels sont les réalisateurs ayant réalisé des films dans les années 70 et les années 2000 ?
3. quels sont les réalisateurs ayant réalisé des films dans les années 70 mais pas les années 2000 ?

par les opérations suivantes :

1. $\Pi_{MetteurEnScene}(\sigma_{Annee \geq 1970 \ et \ Annee \leq 1979}(Films))$
   $\bigcup \ \Pi_{MetteurEnScene}(\sigma_{Annee \geq 2000 \ et \ Annee \leq 2009}(Films))$
2. $\Pi_{MetteurEnScene}(\sigma_{Annee \geq 1970 \ et \ Annee \leq 1979}(Films))$
   $\bigcap \ \Pi_{MetteurEnScene}(\sigma_{Annee \geq 2000 \ et \ Annee \leq 2009}(Films))$
3. $\Pi_{MetteurEnScene}(\sigma_{Annee \geq 1970 \ et \ Annee \leq 1979}(Films))$
   $- \ \Pi_{MetteurEnScene}(\sigma_{Annee \geq 2000 \ et \ Annee \leq 2009}(Films))$

La figure Figure 5.5 montre le formalisme graphique utilisé pour représenter les opérations ensemblistes.

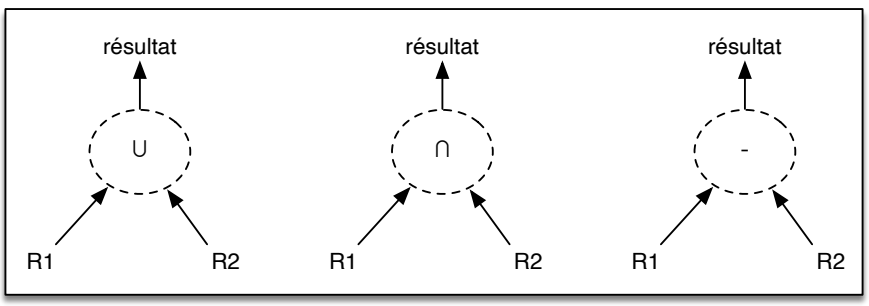

FIGURE 5.5 – Représentation graphique des opérations ensemblistes.

5.5 Produit cartésien

Les relations étant des ensembles, il est sensé de calculer le produit cartésien de deux relations. On notera simplement que le résultat du produit de deux relations $R_a(a_0, \ldots, a_n)$ et $R_b(b_0, \ldots, b_m)$ est une relation de la forme $R(a_0, \ldots, a_n, b_0, \ldots, b_m)$. Cela signifie que la liste des attributs de la relation résultante est la concaténation des listes d'attributs des relations opérandes. La figure Figure 5.6 montre la représentation graphique usuelle du produit cartésien.

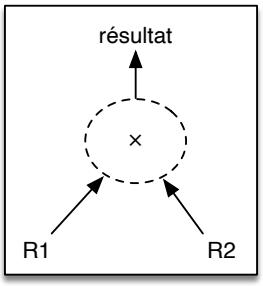

FIGURE 5.6 – Représentation graphique du produit cartésien.

5.6 Jointure

L'opération de jointure est l'opération de prédilection pour faire le lien entre deux tables. Dans notre exemple, nous avons séparé la notion de film de la notion de titre afin de pouvoir stocker facilement des titres en plusieurs langues. Cependant, de nombreuses questions que l'on peut poser à la base de données vont nécessiter de remettre en correspondance les deux relations. Heureusement, nous savons comment les deux tables sont liées, par le biais des clefs primaires et étrangères.

Il manque donc seulement une opération permettant de faire le lien entre les deux tables. En fait, cela est possible à l'aide des opérations que nous avons déjà vues. En effet, on peut calculer le produit cartésien des deux relations $R = Films \times Titres$ où R a pour attributs :

- $Ident \in \mathbb{N}$;
- $Annee \in \mathbb{N}$;
- $MetteurEnScene \in chaines\_caracteres$;
- $Ident' \in \mathbb{N}$ (renommé car il existe déjà un attribut $Ident$) ;
- $Langue \in chaines\_caracteres$;
- $Titre \in chaines\_caracteres$.

Cela donne le résultat présenté à la figure Figure 5.7.

Bien entendu, le produit cartésien contient beaucoup de lignes sans intérêt. Les seules lignes qui nous intéressent sont celles où $Ident = Ident'$. Il suffit de faire une sélection sur R pour obtenir $R' = \sigma_{Ident=Ident'}(R)$, le résultat présenté à la figure Figure 5.8.

Dans la pratique des bases de données, cet enchaînement produit cartésien puis sélection est tellement commun qu'une opération a été définie spécialement. Il s'agit de l'opération de jointure, qui permet de relier deux tables selon une condition. Elle se note :

$$R1 \underset{cond}{\bowtie} R2$$

où $cond$ désigne la condition de jointure. La figure Figure 5.9 montre la représentation graphique de cette opération.

Ident	Année	Metteur en scène	Ident'	Langue	Titre
04143	1972	Francis Ford Coppola	04143	FR	Le parrain
04143	1972	Francis Ford Coppola	04143	EN	The Godfather
04143	1972	Francis Ford Coppola	01345	EN	Scarface
04143	1972	Francis Ford Coppola	19045	EN	Scarface
04143	1972	Francis Ford Coppola	08432	EN	Inglorious Basterds
01345	1983	Brian de Palma	04143	FR	Le parrain
01345	1983	Brian de Palma	04143	EN	The Godfather
01345	1983	Brian de Palma	01345	EN	Scarface
01345	1983	Brian de Palma	19045	EN	Scarface
01345	1983	Brian de Palma	08432	EN	Inglorious Basterds
19045	1932	Howard Hawks	04143	FR	Le parrain
19045	1932	Howard Hawks	04143	EN	The Godfather
19045	1932	Howard Hawks	01345	EN	Scarface
19045	1932	Howard Hawks	19045	EN	Scarface
19045	1932	Howard Hawks	08432	EN	Inglorious Basterds
08432	2009	Quentin Tarantino	04143	FR	Le parrain
08432	2009	Quentin Tarantino	04143	EN	The Godfather
08432	2009	Quentin Tarantino	01345	EN	Scarface
08432	2009	Quentin Tarantino	19045	EN	Scarface
08432	2009	Quentin Tarantino	08432	EN	Inglorious Basterds

(…)

FIGURE 5.7 – Produit cartésien des tables "Films" et "Titres".

Ident	Année	Metteur en scène	Ident'	Langue	Titre
04143	1972	Francis Ford Coppola	04143	FR	Le parrain
04143	1972	Francis Ford Coppola	04143	EN	The Godfather
01345	1983	Brian de Palma	01345	EN	Scarface
19045	1932	Howard Hawks	19045	EN	Scarface
08432	2009	Quentin Tarantino	08432	EN	Inglorious Basterds

(…)

FIGURE 5.8 – Restriction sur le produit cartésien des tables "Films" et "Titres".

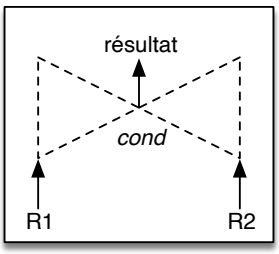

FIGURE 5.9 – Opération de jointure.

Ainsi, la requête permettant de connaître la liste des films avec leurs titres en anglais et les noms des réalisateurs correspondants s'écrit-elle :

$$\Pi_{Titre,MetteurEnScene}(\sigma_{Langue="EN"}(Films \underset{Films.Ident=Titres.Ident}{\bowtie} Titres))$$

5.7 Notion de plan d'exécution

Bien que cette notion ne fasse pas à proprement parler de l'algèbre relationnelle, nous présentons ici la notion de plan d'exécution car elle se comprend facilement lorsque l'on s'appuie sur la représentation graphique qui a été proposée pour les différents opérateurs.

En algèbre relationnelle, on s'intéresse seulement à la manière de combiner les opérateurs précédents pour répondre à des questions relatives aux données contenues dans la base. Dans la pratique, outre cette notion de correction des requêtes, on peut être confronté à des problèmes de performances. Pour diagnostiquer ses performances, on peut demander au SGBD de donner le *plan d'exécution* qu'il utilise pour effectuer une requête.

Un tel plan est essentiellement la combinaison des opérateurs que nous avons vus, sous la forme d'un arbre. Par exemple, si l'on souhaite la liste des titres anglais des films réalisés dans les années 90, on pourra écrire :

$$\Pi_{Titre}(\sigma_{Langue="EN"}(\sigma_{Annee \geq 1990 \ et \ Annee \leq 1999}(Films \bowtie Titres)))$$

qui sera représenté comme à la figure Figure 5.10.

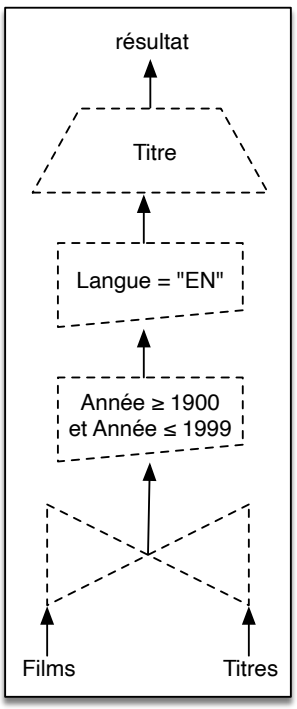

FIGURE 5.10 – Plan d'exécution d'une requête.

Le plan d'exécution d'une requête permet de voir l'enchaînement des opérations effectuées par le SGBD, en partant des données en bas pour aboutir au résultat en haut. Dans notre exemple, on constate que l'opération la plus coûteuse est la jointure car elle correspond à un produit cartésien.

On peut donc chercher à réécrire la requête pour minimiser les tailles des tables en entrée de la jointure. Dans notre cas, cela se fait en déplaçant les sélections avant la jointure :

$$\Pi_{Titre}(\sigma_{Annee \geq 1990 \ et \ Annee \leq 1999}(Films) \bowtie \sigma_{Langue=\text{"}EN\text{"}}(Titres))$$

Cependant, de nombreuses optimisations telles que celle-ci sont effectuées automatiquement par les SGBD. C'est pourquoi il convient de demander au SGBD son plan d'exécution pour une requête afin de déterminer s'il faut ou non réécrire cette dernière. La figure Figure 5.11 présente le plan optimisé de la requête considérée en exemple.

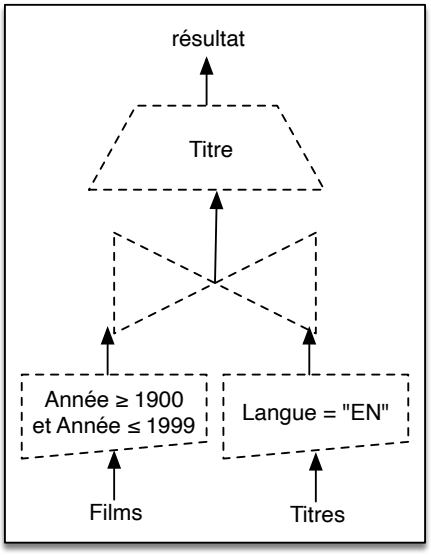

FIGURE 5.11 – Plan optimisé d'exécution d'une requête.

6 SQL : DDL

Le langage SQL (pour structured query language) est le langage d'utilisation commun à l'écrasante majorité des bases de données relationnelles disponibles aujourd'hui. Bien qu'il possède le statut de norme ISO, il est assez fréquent que les différents SGBD ne supportent qu'une fraction du langage, ainsi que des extensions propriétaires. Le langage SQL n'est donc malheureusement pas totalement portable d'un SGBD à l'autre.

Le langage SQL n'est pas sensible à la casse. Pour des raisons de lisibilité, nous avons choisi de faire figurer les mots-clefs du langage entièrement en majuscules et d'utiliser une alternance de majuscules et de minuscules pour désigner les éléments manipulés.

La première sous-partie du langage SQL est le sous-langage DDL, pour Data Definition Language. Comme son nom l'indique, il permet de définir les données (*i. e.* les tables sur lesquelles on va travailler).

6.1 Principaux types disponibles

Les types de données disponibles varient un peu selon le SGBD, mais on trouve en général :

— `SMALLINT` pour les entiers courts ;
— `INT` pour les entiers longs ;

— `FLOAT` ou `REAL` pour les flottants ;
— `CHAR(n)` pour exactement n caractères ;
— `VARCHAR(n)` pour un nombre variable de caractères, au plus n ;
— `DATE` pour une date ;
— `TIME` pour une heure ;
— `TIMESTAMP` pour une heure et une date ;
— `DECIMAL(m, n)` pour un nombre décimal d'au plus m chiffres dont n après la virgule.

6.2 Création de table

La création d'une table se fait simplement en donnant son nom et énumérant entre parenthèses la liste des colonnes avec leurs types, par exemple :

```
CREATE TABLE Films
  (Ident INT,
   Annee INT,
   MetteurEnScene VARCHAR(100));
```

Il est possible de préciser qu'une case ne peut prendre la valeur `NULL` en faisant suivre le type par la mention `NOT NULL`. Il est également possible d'indiquer la clef primaire à l'aide d'une clause `PRIMARY KEY` en fin de déclaration :

```
CREATE TABLE Films
  (Ident INT NOT NULL,
   Annee INT NOT NULL,
   MetteurEnScene VARCHAR(100) NOT NULL,
   PRIMARY KEY (Ident));
```

De même, la déclaration d'une clef étrangère se fait via une clause `FOREIGN KEY`, en indiquant les tables et colonnes référencées :

```
CREATE TABLE Titres
  (Ident INT NOT NULL,
   Langue CHAR(2) NOT NULL,
   Titre VARCHAR(100) NOT NULL,
   PRIMARY KEY (Ident, Langue),
   FOREIGN KEY (Ident) REFERENCES Films(Ident));
```

6.3 Délétion de table

L'effacement d'une table et de toutes les données qu'elle contient se fait simplement par la commande `DROP TABLE` :

```
DROP TABLE Titres;
DROP TABLE Films;
```

6.4 Modification de table

Il est possible d'ajouter ou de retirer des colonnes à une table en utilisant la commande `ALTER TABLE`. L'ajout se fait de la manière suivante :

```
ALTER TABLE Titres ADD COLUMN Budget DECIMAL(12, 2);
```

et le retrait par :

```
ALTER TABLE Titres DROP COLUMN Budget;
```

6.5 Contrainte d'unicité

On peut souhaiter que toutes les valeurs d'une colonne donnée soient différentes deux à deux. Par exemple, dans la base de données sur les films, si l'on définit une colonne pour stocker le classement des films, on veut s'assurer que tous les numéros d'ordre sont différents. Cela se fait simplement en ajoutant une clause `UNIQUE (...)` en fin de déclaration de table :

```
CREATE TABLE Films
  (Ident INT NOT NULL,
   Annee INT NOT NULL,
   MetteurEnScene VARCHAR(100) NOT NULL,
   Classement INT,
   PRIMARY KEY (Ident),
   UNIQUE (Classement));
```

où on laisse volontairement la possibilité d'une valeur `NULL` pour la colonne `Classement` car tous les films ne sont pas classés.

Cette contrainte posée, le SGBD va vérifier à chaque insertion de ligne et modification de la colonne `Classement` que la contrainte d'unicité est vérifiée. Cette vérification est semblable à celle effectuée pour les colonnes apparaissant comme clef primaire, mais la sémantique est différente. En effet, le fait qu'une colonne soit une clef primaire implique une contrainte d'unicité tandis que la réciproque n'est pas vraie. En outre, une contrainte d'unicité supporte la présence de plusieurs valeurs à `NULL`.

Il est également possible d'ajouter une contrainte d'unicité postérieurement à la création de la table en utilisant la construction `ALTER TABLE` de la manière suivante :

```
ALTER TABLE Films ADD UNIQUE(Classement);
```

6.6 Comportement à la suppression et à la modification

On a vu précédemment les contraintes induites par des clefs primaires et étrangères sur les valeurs présentes, modifiées ou ajoutées à une table. Certaines suites d'opérations ne sont pas commutatives au sens où il faut les exécuter dans un certain ordre pour qu'elles soient acceptées par le SGDB. Par exemple, une clef étrangère ne peut exister si la valeur qu'elle référence n'existe pas.

Pour ne pas avoir à réfléchir aux ordres acceptables pour un ensemble d'opérations donné, on peut également prescrire lors de la création des tables le comportement que l'on souhaite obtenir. SQL propose trois modes à la modification ou suppression :

- *no action* est le comportement par défaut qui consiste à refuser une modification ou suppression qui violerait une contrainte d'intégrité ;
- *set null* consiste à remplacer par la valeur `NULL` une référence à une valeur de clef primaire qui disparaît suite à une modification ou suppression ;
- *cascade* consiste à détruire les lignes dont la valeur de clef primaire référencée disparaît suite à une modification ou suppression.

Dans ce dernier mode, le terme *cascade* se justifie par le fait que les modifications induites ne se sont pas qu'à un seul niveau, mais se propagent récursivement *le long des* références entre tables. Ainsi la suppression d'une clef primaire dans une table peut-elle entraîner la suppression d'un nombre arbitraire de lignes dans un nombre arbitraire de tables...

Le choix du comportement s'effectue lors de la déclaration de la table, à la suite de la définition d'une clef étrangère. Ainsi, dans le cadre de notre exemple, nous pouvons déclarer que la suppression d'un film doit entraîner la suppression des lignes correspondantes dans les titres :

```
CREATE TABLE Titres
  (Ident INT NOT NULL,
   Langue CHAR(2) NOT NULL,
   Titre VARCHAR(100) NOT NULL,
   PRIMARY KEY (Ident, Langue),
   FOREIGN KEY (Ident) REFERENCES Films(Ident) ON DELETE CASCADE);
```

Pour choisir un autre mode, il suffit de remplacer le mot `CASCADE` par `NO ACTION` (valeur par défaut) ou `SET NULL`.

Nous avons indiqué que ces modes concernent deux opérations : la modification et la suppression. En conséquence, SQL nous permet de sélectionner des comportements différents selon l'opération. La création de table précédente nous a permis de choisir le comportement à la suppression grâce à la clause `ON DELETE`. Il suffit d'utiliser la clause `ON UPDATE` pour choisir le comportement à la modification.

Dans le cadre de la modification, bien entendu, les modes ne visent plus à savoir si les lignes contenant une clef étrangère doivent elles aussi être supprimées, mais de savoir si leur valeur doit être modifiée à la nouvelle valeur de la clef primaire. Dans notre exemple, on souhaite que la modification de la colonne `Ident` de la table `Films` modifie en conséquence les valeurs correspodantes dans la table `Titres`. On écrira donc :

```
CREATE TABLE Titres
  (Ident INT NOT NULL,
   Langue CHAR(2) NOT NULL,
   Titre VARCHAR(100) NOT NULL,
   PRIMARY KEY (Ident, Langue),
   FOREIGN KEY (Ident) REFERENCES Films(Ident)
     ON DELETE CASCADE
     ON UPDATE CASCADE);
```

Les modes choisis pour la suppression et la modification peuvent tout à fait être différents. Dans

notre exemple, on pourrait vouloir effectuer la propagation lors d'une modification mais placer une valeur NULL lors de la suppression (*p. ex.* pour garder la trace des films supprimés). Cela s'écrirait :

```
CREATE TABLE Titres
  (Ident INT NOT NULL,
   Langue CHAR(2) NOT NULL,
   Titre VARCHAR(100) NOT NULL,
   PRIMARY KEY (Ident, Langue),
   FOREIGN KEY (Ident) REFERENCES Films(Ident)
     ON DELETE SET NULL
     ON UPDATE CASCADE);
```

6.7 Contraintes arbitraires

En plus des contraintes d'unicité et des contraintes liées aux clefs primaires et étrangères, on peut souhaiter restreindre les valeurs possibles dans une colonne donnée. Cette possibilité est très intéressante car elle permet d'exprimer des restrictions sur les types de données. En effet, comme nous l'avons vu précédemment, les types disponibles sont relativement basiques et l'on pourrait souhaiter (dans les programmes en général et dans les bases de données en particulier) indiquer qu'une colonne :

- a pour domaine les entiers ou décimaux positifs (utile typiquement pour des stocks, des prix, ou des années) ;
- qu'une chaîne de caractères a une forme donnée (avec les types, on peut seulement exprimer une contrainte sur sa longueur) ;
- *etc*

En SQL, il est possible d'exprimer une contrainte sur le domaine d'une colonne sous la forme d'une condition telle que celle apparaissant dans la clause WHERE d'une requête SELECT. Cela se fait simplement lors de la déclaration de la table à l'aide d'une clause CHECK :

```
CREATE TABLE T
  (...
   CONSTRAINT nom_contrainte CHECK (cond1),
   CHECK (cond2));
```

ou lors d'une modification de la table :

```
ALTER TABLE T ADD CONSTRAINT nom_contrainte CHECK (cond);
ALTER TABLE T ADD CHECK (cond);
```

L'intérêt de nommer la contrainte est double :

- si la contrainte est violée, le SGBD y fera référence par son nom ce qui simplifiera le diagnostic du problème ;
- il sera possible de supprimer ultérieurement la contrainte à l'aide d'une instruction ALTER TABLE T DROP CONSTRAINT nom_contrainte.

Dans notre exemple, nous pouvons ainsi restreindre le type de la colonne Langue de la manière suivante :

```
CREATE TABLE Titres
  (Ident INT NOT NULL,
   Langue CHAR(2) NOT NULL,
   Titre VARCHAR(100) NOT NULL,
   PRIMARY KEY (Ident, Langue),
   CHECK (Langue IS IN ('DE', 'EN', 'ES', 'FR', 'IT')));
```

pour indiquer que nous ne sommes intéressés que par les titres en français, anglais, italien, espagnol et alllemand.

Il est important de noter que la condition peut éventuellement contenir une requête `SELECT`. Par exemple, si nous disposons d'une table avec les dates de naissance et de décès des différents réalisateurs, nous pouvons ajouter une contrainte indiquant que les années des films doivent être dans l'intervalle entre leur naissance et leur mort [1]. Pour cela nous écririons une requête `SELECT` utilisant les fonctions d'agrégat `MIN` et `MAX` sur une hypothétique table `Biographies`.

6.8 Déclaration de *triggers*

Les *triggers* forment un mécanisme intéressant des SGBD ; cependant, nous allons en présenter le principe sans entrer dans les détails car leur implémentation exacte dépend du SGBD utilisé. Les *triggers* sont des actions qui sont déclenchées lorsqu'un événement remplissant une condition se produit. La création d'un *trigger* se fait de la manière suivante :

```
CREATE TRIGGER nom_trigger
  evenement
  WHEN condition
  BEGIN
    action
  END;
```

où :

- `evenement` caractérise un moment (`BEFORE` ou `AFTER`), une action sur la base de données (*p. ex.* `INSERT`, `UPDATE` ou `DELETE`) et la table concernée ;
- `condition` est une condition portant typiquement sur les lignes impactées par l'événement ;
- `action` est écrit dans un langage dépendant du SGBD utilisé, supportant généralement au moins les instructions classiques de SQL telles que `INSERT` ou `UPDATE`.

Dans notre exemple, si l'on souhaite garder la trace des films supprimés dans une table nommée `FilmsSupprimes`, on pourra écrire un trigger de la forme :

```
CREATE TRIGGER GarderSupprimes
  BEFORE DELETE OF Ident ON Films
  BEGIN
    INSERT INTO FilmsSupprimes VALUES (Ident);
  END;
```

1. En réalité, ce ne serait pas une très bonne idée en termes de modélisation, du fait des publications à titre posthume.

On notera que dans le cadre d'un événement de type `AFTER UPDATE`, il est possible de faire référence aux colonnes sous la forme `old.colonne` et `new.colonne` pour faire référence aux valeurs avant et après modification. Ainsi garder la trace des changements d'identifiants des films se ferait de la manière suivante :

```
CREATE TRIGGER GarderIdentsModifies
  AFTER UPDATE OF Ident ON Films
  BEGIN
    INSERT INTO FilmsModifies VALUES (old.Ident, new.Ident);
  END;
```

6.9 Les *BLOBS* et *CLOBS*

Les BLOBS et CLOBS sont des types de données. Ils ne modifient en rien le fonctionnement des bases de données tel que nous l'avons décrit mais permettent de stocker des données quelconques, possiblement de (très) grande taille. En effet, jusqu'ici, nous n'avons présenté que des types simples (entiers, flottants, chaînes de caractères, *etc*) mais l'on peut souhaiter stocker dans une base de données le contenu d'un fichier entier. Par exemple, on peut vouloir stocker des images ou encore des textes de longueur arbitraire (dépassant les limites de `VARCHAR`).

Les BLOBS (pour Binary Large Objects) permettent de stocker une suite d'octets quelconque et sont donc utilisés pour stocker n'importe quel type de fichier (image, son, vidéo, voire programmes).

Les CLOBS (pour character Large Objects) permettent de stocker une suite de caractères quelconque et sont donc utilisés pour stocker efficacement du texte long sur lequel on n'envisage pas de formuler des conditions (par opposition aux types `CHAR` et `VARCHAR`).

Ainsi, pour stocker l'affiche d'un film et son synopsis, on pourra modifier la table `Films` de la manière suivante :

```
ALTER TABLE Films ADD COLUMN Affiche BLOB;
ALTER TABLE Films ADD COLUMN Synopsis CLOB;
```

Le seul problème avec ces types de données est que leur modification et leur accès dépend du SGBD. Typiquement, ce dernier fournit des fonctions pour charger et extraire les BLOBS depuis et vers des fichiers. Les CLOBS peuvent en général quant à eux être manipulés comme des chaînes de caractères (les différents SGBD imposent cependant parfois des restrictions).

7 SQL : DML

Nous venons de voir comment le sous-langage DDL de SQL permet de définir des tables. Nous allons maintenant présenter les constructions qui permettent de manipuler les lignes de ces tables ; cela se fait à l'aide du sous-langage DML, pour Data Manipulation Language.

Attention : il existe une différence majeure entre le modèle relationnel théorique présenté et les implémentations fondées sur le langage SQL. En algèbre relationnelle, les lignes sont des n-uplets ou tuples appartenant à l'ensemble représentant la relation. En SQL, il peut en revanche y avoir plusieurs lignes qui contiennent les mêmes données. On n'a donc pas affaire à des ensembles mais à des listes de tuples.

7.1 Requête SELECT

Pour effectuer une interrogation sur la base de données, on utilise la construction SELECT qui est une combinaison de la projection, de la sélection, et du produit cartésien de l'algèbre relationnelle. Sa forme la plus courante est SELECT *attributs* FROM *tables* WHERE *condition*.

La liste *tables* contient, séparés par des virgules, les noms des tables utilisées pour le produit/-sélection/projection. Il est également possible de faire suivre une table par un nom court qui sera utilisé dans les autres parties de la requête.

L'expression *condition* doit être une expression logique et permet de choisir les lignes à conserver lors de la sélection. Si aucune condition n'est spécifiée, c'est le produit cartésien des tables qui va être renvoyé. La condition sert donc à la fois pour exprimer une sélection à proprement parler et pour donner les conditions d'éventuelles jointures entre tables.

La liste *attributs* contient, séparés par des virgules, les noms des attributs que l'on souhaite conserver lors de la projection. S'il y a ambiguïté, on peut préfixer à l'aide du nom de la table ou de l'éventuel nom court évoqué précédemment pour lever l'ambiguïté. Si l'on souhaite conserver tous les attributs lors de la projection, il suffit d'utiliser le caractère *.

Par exemple, pour afficher la liste des titres anglais de notre base de données, on pourra écrire :

```
SELECT Titre FROM Films F, Titres T
  WHERE F.Ident = T.Ident
    AND Langue = 'EN';
```

Comme SQL ne manipule pas des ensembles, le titre "Scarface" apparaîtra deux fois. Si l'on souhaite forcer au plus une apparition de chaque valeur, on doit ajouter la clause DISTINCT après SELECT, soit :

```
SELECT DISTINCT Titre FROM Films F, Titres T
  WHERE F.Ident = T.Ident
    AND Langue = 'EN';
```

7.2 Conditions

Dans les conditions, on peut faire apparaître des constantes entières, décimales et chaînes de caractères en plus des noms d'attributs. Ces valeurs peuvent être mises en relation à l'aide des opérateurs de comparaison =, <>, <=, <, >=, >. Il est également possible d'écrire $x >= m \; et \; x <= p$ sous la forme `x BETWEEN m AND p`.

Lorsque l'on travaille avec des chaînes de caractères, il est possible d'utiliser la fonction CHAR_LENGTH pour obtenir le nombre de caractères de la chaîne. De plus, il est possible d'utiliser l'opérateur LIKE qui considère le caractère _ comme pouvant remplacer n'importe quel autre et % comme pouvant remplacer une sous-chaîne quelconque. Ainsi l'expression `x LIKE "%Ford%"` vaut-elle vrai pour toute chaîne de caractère x contenant la sous-chaîne `"Ford"`.

Enfin, pour comparer une valeur x à NULL, il faut utiliser `x IS NULL` ou `x IS NOT NULL`.

7.3 Jointures explicites

Jusqu'ici, nous avons exprimé les jointures à l'aide d'une clause `WHERE`. Ce faisant, nous avons tiré parti de l'égalité $R1 \underset{cond}{\bowtie} R2 = \sigma_{cond}(R1 \times R2)$ pour écrire les jointures sous la forme :

```
SELECT * FROM Films F, Titres T
  WHERE F.Ident = T.Ident;
```

qui est bien un produit cartésien suivi d'une sélection.

Les SGBD reconnaissent automatiquement de telles conditions comme correspondant à des jointures. Cependant, pour des raisons de lisibilité, on peut souhaiter indiquer explicitement que l'on effectue une opération de jointure. On écrira alors :

```
SELECT * FROM Films F JOIN Titres T ON F.Ident = T.Ident;
```

Dans le cas d'une jointure naturelle (c'est-à-dire que les colonnes ont le même nom dans les deux tables), on pourra même simplifier l'écriture de la manière suivante :

```
SELECT * FROM Films NATURAL JOIN Titres;
```

7.4 Intersection, union et différence

Les opérateur d'intersection, d'union et de différence se nomment respectivement *INTERSECT*, *UNION* et *EXCEPT* (ou *MINUS*). De tels opérateurs sont typiquement placés entre deux requêtes `SELECT`.

Ainsi pour répondre aux questions :

- quels sont les réalisateurs ayant réalisé des films dans les années 70 ou les années 2000 ?
- quels sont les réalisateurs ayant réalisé des films dans les années 70 et les années 2000 ?
- quels sont les réalisateurs ayant réalisé des films dans les années 70 mais pas les années 2000 ?

on écrira les requêtes SQL :

```
(SELECT DISTINCT MetteurEnScene FROM Films WHERE Annee BETWEEN 1970␣
↪AND 1979)
  UNION
(SELECT DISTINCT MetteurEnScene FROM Films WHERE Annee BETWEEN 2000␣
↪AND 2009);

(SELECT DISTINCT MetteurEnScene FROM Films WHERE Annee BETWEEN 1970␣
↪AND 1979)
  INTERSECT
(SELECT DISTINCT MetteurEnScene FROM Films WHERE Annee BETWEEN 2000␣
↪AND 2009);

(SELECT DISTINCT MetteurEnScene FROM Films WHERE Annee BETWEEN 1970␣
↪AND 1979)
  MINUS
```

(suite sur la page suivante)

(suite de la page précédente)

```
(SELECT DISTINCT MetteurEnScene FROM Films WHERE Annee BETWEEN 2000␣
↪AND 2009);
```

7.5 Spécification d'un ordre

Si l'on souhaite obtenir les réponses à une requête dans un certain ordre, il suffit d'ajouter en fin de construction `SELECT` une clause `ORDER BY` suivie de la liste des attributs sur lesquels effectuer le tri. Le tri s'effectue de manière lexicographique sur la liste donnée. Pour chaque attribut, on peut spécifier si le tri doit s'effectuer de manière croissante (`ASC`, le sens par défaut) ou décroissante (`DESC`). Ainsi, si l'on veut la liste des titres de films en français par ordre alphabétique, on écrira la requête :

```
SELECT DISTINCT Titre FROM Titres
  WHERE Langue = 'FR'
  ORDER BY Titre ASC;
```

7.6 Fonctions d'agrégat

Les constructions que nous avons vues jusqu'ici ne sont pas suffisantes pour répondre à des questions mettant en relation plusieurs lignes. Les fonctions d'agrégat permettent de calculer une valeur pour une colonne à partir de plusieurs lignes. Les fonctions les plus courantes sont la somme (`SUM`), le minimum (`MIN`), le maximum (`MAX`), la moyenne (`AVG`) et le comptage (`COUNT`).

Par exemple, si l'on souhaite connaître le nombre de films dans la base de données, on écrira :

```
SELECT COUNT(Ident) FROM Films;
```

ou

```
SELECT COUNT(*) FROM Films;
```

Si l'on souhaite compter le nombre d'occurrences distinctes, par exemple pour connaître le nombre de réalisateurs, on utilisera `DISTINCT` ;

```
SELECT COUNT(DISTINCT MetteurEnScene) FROM Films;
```

7.7 Sous-requêtes

Les fonctions d'agrégat sont souvent combinées avec des sous-requête. Une sous-requête est simplement une requête de la forme `SELECT ... FROM ... WHERE ...` qui est utilisée comme valeur, par exemple dans une condition. Si la sous-requête renvoie une et une seule valeur, son résultat peut être comparé à l'aide des opérateurs habituels. En revanche, si la sous-requête peut renvoyer un nombre quelconque de valeurs, il faut utiliser des opérateurs spéciaux tels que `IN` et `NOT IN` (qui ont la même signification qu'en Python).

Supposons que l'on souhaite, pour un réalisateur de nom n obtenir le titre de son premier film. On commence par calculer l'année de son premier film de manière suivante :

```
SELECT MIN(Annee) FROM Films WHERE MetteurEnScene = n;
```

on peut alors calculer l'identifiant de ce film grâce à la requête :

```
SELECT Ident FROM Films
  WHERE Annee = (SELECT MIN(Annee) FROM Films WHERE MetteurEnScene =␣
↪n);
```

et il ne reste plus qu'à faire la jointure pour obtenir le(s) titre(s) :

```
SELECT Titre FROM Films F, Titres T
  WHERE Annee = (SELECT MIN(Annee) FROM Films WHERE MetteurEnScene = n)
    AND F.Ident = T.Ident
    AND T.Langue = 'EN';
```

On notera que les fonctions `SUM`, `MIN`, `MAX` et `AVG` appelées sur des ensembles vides renvoient la valeur `NULL`.

7.8 Instruction `INSERT`

Jusqu'ici, nous n'avons vu que des constructions qui permettent de poser des questions sur l'état de la base de données (c'est pourquoi on les appelle aussi requêtes). Nous allons maintenant voir des constructions qui permettent de modifier l'état de la base de données.

Pour insérer des données dans une table, on utilise la construction `INSERT`. Le SGBD n'acceptera bien entendu l'ajout que s'il est cohérent avec les contraintes de types et de clefs portant sur les colonnes.

L'instruction d'insertion s'écrit `INSERT INTO` *table* `VALUES (...)` où l'ellipse correspond aux valeurs du tuple à insérer. Pour ajouter le film "Snake Eyes" à notre base de données, on exécutera les deux instructions suivantes :

```
INSERT INTO Films VALUES (6534, 1998, 'Brian de Palma');
INSERT INTO Titres VALUES (6534, 'EN', 'Snake Eyes');
```

On notera qu'il n'est pas possible d'effectuer les insertions dans l'ordre inverse. En effet, l'attribut $Ident$ de la table $Titres$ est une clef étrangère référençant l'attribut $Ident$ de la table $Films$. En conséquence, le SGBD n'acceptera pas d'insertion dans $Titres$ d'un tuple dont l'identifiant n'existe pas dans la table $Films$.

Il est également possible d'ajouter un ensemble de lignes en une seule opération en utilisant une autre forme de l'instruction `INSERT`. Par exemple, si l'on crée une table avec tous les films des années 80, il suffit d'écrire :

```
CREATE TABLE Films80
  (Ident INT NOT NULL,
   Annee INT NOT NULL,
   MetteurEnScene VARCHAR(100) NOT NULL,
   PRIMARY KEY (Ident));
```

(suite sur la page suivante)

(suite de la page précédente)

```
INSERT INTO Films80
  SELECT * FROM Films WHERE Annee BETWEEN 1980 AND 1989;
```

7.9 Instruction DELETE

Pour supprimer des lignes d'une table, on a recours à l'instruction DELETE. Il suffit d'indiquer la table dont on souhaite supprimer des tuples ainsi que la condition de suppression. Par exemple, pour supprimer tous les films antérieurs à 1950, on écrira :

```
DELETE FROM Films WHERE Annee < 1950;
```

Cependant, le SGBD n'acceptera pas cette suppression car elle aurait pour conséquence que des identifiants de la table $Titres$ ne feraient plus référence à des identifiants de la table $Films$ (sauf bien entendu si l'on a demandé la délétion en cascade). En conséquence, on commencera pas supprimer les lignes concernées de la table $Titres$ à l'aide de l'instruction suivante :

```
DELETE FROM Titres
  WHERE Ident IN (SELECT Ident FROM Films WHERE Annee < 1950);
```

Il est également possible de d'effacer toutes lignes de la table par la commande TRUNCATE TABLE :

```
TRUNCATE TABLE Titres;
TRUNCATE TABLE Films;
```

7.10 Instruction UPDATE

On peut souhaiter modifier seulement certains éléments d'une table plutôt que les supprimer ou en ajouter de nouveaux. Ce rôle échoit à la construction UPDATE qui prend la forme UPDATE $table$ SET $affectations$ WHERE $condition$. Seules les lignes satisfaisant la condition sont mises à jour par les $affectations$ qui sont une liste d'éléments de la forme $attribut$ = $valeur$ séparés par des virgules.

Il est bien entendu possible dans $valeur$ de faire référence à l'ancienne valeur d'un attribut par son nom. Ainsi, si l'on souhaite passer en majuscules tous les titres de films en anglais, on écrira :

```
UPDATE Titres
  SET Titre = UPPER(Titre)
  WHERE Langue = 'EN';
```

7.11 Renommage d'une colonne

Il ne s'agit pas à proprement parler d'une nouvelle forme de requête, seulement de la possibilité d'effectuer un renommage à des fins d'affichage. Ainsi, il est possible de faire suivre le nom d'une

colonne après le mot-clef `SELECT` d'une clause `AS` donnant le nom que l'on souhaite utiliser. Ainsi, on pourra écrire :

```
SELECT Annee AS Annee_de_sortie, MetteurEnScene AS Realisateur
  FROM Films;
```

7.12 Requêtes de regroupement

Nous avons vu ci-dessus les fonctions d'agrégat qui permettent d'effectuer des calculs sur plusieurs lignes. Les fonctions d'agrégat les plus courantes sont la somme (`SUM`), le minimum (`MIN`), le maximum (`MAX`), la moyenne (`AVG`) et le comptage (`COUNT`).

Il est également possible dans une requête de demander le regroupement des lignes en fonction des valeurs d'une colonne grâce à une clause `GROUP BY` suivie d'une liste de noms de colonnes. Ainsi, si l'on effectue sur la base de données présentée à la figure Figure 7.1 la requête suivante :

```
SELECT * FROM Films GROUP BY MetteurEnScene;
```

on obtiendra le résultat montré à la figure Figure 7.2.

table "Films"

Ident	Année	Metteur en scène
04143	1972	Francis Ford Coppola
01345	1983	Brian de Palma
19045	1932	Howard Hawks
(…)		
08432	2009	Quentin Tarantino
09432	1998	Brian de Palma
100473	2003	Quentin Tarantino

table "Titres"

Ident	Langue	Titre
04143	FR	Le parrain
04143	EN	The Godfather
01345	EN	Scarface
19045	EN	Scarface
(…)		
08432	EN	Inglorious Basterds
09432	EN	Snake Eyes
100473	EN	Kill Bill

FIGURE 7.1 – Extrait de la base de données.

Les fonctions d'agrégat se comportent différemment lorsque l'on utilise une clause de regroupement : au lieu d'effectuer le calcul sur l'ensemble des lignes, elles effectuent les calculs sur chacun des groupes formés. Ainsi, la requête suivante permet-elle de compter le nombre de films pour chaque réalisateur (*cf.* Figure 7.3) :

Ident	Année	Metteur en scène
04143	1972	Francis Ford Coppola
01345	1983	Brian de Palma
09432	1998	Brian de Palma
19045	1932	Howard Hawks
08432	2009	Quentin Tarantino
100473	2003	Quentin Tarantino

FIGURE 7.2 – Regroupement selon le nom du réalisateur.

```
SELECT MetteurEnScene AS Realisateur, COUNT(Ident) AS Nb_films
  FROM Films
  GROUP BY MetteurEnScene;
```

Realisateur	Nb_films
Francis Ford Coppola	1
Brian de Palma	2
Howard Hawks	1
Quentin Tarantino	2

FIGURE 7.3 – Nombre de films par réalisateur.

Enfin, il est possible de poser des conditions sur les regroupements grâce à la clause `HAVING`. La clause `HAVING` se comporte sur les regroupements comme la clause `WHERE` se comporte sur les lignes. Ainsi, pour obtenir la liste des réalisateurs ayant mis en scène plus d'un film, on écrira la requête suivante (*cf.* Figure 7.4 pour le résultat) :

```
SELECT MetteurEnScene AS Realisateur
  FROM Films
  GROUP BY MetteurEnScene
  HAVING COUNT(Ident) > 1;
```

Realisateur
Brian de Palma
Quentin Tarantino

FIGURE 7.4 – Réalisateurs ayant mis en scène plus d'un film.

7.13 Produit cartésien

Le produit cartésien entre n tables s'effectue simplement en listant les tables derrière le mot-clef `FROM`, sans utiliser de clause `WHERE`. Ainsi, le produit cartésien de tables $T_1, \ldots, T_n$ s'écrit-il :

```
SELECT * FROM T1, ..., Tn;
```

Il est bien sûr possible de renommer des colonnes pour éviter des synonymes, mais cela oblige à donner explicitement la liste des colonnes en lieu et place du caractère "*" :

```
SELECT T1.C1 as A_1_1, ..., Tn.Cm as A_n_m FROM T1, ..., Tn;
```

7.14 Notion de transaction

La notion de transaction est cruciale en bases de données. Comme nous l'avons déjà dit, l'un des objectifs des bases de données est de permettre de nombreux accès simultanés aux données. Or, si des accès concurrents pour lire des données ne posent pas de problèmes autres que de performance, des accès concurrents qui modifient les données peuvent rendre la base de données incohérente s'ils sont mal traités.

L'exemple canonique de transaction est celui d'un transfert bancaire. Pour assurer que le transfert s'effectue correctement il faut modifier deux lignes de comptes (pour le crédit/débit). Cela passera par deux instructions `UPDATE` et il faut absolument être assuré que les deux opérations auront lieu "comme une seule", c'est à dire sans interruption par un autre traitement. De plus, en cas de panne on veut garantir que l'on a exécuté les deux instructions ou aucune pour être certain que le montant total géré est inchangé par le transfert.

Dans le domaine des bases de données, on dit que les transactions doivent respecter quatre propriétés désignées par l'acronyme ACID, pour :

- Atomicité ;
- coherence ;
- Isolation ;
- Durabilité.

L'atomicité correspond à l'idée que la transaction est exécutée soit entièrement soit pas du tout. Typiquement, cela signifie qu'en cas de panne au milieu d'une transaction, c'est l'état antérieur au début de la transaction qui sera restauré par le SGBD.

La cohérence correspond à l'idée que, quoi qu'il arrive (erreur de l'utilisateur ou panne), les contraintes d'intégrité seront maintenues. Ainsi, si une transaction échoue du fait d'une *no action* lors d'une suppression, toutes les modifications faites depuis le début de la transaction sont *déjouées*. En conséquence, on se retrouvera dans l'état qui précédait la transaction.

L'isolation correspond à l'idée que l'utilisateur de la base de données a seulement à indiquer les début et fin de chaque transaction. Il peut alors considérer que chaque transaction s'exécute indépendamment des autres. La charge d'ordonnancer au mieux les transactions et de gérer leur possible entrelacement revient au SGBD.

La durabilité correspond à l'idée que si la transaction a été exécutée, alors toute panne survenant mènera à une restauration par le SGBD d'un état postérieur à la transaction.

Les transactions garantissent donc un certain nombre de propriétés cruciales pour la qualité et la pérennité des données. L'utilisateur de la base de données a seulement à marquer :

- le début d'une transaction (en général [1] par `BEGIN_TRANSACTION`) ;
- la fin d'une transaction (en général par `COMMIT_TRANSACTION`) ;
- l'échec explicite d'une transaction [2] (en général par `ABORT_TRANSACTION` ou `ROLLBACK_TRANSACTION`).

1. En effet, la syntaxe est variable selon le SGBD.
2. Par exemple, on peut avoir un *trigger* qui sous certaines conditions annule la transaction en cours.

Annexes

À retenir

Il est important de retenir les notions de clef primaire, de clef étrangère et de valeur `NULL`. La connaissance des bases de l'algèbre relationnelle (domaine, attribut, tuple, relation) et de ses opérations primitives (projection, sélection, jointure) fournit le modèle fondamental des bases de données relationnelles.

Concernant SQL, il est utile de savoir comment créer et supprimer une table, et d'être capable d'écrire des requêtes simples, avec des jointures, voire utilisant des fonctions d'agrégat. La notion de jointure externe est en revanche accessoire.

Références

— "Bases de données - Concepts, utilisation et développement" (Jean-Luc Hainaut, Dunod) propose une introduction complète aux divers aspects des bases de données relationnelles.
— "Algèbre relationnelle - Guide pratique de conception d'une base de données relationnelle normalisée" (Michelle Clouse, ENI éditions) propose une vision complète de l'algèbre relationnelle et de la modélisation.